Testing Self-Adaptive Software: Requirement Analysis and Solution Scheme

Georg Püschel, Sebastian Götz, Claas Wilke, Christian Piechnick, and Uwe Aßmann
Software Technology Group, Technische Universität Dresden
Email: {georg.pueschel, sebastian.goetz1, claas.wilke, christian.piechnick, uwe.assmann}@tu-dresden.de

***Abstract*—Self-adaptive software reconfigures automatically at run-time in order to react to environmental changes and fulfill its specified goals. Thereby, the system runs in a feedback loop which includes monitoring, analysis, adaptation planning, and execution. To assure functional correctness and non-functional adequacy, verification and validation is required. Hence, the feedback loop's tasks have to be examined as well as the adapted system behavior that spans a much more complex decision space than traditional 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. Alternatively, the models can be executed directly for which simulation-based validation can be employed. For both methods, an engineer has to specify validation models expressing the system's externally perceivable behavior as well as expectations derived from requirements. In this paper, we perform a Failure Mode and Effects Analysis on a generic perspective on self-adaptive software in order to derive additional requirements to be coped within test modeling. Besides functional requirements, we discuss non-functional requirements in particular. From these requirements, a reference solution scheme is derived that can be used to construct and evaluate validation methods for self-adaptive software. For illustration, we provide an example from the home robotics domain.**

***Keywords*—Self-adaptive Software; Model-based Testing; Simulation; Failure Modes and Effects Analysis.**

I. Introduction

In our original work [1], we studied requirements that have to be coped with in testing self-adaptive software (SAS, [2]). This certain kind of system reconfigures automatically at run-time according to sensed context changes. Thus, it is able to effectively and efficiently fulfill its specified goals under changing conditions. For instance, a beneficial application area of SAS are Cyber-physical Systems (CPS, [3]). CPS reflect physical objects, software, and their interplay in order to reason about them. Due to the SAS's ability to automatically adapt to changes in such a context, a CPS may operate autonomously without the requirement of a strictly controlled factory environment. Thus, the development of sophisticated methods and technologies for developing SAS may help to make systems like autonomous cars and home robotic systems become reality.

The user of an SAS can delegate tasks to the system at run-time. Such tasks do not have to be known to the system in advance. Several systems support descriptive formats like goal models or rules for this purpose. Furthermore, there may be unanticipated events in the environment that have to be considered in the SAS's decision process as well as external adaptation mechanisms that change the system structure in an unforeseen manner. In consequence, an SAS engineer has to be aware of several unpredictable behaviors that may impact design decisions.

However, the manufacturer of a system has to give promises to customers about its correctness (e.g., in form of a certificate). Therefore, at least a subset of the SAS capabilities have to be verified or validated before delivery. As each step of the development life cycle is equipped with a limited budget, it is difficult or even impossible to examine the system's correctness completely. Thus, a more scenario-based examination of the SAS is preferable. Additionally, self-testing [4] or even self-verification [5] mechanisms can be built in the system and triggered at the point in time when the system is adapted.

Another problem is that SAS engineers face additional complexity. During each test scenarios' workflow, the test steps can be adapted to changing context situations. In consequence, in the worst case, the state space of the system is combinatorial between adaptation state and workflow state [6, p. 17]. Furthermore, it has to be considered that a context change, that causes an adaptation, has to be taken into account in order to reproduce specific adaptation states. Due to all these complexity factors, the difficulty of applying verification methods like model checking increases enormously. Hence, in our work, we instead focus on determining validation techniques that provide appropriate abstraction means for SAS' state spaces.

The most abstract view on a system can be taken, if it is considered a black box. That is the system's internals are invisible to the tester, except for service interfaces which allow to interact with the black box. These interfaces provide a set of methods that may accept or produce messages and can be used according to a protocol. Instead of examining the internal state of the system, only the observable external state, which is given by the service protocol, is examined [7]. To validate the system, it has to be checked whether the expectations on the interface interaction hold. This can be achieved, e.g., by running test cases. In order to face the SAS's behavioral complexity, these test cases do not have to be designed manually. The state space can automatically be searched for appropriate test cases, if the protocol is represented as a formal model. Test case generation from models is typically called Model-based Testing (MBT, [8]) and has been subject to recent research.

In general, a test case is a sequence of actions of two different types. Firstly, there are actions that produce certain messages to the system under test (SUT) in order to *enforce* (i.e., reproduce) a certain state of the protocol. Secondly, *assertions* retrieve messages from the system in order to verify their data against the expected values. Hence, this data has to be computed by a so-called test *oracle*, which is a mapping function from input to output data. Both, the values to be enforced and assertions to be validated, are central entities of a test model.

Before the test run, the environment of the SUT has to be properly set up. In order to validate SAS, changes have to be applied to this test environment such that self-adaptation is triggered. However, some of the environment's properties are not always controllable with acceptable effort. For instance, changing the weather in outdoor scenarios may only be feasible by mocking sensor data. A related problem constitutes when the test model is not precise enough to predict exact reactions of the SUT. For instance, a robot that is controlled by an SAS does not drive very precisely to a predicted place. The decision logic of the SAS under test may recognize the drift and react properly, but the test oracle does not take the drift into account as it is based on the incomplete formal model. In summary, some validation steps may depend on information that is only available at test run-time and has to be gathered using sensors. For such situations, the validation mechanism has to keep a decision model in memory in order to adjust the validation process accordingly. In consequence, the test model is executed and adjusted at run-time, which is identical to simulation-based validation. The model is taken as a simulation input and its state is validated against the real system during execution.

Both methods, MBT and simulation, are based on a model that has to reflect the black box's external behavior. Like in SAS design, the test metamodel should be expressive enough for defining concise test models with a minimal effort. In [1], we have already investigated which properties of SAS are relevant for SAS testing. In order to provide model properties that hold for arbitrary SAS, we derived a general notion of such systems based on the concept of feedback loops that are commonly accepted in research as the central concept of all SAS [2]. In our original work, we extracted three different types of artifacts from this concept:

1) Failure scenarios, that can be employed for estimating the effort and progress of system validation.
2) Properties of failures for identifying meaningful verdicts (i.e, semantics of test results).
3) Potential error propagations and their causal chains.

The investigation process was based on *Failure Mode and Effects Analysis* (FMEA, [9]), a safety engineering method. By applying this sophisticated tool set, the analysis was founded on a solid methodology. In this paper, we extend the contributions of our original work as follows:

1) *Example*: In order to improve the understanding of the analysis process, we provide a domestic home robot application and illustrate the single analysis steps based on this example.
2) *Non-functional properties:* In our original work, primarily functional criteria were considered. In this paper, we additionally discuss the challenges originating non-functional criteria.
3) *Abstract reference solution scheme:* Based on the identified requirements, we propose methods and a reference solution scheme of interconnected artifacts that separate the minimal requirements of imaginable solution metamodels.

Besides these new contributions, we enrich our explanations with additional details.

The remainder of this paper is structured as follows: We start with related work in Section II. In Section III, we present the example SAS. In Section IV, we recite and extend our FMEA-based investigation for SAS by non-functional criteria and in Section V, we state the resulting modeling requirements. In Section VI, we propose the solution scheme. Finally, in Section VII, we outline future work.

II. Related Work

In literature, related approaches concerning testing adaptive systems have been discussed in two different research directions. Firstly, in context-aware and context-adaptive system research, model-driven test approaches were found that derive test data from context models. Secondly, several research groups developed methods around SAS engineering.

The most advanced approach concerning context adaptivity is, w.r.t. our knowledge, the work of Wang et al. [10]. The authors propose to construct an abstract control flow graph (CFG) directly from code artifacts by searching for access instructions on data that was delivered from a context middleware. Thus, a grey box perspective is taken, where at least the points of interest that rely on context data are identified throughout the source code. The CFG is an operational test model consisting of transitions and nodes (so-called *context-aware program points*, capps) that point to program locations. From the CFG, sequences of capps are generated. A context manipulator component generates sequences of context manipulations that are applied to the real system. In advance, the system code has been instrumented with feedback instructions that let the context manipulator monitor whether it triggers a certain capp. The sequences of context manipulations are optimized in order to trigger new states in the generated capp sequences. If they do so, a new test case can be assembled from the context manipulating actions. Using Wang et al.'s methodology, relevant operational orders of context manipulations can be automatically derived. Furthermore, their impact on the system can be identified such that a causal link between environment data and adaptation can be defined. However, the rest of the adaptation remains unconsidered. There are no means to validate whether any adaptation outcome is correct.

In SAS research, the earliest statements on the necessity of testing were published by Cheng et al. in [6]. The authors proposed to focus on adaptive requirements engineering and run-time validation to assure SAS's quality. However, they also constructed an abstract model of adaptive software's states consisting of an inner system state plus an adaptation mode or phase. The latter one describes in which variant a system works. Each transition, concerning either mode or state, changes the overall configuration of the system and has to maintain certain local or global properties. It is also discussed that a steady model as behavioral specification is insufficient for a behavior specification of SAS. While the authors' 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 those 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.

A concrete research project on self-adaptivity is DiVA (Dynamic Variability in complex Adaptive systems). Despite comprehensive findings on engineering SAS, it includes a methodology for testing [11][12]. 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, which can be used to find a set of valid configurations. (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 arrays describe multiple context instances that are scheduled as test sequences and provide means for defining coverage criteria on sequences of adaptations. There are also fitness functions that help to minimize the test cases while preserving a good coverage. A drawback of this approach is that the test oracle is manual and does not depend on the previous configuration of the tested SAS. The latter point makes it impossible to examine stateful adaptations where not every system variant can be reached in arbitrary situations.

In context of DiVA, Munoz and Baudry presented in [13] an extended approach that bases on the same workflow. The authors formalize context and variant models and generate sequences of context instances by using Artificial Shaking Table Testing (ASTT). In order to measure the similarity between two context instances, a difference function is defined. Using the means of statistical distribution, context sequences are then generated, which are optimized towards a specific distribution of differences throughout the sequences. The theory behind this work states that sequences with at least one violent peek of difference between three following context instances would be best for testing the SAS. The theory was examined by showing an improved number of found failures in comparison to “non-violent” sequences. The ASTT approach is more powerful than the original DiVA test method as the oracle is defined formally and its predicted system variants also depend on the previous system configuration. However, the drawback of both DiVA test concepts is the lack of a method to validate behavioral adaptation.

Remedy to this lack give Abeywickrama et al. with their *State Of The Affairs (SOTA)* modeling and simulation platform [14]. They propose to model the complete behavior of SAS by specifying feedback loops as first class entities in form of activity diagrams. The feedback loops may communicate by hierarchy, shared components, or events such that different aspects of the system can be separated properly. Despite its modeling capabilities, a simulator, called SimSOTA, was developed that allows the live execution of SOTA models and their inspection during this execution. While the approach allows the definition of parametrical adaptation (using variable assignments) and behavioral adaptation, there are no means to predefine environment change. Thus, no system state can be automatically reproduced and the tester relies on external mechanisms for this purpose. In consequence, SimSOTA is more appropriate to be used in debugging a system in order to observe inconsistent states.

Another promising early-state work has been proposed by Nehring and Liggesmeyer in [15]. The approach enables an SAS engineer to inspect the system’s state space exploratory. The authors assume that the system is component-based and the adaptation is a reconfiguration of this component structure. Along six examination iterations, transactional reconfigurations are investigated with a grey-box knowledge on the components and their interconnections. During the first iteration, a system model is constructed and workloads are defined to stress the system to examine its reaction to different situations. In the second iteration, the structural changes are checked. A third iteration is run in order to analyze whether data integrity is violated when different workloads are applied. During the fourth iteration, correctness of state transfers between exchanged components is examined. In order to evaluate the correctness of transactions that are caused by adaptations, a fifth iteration is run. The sixth and last iteration has the purpose of checking identity relations of components before and after adaptation. As SOTA, this approach can be seen as a special form of debugging without means for environment situation enforcement.

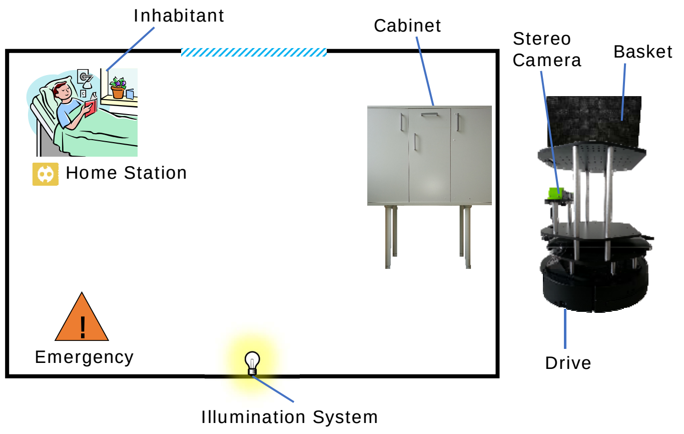

Fig. 1: The HomeTurtle application.

In summary, existing approaches either lack the consideration of behavioral adaptation or they are not capable of enforcing a certain environment or adaptation state, which would be necessary for a systematic validation. Furthermore, non of the proposals has explicit means for tackling non-functional properties. However, any potentially complete test approach for SAS should be able to deal with non-functional requirements as well.

III. Example Application

In order to illustrate the analysis and solution scheme in the remainder of this paper, we sketch an example SAS in the following. We built the domestic home robot system *HomeTurtle*, which supports a disabled person at home. An example scenario is depicted in Figure 1. The service aims to deliver requested items to the inhabitant from a software-controlled storage cabinet. The scenario involves three active elements:

- *Transport Robot:* An extension of the TurtleBot platform (`http://www.turtlebot.com`) operates as transportation system. On top of the mobile robot, a basket is mounted where items can be put in. The system includes an autonomic computing unit and a stereo camera such that it is able to locate itself.
- *Storage Cabinet:* The cabinet is controlled by a WiFi-connected embedded device. This device is capable of triggering magnetically hold flaps, which lock boxes each containing an item. After opening a flap, the item drops out and falls through the cabinet’s base into the robot’s basket. The robot’s battery is charged when it parks on the its home station.
- *Illumination System:* If the natural illumination is insufficient for the robot to locate itself, a lamp can be switched on automatically. For this purpose, the Philips Hue is used (`http://www.meethue.com`). The bulb contains a WiFi-connected embedded control device as well.

In order to start the interaction, the inhabitant requests a specific desired item (by speech input) and the robot starts driving automatically. It plans its way through the room and avoids obstacles that are recognized by using its stereo camera. After finding the cabinet, the robot parks thereunder. By WiFi connection the cabinet's embedded device is signaled to drop the requested item. Then, the robot drives back to the inhabitant for delivery. In a final step, the robot drives on its home station where its battery is charged as long as no request is being operated.

All decisions are made autonomously by an SAS. The computations take place on the computation unit hosted on the robot. By adding several self-adaptive capabilities, this software allows the robot to work in different situations effectively. Firstly, the correct recognition of walls and obstacles relies on appropriate room illumination. When the natural brightness (through the windows) is insufficient, the system automatically switches on the illumination system to improve the quality of obstacle detection. Secondly, the inhabitant can use an emergency switch. If this switch is triggered, the robot is expected to immediately cancel its current action and navigate to an emergency location as labeled in the figure. The purpose of this operation is to avoid the robot being an obstacle while human emergency responders are in the room. Thereby, the moving robot would be a danger itself.

We implemented the HomeTurtle system to experiment with our adaptive software framework *Smart Application Grids* (SMAGs) [16]. Applications that are built using SMAGs are able to adapt their component-based architecture. For example, component implementations are exchanged or components are automatically connected by generated adapters at run-time. Based on these features, we have built the above described home robot system. As the decision logic of the HomeTurtle bases on the reflection of physical objects and actuators, the system can be categorized as CPS. In the remainder of this paper, the capabilities of the HomeTurtle are used to illustrate our analysis process as well as the solution scheme proposal.

IV. Failure Analysis

In this section, we analyze relevant failure characteristics and scenarios of SAS. For this purpose, we apply FMEA [9][17]. 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 [18][19].

According to [7], a *failure* is an event of service deviation from an expectation. The expectation is defined in a specification document, e.g., in form of requirements. An *error* is the inconsistent part of the total system state (internal state plus perceivable external state) which may *propagate* the failure. The cause that *activates* an error is a *fault* that is *active*. There may also be *dormant* faults, which do not cause errors. A failure may trigger a new fault in another component as well. This interaction is called *causation*.

Based on the FMEA process, 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*.

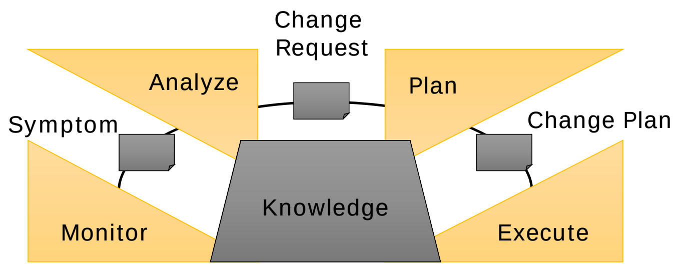

Fig. 2: The MAPE-K feedback loop (cf. [2]).

Step (3) is not an actual part of FMEA. Usually, a *Fault Tree Analysis* (FTA) [20] is performed to visualize the scenarios' dependencies and to enable engineers to trace which faults may have caused a certain failure. The result of FTA is a Fault Tree Set (FTS) comprising multiple trees that represent how a fault may be propagated through the system. Because SAS run a feedback *loop*, error propagation in SAS cannot be described in the form of a tree in general. Hence, we customize the analysis process in this step by constructing a directed graph with logical gates instead.

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

Before starting the analysis, the level of detail has to be specified in order to set up a fixed abstraction perspective on SAS including a well-defined system boundary. FMEA is designed to be ran against an existing technical architecture, which we cannot generally assume to be widely similar in all existing or future developed SAS. Hence, we discard the strict understanding of FMEA by analyzing a general conceptional architecture that comprises minimal necessary components and data flows in between. 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 relies on the feedback loop principle of autonomous systems. The steps that are performed during the execution of this loop are (1) Monitoring of sensor values, (2) Analyzing whether adaptation is required, (3) Planning the adaptation and (4) Executing the plan. During these phases, internal or external Knowledge sources can be used to retrieve or store information relevant to the decision mechanism. This process concept is called the *MAPE-K* feedback loop [2].

As illustrated in Figure 2, the phases of MAPE-K exchange multiple information entities. The system monitors a set of data sources such as *sensors* for external entities or *system interfaces* for internal properties. In our HomeTurtle SAS, the monitored data encompasses a brightness value (detected by the stereo camera) as well as a WiFi-retrieved temperature signal. The set of environment and system states is the relevant computation base for all later decisions–it assembles the *context* of the SAS. The captured information is then inferred to symbolic situation specifications, called *symptoms*. The HomeTurtle software maps the concrete brightness values to symbolic values like `Illuminated`/`Too_Dark` and produces a symptom `Emergency`, if the temperature exceeds a certain level. The symptoms are forwarded to the analysis phase where the system reasons about the necessity of adaptation. Therefore, the conditions of adaptation policies are compared with the symptoms of the current situation. The HomeTurtle's policies are based on Event-Condition-Action (ECA) rules. For instance,

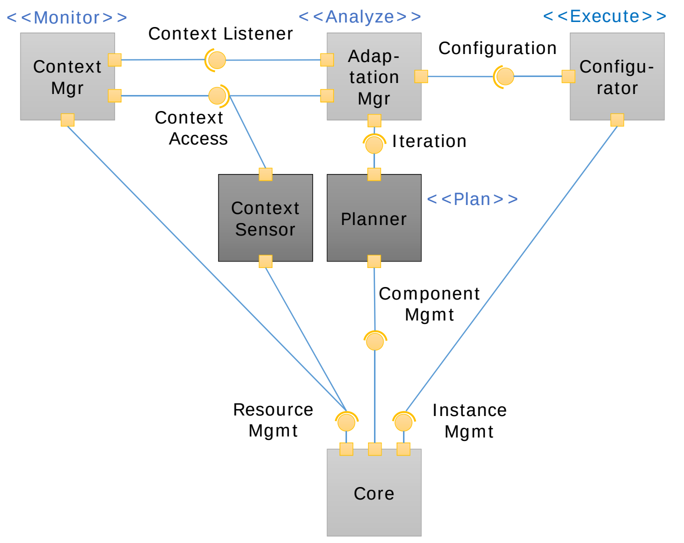

Fig. 3: Example SAS architecture according to [21].

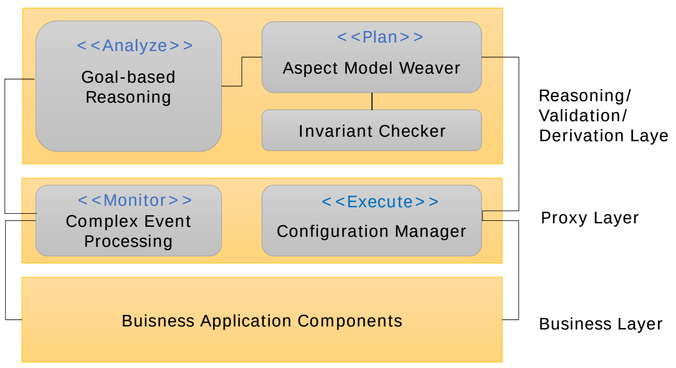

Fig. 4: Example SAS architecture of DiVA according to [22].

there is a rule that is triggered by the emergency signal and commands the robot to perform an emergency adaptation.

Up to this point, the system has determined *if* an adaption should be performed, which is signaled by a *change request*. During the subsequent plan phase, a *change plan* is generated. This plan comprises an operational definition of adaptation actions. An action defines how components of the SAS are changed in detail (e.g., by setting new parameter values or by re-composing the system from modules). In case of the HomeTurtle, the action is to cancel the current process and to redirect the robot to the emergency position using the navigation components. Subsequently, during the execute phase, the plan is applied. This may also involve *effectors* manipulating external entities of the environment. The whole feedback loop is re-ran from this point periodically such that a self-adaptive system always has the intended state (e.g., according to the supposed utility function or goal, and available knowledge) to fulfill its task.

Any SAS architecture adheres to variants of this feedback loop. For instance, Hallsteinsen et al. proposed in [21] a platform based on dynamic product line techniques as depicted in Figure 3. The `Context Manager` component can directly be associated with the monitor phase as it collects and reasons about information that were gathered from resources, the environment (by sensors), or humans. The `Adaptation Manager` then decides whether an adaptation is required based on the context changes and, thus, implements the analyze phase. The `Planner` is responsible of generating a plan for reconfiguring several variation points. Based on this plan, the `Configurator` applies the reconfigurations with the help of the core's instance management interface.

Another example is the DiVA [22] project, whose architecture is depicted in Figure 4. On the lowest `Business Layer` the architectural model of the managed applications is hosted. It contains probes that generate run-time events. These events are consumed by a `Complex Event Processing` component on the `Proxy Layer`. The events are monitored, interpreted, formatted and a context model is updated monitor phase. This context model is input to the `Goal-based Reasoning` component on the upper level which decides whether adaptation is required (analyze phase). As the adaptive capability of the DiVA approach is based on Aspect Weaving, the respective component plans the adaptation. During the plan phase, an `Invariant Checker` determines if the result of weaving fulfills a set of given constraints. Finally, the weaving plan is directed to the `Configuration Manager` that applies the changes to the business application (execute phase).

As the approaches of Hallsteinsen et al. and DiVA, components of arbitrary SAS can be mapped to the feedback loop principle. 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, which are observed or controlled by sensors and effectors respectively. In the following analysis, we use MAPE-K as common viewpoint on SAS architectures. We assume engineers, who are in charge of validating the adaptation capabilities of the system, are equipped with the required tooling to observe the data exchange between the loop's phases.

B. *Step 1) Failure Domain Model (FDM)*

In the following, we provide a set of generic failure properties for SAS. As there are several classes of SAS [23], we cannot assume that each property is reasonable in every concrete system. Instead, the proposed properties are a superset of properties that can occur in MAPE-K-based systems on the discussed abstraction level. Furthermore, we exclude failures that originate in sensors and effectors to create a fixed boundary around the software's scope.

As briefly discussed, each component of an SAS provides services through its interface. In the system's requirement specification the expected behavior of these services is defined. Notably, the specification comprises functional and non-functional requirements. Thus, besides concrete features, which have to be supported by the SAS, the quality of how the features work is constrained. In the HomeTurtle example, a functional requirement is the ability of the robot to navigate through the room without colliding with any obstacles. Associated non-functional requirements are, for example, a lower bound for the precision of the obstacle detection (i.e., percentage of correctly detected obstacles) and upper bounds for the total time required by the robot to navigate from the inhabitant to the cabinet and back.

Basically, the definition of faults, errors and failures [7] holds for arbitrary systems. However, in self-adaptive applications, the boundary between faults and errors can become blurry. Consider, for example, a system that uses models at runtime [24], where the current system state is kept and abstracted in a model to specify the adaptation logic as decision rules against this model. This system is, in principle, able to adapt these rules such that the adaptation logic of the system changes. If this adaptation has not been performed correctly, a new fault is introduced that was not created at design time but at runtime. In other words, due to the ability to dynamically change the system specification, failures can create new faults at runtime. Thus, for adaptive systems which are able to manipulate their own decision logic, faults and errors may not always be distinguishable.

In [7], for each fault, error, and failure a comprehensive list of property dimensions is given. However, as we already defined the level of abstraction and only consider elements that are generic to arbitrary implementations of the feedback loop, this list can be filtered. For instance, it includes properties that specify severity, the cause why, or the life cycle phase when a fault was created. In black box testing against requirements, these information cannot be evaluated as they can neither be observed nor deduced. Concerning failures, for instance, detectability is not relevant in black box testing as non-detectable failures can never be found due to the lacking knowledge on dormant faults. Furthermore, when comparing different notions of FMEA (e.g., [7] and [18]) the FDMs (that are assumed to be generic) differ. In consequence to these issues, we designed our own FDM that only proposes properties that are relevant to SAS in particular.

The resulting FDM for SAS is depicted in Figure 5. Concerning faults, the only property that can be deduced from observed behavior is their *persistence* which may either be *permanent* or *transient*. Detecting permanent faults is usually less challenging than detecting transient faults. For instance, the HomeTurtle aggregates the last hundred temperature values in order to compute their average and to decide whether a fire alert has to be signaled. If the collected values are not deleted after an appropriate time period, the system may run into a memory overflow and stops working permanently. In contrast, when only an outlier value distorts the current value queue, this fault vanishes after a certain time period and is transient to an observer.

Regarding errors, we propose the dimensions *type* and *localization*. In our black box abstraction, the engineer is able to observe whether a false state establishes in the inner knowledge model of the system or in the computational process. For the first type, two deviations are possible: either the model does not correctly reflect its subject (e.g., the stored location of the transport robot is not correct) or its inner constraints are incorrect (e.g., more brightness values are stored than expected). The HomeTurtle may also have process-related errors, for instance, when the emergency symptom is erroneously produced and the system runs an unnecessary adaptation. Furthermore, errors can localize either locally or globally in the potentially distributed SAS. In our example, the set-up of items in the cabinet could not reflect its real contents. This error would be shared between all technical elements over the WiFI network.

Concerning failures, the authors of [7] distinguish between content and timing (early/late) correctness. In contrast, an SAS's goal definition may aim on other non-functional properties like energy-usage. Thus, we alter the original distinction to *non-functional* (i.e., the service performs not with the expected quality) or *functional* (i.e., incorrect service behavior). In our example, we can distinguish between failures where the robot does not deliver an item within a required time or completely fails in delivering it. Furthermore, failures in SAS can either manifest *internally* (e.g., in an inconsistent model) or *externally* (e.g., when the robot heavily collides with an obstacle). Concerning, the *Appearance* dimension, sensed and analyzed information may lead to un-intentional (*false-positive*), missed (*false-negative*), or *semantically* wrong sensor events, change request, or adaptation actions. The HomeTurtle, for instance, may start driving to the emergency position without cause, it may miss the emergency signal or mis-interpret it.

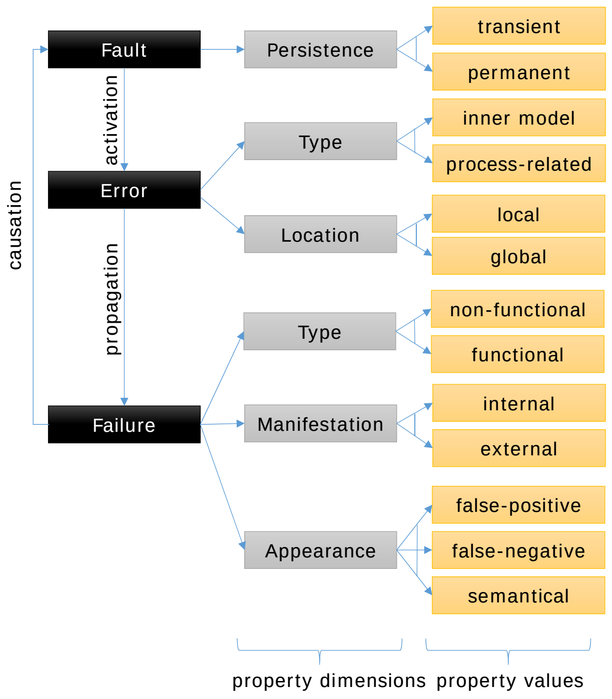

Fig. 5: The SAS Failure Domain Model.

This FDM is a valuable source for classifying faults, errors, and failures in concrete SAS. It describes their abstract properties that can be instantiated for real world systems. Verdicts (i.e., the classification of test results, for instance `Pass`, `Fail` or `Inconclusive`), can be parametrized by these information.

C. Step 2) Failure Scenarios

In this section, we identify scenarios of failure occurrence in SAS. In contrast to other FMEA applications, we cannot give a general method of prioritizing these scenarios as this strongly depends on domain-specific conditions. The only applicable, general evaluation standard is *criticality*. In this case, according to [23], each adaptation operation can be either harmless, mission-critical, or safety-critical. When the HomeTurtle robot drives along an non-optimal path, this failure is harmless. In contrast, not delivering an item would be mission-critical. If the robot collides heavily with a human being, the failure can even be safety-critical. However, other SAS may have completely different requirements such that statements about the severity of failure scenarios cannot be generalized.

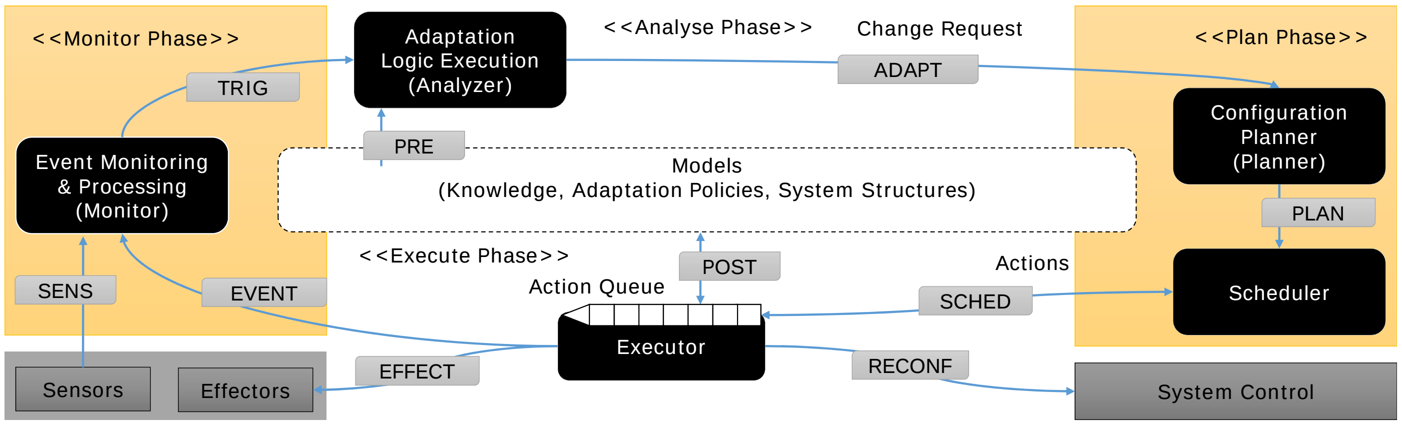

Fig. 6: Conceptional architecture of SAS.

TABLE I: SAS failure scenarios.

FID	**CID**	**Fault** in...	**Error** in...	**Failure** in...	**Propagation**
SENS	Monitor	sensor interpretation ► transient or permanent	environment reflection ► process-related ► local or global	produced symptoms ► functional or non-functional ► internal ► false-positive, false-negative or semantical	TRIG
TRIG	Analyzer	symptom interpretation ► transient or permanent	adaptation decision ► process-related ► local or global	change request ► functional or non-functional ► internal ► false-positive, false-negative or semantical	PRE/ADAPT
PRE	Analyzer	model interpretation ► transient or permanent	adaptation decision ► process-related ► local or global	change request ► functional or non-functional ► internal ► false-positive, false-negative or semantical	TRIG/ADAPT
ADAPT	Analyzer	reasoning algorithm ► transient or permanent	adaptation decision ► process-related ► local or global	change request ► functional or non-functional ► internal ► false-positive, false-negative or semantical	PLAN
PLAN	Planer	planning algorithm ► transient or permanent	planning decisions ► process-related ► local or global	change plan ► functional or non-functional ► internal ► semantical	SCHED
SCHED	Scheduler	scheduling algorithm ► transient or permanent	scheduling decisions ► process-related ► local or global	wrong order of actions ► functional or non-functional ► internal ► semantical	POST/EVENT/ EFFECT/RECONF
POST	Executor	model manipulation ► transient or permanent	model ► inner model ► local or global	model inconsistent ► functional or non-functional ► internal ► semantical	PRE/PLAN
RECONF	Executor	reconfiguration ► transient or permanent	system configuration ► inner model ► local or global	system reflection ► functional or non-functional ► internal ► semantical	–
EVENT	Monitor	system event monitoring ► transient or permanent	system reflection ► process-related ► local or global	system events ► functional or non-functional ► internal ► false-positive, false-negative or semantical	TRIG
EFFECT	Executor	actual control ► transient or permanent	erroneous actuator commands ► process-related ► global	environment state ► functional or non-functional ► external ► false-positive, false-negative or semantical	(SENS)

The scenario identification relies on the feedback loop's conceptional structure, which is depicted in Figure 6. Hence, we decompose the MAPE-K loop in five components: Monitor, Analyzer, Planner, Scheduler, and Executor. The latter two are separated, to enable the consideration of *interaction* between adaptation and running processes. After the Analyzer detected *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 current system actions (retrieved from the Executor) and composes them with adaptation intents. In the HomeTurtle system, this feature plays an important role as well. For instance, when an emergency is signaled, the delivery process is expected to cancel immediately. In contrast, an adaptation concerning illumination conditions would make no sense while the robot is parked under the cabinet. In this case, the scheduling implementation is expected to first atomically execute the waiting operation and run the adaptation afterwards.

All components are considered as black boxes and are connected by data flow edges (blue arrows). Additionally, the process contains Sensors, Effectors, System Control, and the central knowledge Models. The latter one contain information about the system structure, adaptation logic, and further knowledge relevant to adaptation decisions. Sensors and Effectors communicate with the external world (e.g., other systems or the physical reality). System Control provides an interface for system reconfiguration actions that are controlled by the Executor.

Based on this structure, we derive failure scenarios by investigating potential wrong processing of input data by a certain component. As there may be multiple outputs of a component, each component can produce multiple failures. We list our found failure scenarios in a worksheet as presented in Table I. A scenario represents the possible occurrence of a fault and its related causality chain. The description of each scenario 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 6. In the following, each scenario is described in detail. As an extension to our original work, we add examples and discuss non-functional aspects as well.

SENS: The first scenario comprises test input received from the Sensors and misinterpreted by the Monitor component. During the interpretation, values are mapped, aggregated and inferred. It might also be the case, that a history of values is maintained in order to infer over them. Such a fault can activate an error that comprises an incorrect reflection of the environment such that the produced symptoms are incorrect.

Example: The HomeTurtle's monitoring component collects brightness values over the last ten minutes but fails to discretize them correctly. The symptom that indicates a low illumination is not being produced as expected.

Non-functional aspects: This scenario comprises test input received *too late* from the Sensors. Faults of this kind activate errors comprised of the reflection about outdated observations of the environment such that the produced symptoms are incorrect.

Example: The HomeTurtle's monitoring component collects brightness values only every minute. Thus, in the worst case, the HomeTurtle will not adjust itself to a changed illumination for almost a minute, which is not the expected behavior. A higher sampling rate has to be used.

TRIG: A symptom produced by the Monitor does not or unintentionally trigger a corresponding change request or a wrong one.

Example: The HomeTurtle's analysis components fails matching the symptom description with an adaptation policy's condition. Thus, the expected adaptation is not being performed as expected.

Non-functional aspects: A symptom produced by Monitor triggers a corresponding adaptation *too late*.

Example: The HomeTurtle observes its own remaining battery capacity. If a user requests the HomeTurtle to deliver an item from the cabinet, the HomeTurtle checks whether its remaining energy suffices to fulfill the request and will incorporate a stop at the charging station if the remaining capacity is too low. If the analysis' result reflecting whether the capacity suffices or not is send too late to the Planner, an erroneous plan is generated, which does not include the stop at the charging station.

PRE: The SAS knowledge resources contain information that may constitute pre-conditions to an adaptation decision. If these information are misinterpreted, the adaptation decision can differ from the designer's expectation.

Example: According to the HomeTurtle adaptation policies, the illumination system has to be switched on as soon as the obstacle recognition precision deceeds a certain level. Thus, the current precision is inferred using the empirical metrics on the stereo camera's precision that is parameterized with the currently detected brightness value. If the empirical model is corrupt, the adaptation of the illumination system is not performed in the relevant situations.

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: Depending on the deduced adaptation decision, the system is in charge to produce and correctly interpret the change request. In cases where this output is corrupt, adaptations are not performed as expected.

Example: After the brightness value was found too low, the respective change request was derived but is not forwarded to the planner due to an exception. Thus, the adaptation initiation is lost.

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 has 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.

Example: In the emergency case, in the constructed adaptation plan the canceling of the current delivery task and is queued after the actions necessary for driving to the emergency position. Thus, the robot first delivers the item and approaches the emergency position subsequently, which was not intended.

Non-functional aspects: Often, the Planner reasons over non-functional properties like response time or energy consumption. But, the planning task itself effects these properties by utilizing the same resources. In consequence, the resulting decision of the Planner is infringed, because the assumptions taken by the Planner are violated.

Example: The HomeTurtle shall drive to its charging station if the battery capacity falls below 10%. To reach the charging station, the HomeTurtle consumes energy. But, executing the planner, to decide whether to drive to the charging station or not, consumes energy, too. Thus, by executing the planner, the maximum distance of the HomeTurtle to its charging station is decreased. If the planner does not consider this decreased distance, the decision to go for charging, will be made too late.

SCHED: Reconfiguration actions potentially interact with the system's control flow. Such problems arise because variability cannot be completely orthogonal to the system's task execution. The expectations of the designer may even encompass that certain actions may be transactional. Differing from these expectations activate errors in the scheduling process and are observable as wrongly ordered actions.

Example: Due to a wrongly designed scheduling component, new adaptations actions are enqueued behind all previously initiated system actions. Thus, for instance, the illumination adaptation is being deduced while driving to a certain position but performed after the driving process. Then it may be too late to avoid a collision.

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.

Example: The driver for the control of the robot's wheels is implemented as a blocking service such that the emergency adaptation cannot be performed immediately. Instead, the designer expects the system to cancel the operation.

POST: The Model's part that represents the reconfigured systems may be inconsistent after the execution because a model manipulation was performed erroneously by the Executor. Thus, the reflected system state may differ from its real configuration. This deviation may harm future adaptation decisions in form of wrong preconditions (PRE). The failure can be observed in the model's state.

Example: The illumination adaptation is correctly performed but due to an exception this change is not reflected in the model. During the next adaptation loop, the decisions that depend on this information will be erroneous.

EFFECT: Another output of the Executor can be actions that have to be performed by external systems using the Effectors. If actions are not generated correctly and forwarded to the effectors (e.g., due to corrupt drivers), the representation of these externals loose synchronization with potential internal model representations. As the Sensors may perceive data from the manipulated system, a SENS scenario may be caused indirectly.

Example: The signal to the illumination system may be misinterpreted and the Hue bulb is not switched. While the physical condition remains unchanged, the SAS now assumes the environment to be altered as it reflects its own actions in the knowledge model.

EVENT: The last failure scenario is related to events that are produced in the software system and are propagated to the Monitor component which makes them part of the context representation. In the related failure scenario, the generated events are erroneous.

Example: In order to avoid concurrent adaptations, the HomeTurtle is expected only to obtain new sensor data when all adaptation actions are finished. Afterwards, a respective event is produced by software that causes a re-start of the feedback loop. Loosing this trigger event constitutes a failure.

Properties of faults, errors, and failures as proposed in our FDM can depend on the concrete architecture of the system. Based on the assumption that the system was built as our abstract feedback loop suggests, some property values can be neglected. In our scenarios worksheet, the remaining ranges are denoted. Regarding faults such a restriction cannot be deduced from the general feedback loop. Thus, arbitrary SAS can include transient or permanent faults in each of the proposed components.

Concerning errors, only the Executor is expected to directly manipulate the model. Thus, the inner-model error type only occurs in this component when reality is no longer correctly reflected in the system's knowledge base. All other faults impact to process-related state of the system. Depending on whether the SAS is distributed or not, each component may propagate its potentially erroneous outputs through the complete infrastructure. However, effects that propagate out of the system always have to be considered global, as all monitoring components may detect external changes.

The appearance of a failure depends on the content of a component's output. In cases of decisions (symptoms, change requests, system events, effector actions), they may be missed (false-negative), unintentionally performed (false-positive) or semantical wrong. In cases of adaptation plans and model manipulations, which are always expected to be produced, the potential errors impact their contents only. All components may produce such functional failures. Despite the non-functional aspects which were discussed in context of the SENS, TRIG, and PLAN scenario, each computation can be limited in its budget usage in general. Thus, we propose to consider non-functional failures in all scenarios.

D. Step 3) Fault Dependency Graph

As final artifact, we construct a faullt dependency graph as depicted in Figure 7. The nodes of the graph identify each a certain failure scenario or a logical or gate. Connections reflect the causations as listed in respective column of Table I. The visualization illustrates the potential cyclic failure propagation through inner system events, model manipulation, or physical sensors or effectors 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.

The graph can be used by engineers to identify potential sources of observed failures. Furthermore, quality assurance can be steered using the graph by measuring or estimating each scenario's probability of occurrence. Thus, it can even be deduced how probable a certain causal chain or how severe its a fault's impact is.

V. Requirements to Models for SAS Testing

All following requirements for self-adaptation test methods are based on a selection of the presented failure scenarios. In the

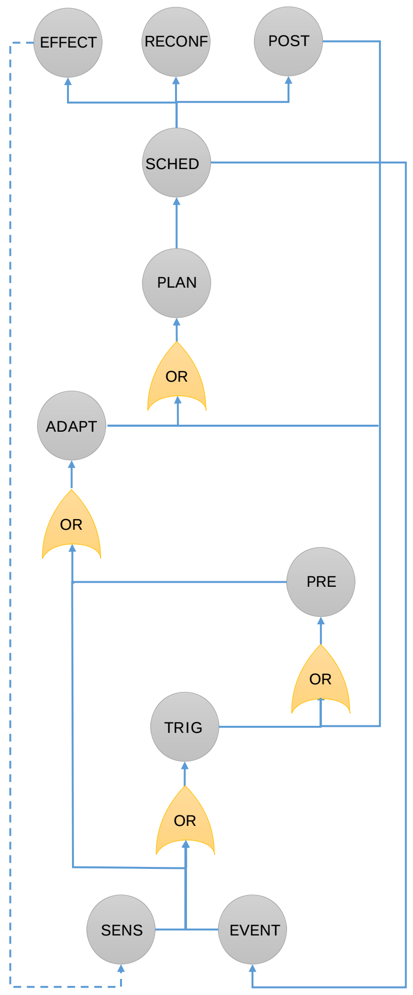

Fig. 7: Fault Dependency Graph.

following, we list these mapped requirements and name each of them for later reference. The requirements are formulated as *assurance tasks* that have to be fulfilled by employing validation methods.

A. Functional Requirements in SAS Testing

F1) **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)

F2) **Correct adaptation initiation:** Assure that events initiate the correct adaptation if all preconditions in the model hold. Events, conditions, and adaptation decisions have to be associated in the models. (↦TRIG/PRE/ADAPT)

F3) **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)

F4) **Consistent interaction between adaptation and system behavior:** Assure that the generated adaptation plan is correctly scheduled with the system's control flow. A model is required to define which adaptation is allowed in which state of application control. (↦SCHED)

F5) **Consistent adaptation execution:** Assure that (1) the generated adaptation schedule is applied to system structure and (2) the synchronization between the running system and the models is consistent after adaptation. (↦POST/RECONF)

F6) **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)

B. Non-functional Requirements in SAS Testing

Whereas the requirements to SAS testing in the last subsection focused on assurance of the SAS's functionality, a second type of requirements to SAS testing exists: non-functional requirements. The central concept for non-functional testing of SAS is the budget, which covers a boundary for a selected non-functional property. For example, the limited capacity of a battery sets a hard budget in terms of energy, which must not be exceeded. The following aspects of SAS require the consideration of non-functional requirements:

NF1) **In-budget sensor interpretation**: A failure due to incorrect sensor interpretation has a non-functional dimension in that, e.g., the timing behavior of the sensor interpretation is faulty. In other words, sensor interpretation has non-functional requirements, which must not be violated. These requirements include the ability to handle imprecise sensor data (precision), timing constraints on when and how long sensor data is valid and resource budget constraints, if the sensors are power by a battery, which is characterized by a limited energy capacity. (↦SENS)

NF2) **In-budget adaptation initiation**: In addition to the assurance of the correct adaptation to be initiated, it is important to ensure that this adaptation is initiated in time. If an adaptation is initiated too late, multiple qualities are potentially infringed. Imagine, for example, a self-adaptive system optimizing for energy efficiency. The later an adaptation to save energy is initiated, the more energy is consumed in the meantime, which infringes the goal of the self-adaptive system. (↦TRIG)

NF3) **In-budget adaptation planning and execution**: Also for planning, non-functional requirements have to be fulfilled besides correctness. Again, budgets of non-functional properties must not be exceeded. A particular problem in this regard is the assessment of the planning step itself in terms of various non-functional properties. For example, the runtime of an optimizer using integer linear programming is double exponential in the worst case, but almost linear in the average case [25]. In addition, the determination of the size of the respective budgets is a highly complex task. This is because often the goal of the self-adaptive systems is to optimize for selected non-functional properties, but performing optimization (and adaptation) effects these non-functional properties. (↦PLAN)

C. Required Test Adequacy Criteria and Coverage Metrics

Additionally, in testing, adequacy criteria are required to restrict the tested behavioral space and, nevertheless, have a reasonable and meaningful test result. Furthermore, coverage metrics are used to compute in which degree the state space is covered by the generated or performed test cases. For less complex systems, many criteria for test adequacy and coverage metrics 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 adequacy criteria and coverage metrics for SAS which can be composed:

C) **Adequacy criteria and coverage metrics for SAS:** Find constructive adequacy 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 metrics for measuring a test suite or its execution.

VI. Reference Solution Scheme

In order to help testers facing the challenges stated above, we propose a scheme consisting of methods of fault detection and representation artifacts.

A. Considerable Methods

In black box testing, the SUT is represented by well-defined interfaces without defining its internal behavior. In our analysis, we based on the most abstract definition of an SAS, namely the MAPE-K feedback loop. We have derived five crucial components that are black boxes and provide interfaces where data can be sent to or received from. Despite the assumed informational entities defined by the knowledge element of the feedback loop, the components' internal state may effect the outcome of their computations. Thus, the black box interface can have a stateful protocol as well.

In order to validate the correct outputs of each component, several input data scenarios have to be specified and the output has to be predicted. Such test cases *enforce* a certain state of the interface protocol that is relevant during the prediction. In testing, the prediction task is solved by oracles. An oracle has to be specified according to the requirements of the SUT. In order to automate the validation process as far as possible, a specification is most useful in a formal representation such as a model. Models can be employed either for generating test cases or by executing them directly. The latter method is identical to simulation-based validation. Hence, the simulation model is executed in parallel to the SUT and the simulation's propositions are frequently validated against the SUT.

Both generation and simulation have several pros and cons. In Figure 8, the relation between the two methods is depicted. The first artifact, which is assembled during the system's development process, is the requirement specification. Based on this specification, the design is derived during the system's development process. The design models are refined in incremental steps until the code level is reached, which is the final representation. The refinement process may be be manual

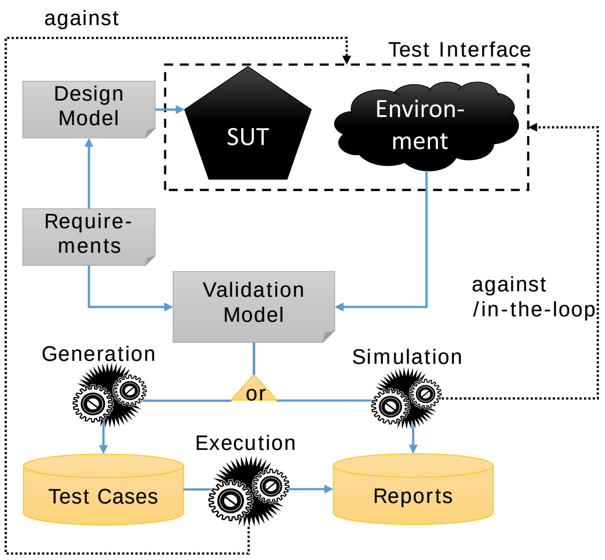

Fig. 8: Validation Methods.

or partially model-driven (i.e., Model-Driven Architecture). When the design model is specified, misinterpretations and mistakes in the design itself will create faults that later have to be uncovered by validation. Despite these fault sources, during the refinement the loss of information between two levels of abstraction may have effect on the system's correctness. Furthermore, due to the change of requirements during the whole process, inconsistencies of already implemented artifacts may arise. In the end, validation is responsible to examine how far the implementation still matches the requirements. Therefore, an engineer has to define a validation model according the initial requirements. The model should be independent of the system's design in order to avoid the re-implementation of erroneous interpretations. The validation model specifies inputs and assertion actions against the interface of the SUT. Additionally, it has to specify environment changes in order to trigger and validate the self-adaptation. Therefore, the environment's properties and the change of their values over time has to included by appropriate model representations.

The automatic generation of test cases from this model is called Model-based Testing (MBT) [8]. The generation process is controlled by an adequacy criterion, that specifies which entities of the model (i.e., states, transitions, data elements) have to be covered before the production of test cases terminates. A generated test case specifies a strict sequence of test actions that are applied against the SUT. All test cases are generated only a single time, as long the requirements stay fix as well. They are saved in a test suite and can be replayed over multiple regressions when the implementation has changed. During the test case execution, inputs are sent to a test interface and the resulting outputs are checked against the predicted data using *assertions*. The test interface provides appropriate access to system functions and environment control as well monitoring capabilities. For each run, it can be reported whether it completely succeeded or in which execution step a failure occurred. The MBT approach has two advantages. Firstly, all test cases are assumed to be strictly reproducible as their execution solely depends on the action that are sent to the SUT. Secondly, a coverage can be measured, e.g., by counting the number of executed test cases in the relation to the complete test suite.

In contrast, simulation directly executes the model. Decision points during the model interpretation have to be made randomly or they have to be controlled by a user or a heuristic (similar to an adequacy criterion). During the simulation, failures can be observed in form of deviations from the real system's behavior. Each derivation can be stored in a report. During the simulation, data from SUT and environment can be queried and used to determine decisions. This principle is called "in-the-loop" and states the main advantage of simulation in comparison to MBT.

The decision between MBT and simulation depends on the existence of uncertainty-establishing artifacts that have to be involved in the loop. Both methods rely on a validation model, which has to implement the requirements stated in our previous analysis. Each of the defined requirements is a concern to this model. In the following section, we propose a conceptional concern-separated structure for such a validation model.

B. Counter Feedback Loop

The SAS deduce adaptations from monitored context changes. In contrast, a validation mechanism has to work exactly vice versa. In our work, we call this principle *Counter Feedback Loop* as depicted in Figure 9. The context has to be actively changed in order to trigger the SAS to adapt and enforce a certain adaptation state, whose effects can be examined. Hence, the test model has to include a `Change` specification containing a set of scenarios. Executing such scenarios allows to check whether sensor data is correctly obtained and inferred. (requirement (F1)). In case the monitoring or processing of this data involves performance requirements (requirement (NF1)), budgets have to be defined in this artifact.

Afterwards, the reaction of the system has to be observed. Hence, a `Causal Connection` between sequences of change and adaptation initiation have to be specified (requirement (F2)). Sensor data on objects that are observed using an in-the-loop mechanisms cannot be predicted but queried whether a certain value is matched. From this, an adaptation decision can be predicted and examined as well. The specification of causal connections can be obtained by defining which symptoms and in which change requests are expected to be produced in a certain context state. To support this inference it can be beneficial to capture the `Environment Structure`. Thus, the effects of past changes can be stored as configuration states during the simulation or generation state an can be taken into account when symptoms are derived. If their are time restrictions on adaptation initiation, budgets have to be specified again (requirement (NF2)).

In the next step, it has to be specified which symptoms end up in which `Adaptation Plans` (requirement (F3)). The respective model maps symptoms to a certain operational set of changes, that are going to be applied against the SAS. In an additional artifact, the system's externally observable behavior has to be modeled in form of a `Service Specification`. This informational artifact captures how the observable interface protocol of the SAS changes in a certain adaptation mode (requirement (F5)). Thus, it involves also propositions on the interaction between the produced adaptation plans and the running processes (requirement (F4)). Finally, the system action's impact on the environment can be examined by predicting these actions in the behavioral system model (cf. requirement (F6)). All qualitative expectations concerning the specified service

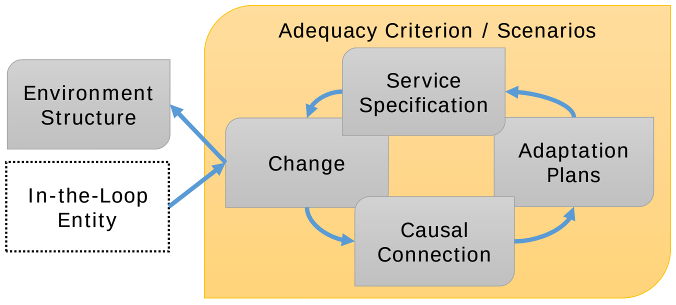

Fig. 9: The Counter Feedback Loop principle.

and its behavior after the adaptation's execution can be taken into account by budget values (requirement (NF3)). As the selection of test sequences through the modeled scenarios is indeterminstic, an adequacy criterion or simulation heuristic has to be specified (requirement (C)).

In summary, the validation model consists of relevant scenarios that stress the system, and oracles that map the scenarios' inputs to the intermediate or final outcomes of each feedback loop component. The different requirements establish a set of concerns that have to be contained in a model and specifiable by its metamodel. By separating these concerns, as proposed by our Counter Feedback Loop principle, different aspects of validation can be decoupled.

VII. Conclusion and Future Work

In this paper, we extended our original work [1] where we applied a customized Failure Mode and Effects Analysis (FMEA) to a conceptional SAS based on the minimal structural assumptions of MAPE-K. We derived a Failure Domain Model in order to provide a system in which faults, errors and failures can be classified. Subsequently, we derived ten distinct failure scenarios that may occur in the process of adaptation. By building a fault dependency graph, we visualized potential cyclic propagation of failures in such systems. In consequence, a set of *founded* modeling requirements were stated that all can be mapped to one or more of the described failure scenarios. 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 respective assurances are considered.

The extensions of this paper are threefold. Firstly, we exemplified the analysis with our HomeTurtle system in order to clarify the complete process. Secondly, we improved the consideration of non-functional properties that have to be dealt with in validation. Thirdly, we propose to use either Model-based Testing or Simulation for examining SAS against their requirements. Both methods are based on models, whose necessary information have proposed in this paper as well. Based on this premises, test engineers are equipped with indicators when building appropriate generation or simulation frameworks.

For further investigation, it is necessary to instantiate the identified requirements for real-world SAS systems. If implementations can be mapped to several adaptivity frameworks and express the majority of necessary test cases, our approach can be attested substantial and generic.

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 Collaborative Research Center 912 (HAEC) and the "Center for Advancing Electronics Dresden" (cfaed) as well as T-Systems Multimedia Solutions.

References

[1] G. Püschel, S. Götz, C. Wilke, and U. Aßmann, "Towards systematic model-based testing of self-adaptive software," in ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications, 2013, pp. 65–70.

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

[3] M. Broy, M. V. Cengarle, and E. Geisberger, "Cyber-physical systems: Immanent challenges," in Large-Scale Complex IT Systems. Development, Operation and Management. Springer, 2012, pp. 1–28.

[4] 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.

[5] S. S. Kulkarni and K. N. Biyani, "Component-based software engineering." Springer, 2004, ch. Correctness of Component-based Adaptation, pp. 48–58.

[6] 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.

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

[8] M. Utting, Practical Model-based Testing: A Tools Approach. Morgan Kaufmann, 2007.

[9] H. E. Roland and B. Moriarty, System safety engineering and management 2nd edn. John Wiley & Sons, Chichester, 1990, ch. Failure mode and effect analysis.

[10] 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.

[11] V. Dehlen and A. Solberg, "DiVA Methodology (DiVA Deliverable D2.3)," https://sites.google.com/site/divawebsite, visited 02/01/2014, 2010.

[12] A. Maaß, D. Beucho, and A. Solberg, "Adaptation Model and Validation Framework – Final Version (DiVA Deliverable D4.3)," https://sites.google.com/site/divawebsite, visited 02/01/2014, 2010.

[13] F. Munoz and B. Baudry, "Artificial table testing dynamically adaptive systems," arXiv preprint arXiv:0903.0914, 2009.

[14] D. B. Abeywickrama, N. Hoch, and F. Zambonelli, "Simsota: Engineering and simulating feedback loops for self-adaptive systems," in Proceedings of the International C* Conference on Computer Science and Software Engineering, ser. C3S2E '13. ACM, 2013, pp. 67–76.

[15] K. Nehring and P. Niggesmeyer, "Testing the rconfiguration of adaptive systems," in ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications, 2013, pp. 14–19.

[16] 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.

[17] "MIL-STD-1629A (1980)," Procedures for performing a failure mode, effect and criticality analysis. Department of Defense, USA.

[18] H. Sozer, B. Tekinerdogan, and M. Aksit, Archtitecting Dependable Systems IV. Springer, 2007, ch. Extending Failure Model and Effects Analysis Approach for Reliability Analysis at the Software Architecture Design Devel.

[19] 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.

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

[21] S. Hallsteinsen, E. Stav, A. Solberg, and F. J., "Using product line techniques to build adaptive systems," in 10th International Software Product Line Conference, 2006.

[22] B. Morin and A. Solberg, "Reference architecture (DiVA – deliverable D3.3)," https://sites.google.com/site/divawebsite, visited 02/01/2014, 2010.

[23] J. Anderson, 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.

[24] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," Computer, vol. 42, no. 10, 2009, pp. 22–27.

[25] G. L. Nemhauser and L. A. Wolsey, Integer and combinatorial optimization. Wiley Interscience, 1999.