A Combined Simulation and Test Case Generation Strategy for Self-Adaptive Systems

Georg Püschel, Christian Piechnick, Sebastian Götz,
Christoph Seidl, Sebastian Richly, Thomas Schlegel, and Uwe Aßmann
Software Technology Group, Technische Universität Dresden
Email: {georg.pueschel, christian.piechnick, sebastian.goetz1,
christoph.seidl, sebastian.richly, thomas.schlegel, uwe.assmann}@tu-dresden.de

***Abstract*—With the introduction of self-adaptivity in software architecture, it becomes feasible to automate tasks that are performed under changing conditions. In order to validate systems with such capabilities, the conditions have to be enforced and reactions verified. An adequate set of scenarios must be performed to assure the required quality level. In our previous work, we investigated a set of requirements for a self-adaptive system validation strategy as well as a high-level solution scheme. In this paper, we instantiate this scheme and propose a set of timed models that work together as black box test model for our example SAS HomeTurtle. The model can be either used for simulation or test case generation; for both approaches, a unifying infrastructure is described. We further show an example simulation run and present our implementation—the Model-driven Adaptivity Test Environment. The proposed methodology enables test experts to maintain the complex behavior of SAS and cover an adequate part of it in testing.**

Keywords*—*Self-Adaptive Systems; Service Robots; Model-Based Testing; Simulation; Feedback Loops

I. INTRODUCTION

In our original work [1], we introduced a strategy for creation and execution of timed simulation models for Self-adaptive Systems (SAS). This kind of system adapts itself according to changes in its environment [2][3]. The continuous execution of sensor monitoring, analysis, planning, and adaptation execution is organized in feedback loops [4]. Due to the use of intelligent reasoning strategies, an SAS is capable of fulfilling its tasks more efficiently or it even may find solutions to tasks that were not explicitly defined at design time. Potential adaptations encompass simple changes of certain control variables, structural re-organization of components and the exchange of behavioral strategies that might better fit for the found environment situations.

In our work, we aim to provide solid SAS development methods and, thus, we also require a validation approach that is able to deal with the complexity of such self-adaptive behavior. The mechanisms that decide autonomously have to be validated extensively before deploying the system in a productive environment. A limitation constitutes from the fact that an SAS can be adapted externally or reason about unanticipated events. These aspects can never be tested comprehensively before delivery and, thus, are excluded from the scope of our proposed solution.

However, even for these systems, the user's trust has to be gained by examining the system in an appropriate variety of scenarios. Hence, validation methods can be performed on different abstraction layers as, for instance, the German V-Modell [5] proposes. On the lowest abstraction layer of modules, knowledge of code and design models can be utilized. However, due to the complexity and large variety of possible situations, performing a comprehensive validation (e.g., by deriving and executing test cases) on these levels is expensive.

In contrast, validating SAS applications on acceptance level, based on requirements of a more abstract specification, is more promising. For this purpose, the engineer no longer relies on detailed knowledge of the system interior but on a black box interface that is used to enforce situations and verify the outcome. In contrast to white box testing, black box approaches cannot locate faults. Thus, each found failure has to be analyzed by additional means in a subsequent step.

Setting up a black box interface that provides all necessary operations to interact with the system and to query information that has to be examined is the first crucial task during the validation phase. The expected behavior of the SUT can be specified based on this interface. An appropriate method for such specification is model-based testing (MBT, [6]). In this approach, a test model is specified and test cases are generated from it. In the most comprehensive variant of MBT, the model captures all information about which test data is sent to the tested system and which reactions are expected. In this way, the test model serves as a *test oracle*, which determines the correctness of observed reactions or predicts these reactions.

A further problem can arise when the SAS is deployed in complex environments where not every property of a situation can be enforced. For instance, the interaction with certain entities (e.g., hardware controllers or physical objects) is difficult to formalize. Instead, the test model designer may specify some future decisions depending on the state that is observed from these entities at test execution time (i.e., a run-time-dynamic oracle). Test cases do not support decisions on run-time information, as the generated actions cannot be exchanged or reordered in case of adaptations depending on such properties. Instead, the test model has to be executed directly (i.e., without test case generation). Therefore, we propose to perform simulation and capture the discussed non-specifiable parts of the system or test environment *"in-the-loop"*. In our concept simulation means to produce inputs that are given to the real SUT and the test model and compare the results of both. A drawback of simulation in comparison to the test case generation is that there is no fixed set of test cases to be replayed for regression. In consequence, both test case generation or simulation may be employed depending on the quality requirements and relevant context of a set of system parts under test.

However, both methods rely on a common artifact—a test or simulation model. A generic SAS testing framework has to provide a respective metamodel that is expressive enough for compact specification of all behavioral and adaptation-related

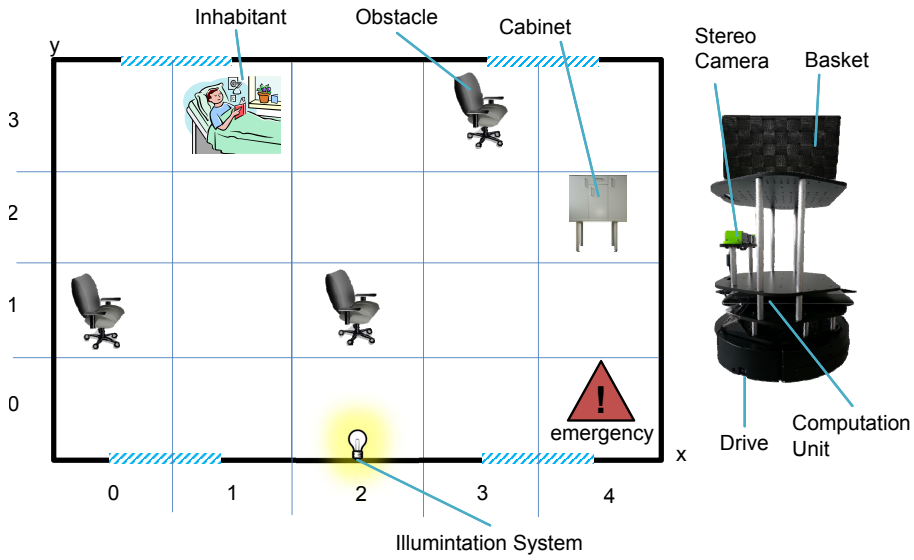

Fig. 1. Scenario: HomeTurtle operating in a flat.

aspects. These aspects are given by several requirements that we derived in our previous work [7]. The following requirements were formulated as goals:

(1) Correct sensor interpretation
(2) Correct adaptation initiation
(3) Correct adaptation planning
(4) Consistent adaptation/system interaction
(5) Consistent adaptation execution
(6) Correct system behavior (especially actuator actions)

Goals (1) and (6) include the validation of the correctness in sensor perception and actuator control. Both properties can be checked in isolation by instrumenting the respective drivers. In this paper, we focus on the goals (2)-(5), which directly deal with the SAS feedback loop (sometimes referred to as MAPE loop: monitor, analyze, plan, execute [4]). In order to match the requirements, the test metamodel has to provide means for defining in which situations an adaptation has to be initiated (goal 2), how the system is expected to adapt (goal 3), how the adaptation is expected to be scheduled with non-adaptation-related behavior (goal 4), and how the result of the adaptation must look like (goal 5).

In [1], we proposed a methodology to separate all these aspects in components of a composite simulation model. Parts of our model are enriched with *assertions* on the System Under Test's (SUT) interface in order to define how a simulation state has to be verified. The complete modeling methodology is illustrated using our *HomeTurtle* domestic robot application. Throughout the paper, the Unified Modeling Language (UML) and Object Constraint Language (OCL) are used for representing almost all details of the model by a widely-understood standard syntax and semantics.

In the HomeTurtle scenario, a robot is deployed in a flat of a handicapped person and is capable of delivering various items, which are stored in a software-controlled cabinet. Besides reciting this illustrative example as well as the introduced test methodology, we contribute the following aspects in this paper:

1) **Simulation- vs. generation-based validation:** We describe how the proposed modeling concepts are used for simulation alternatively to test case generation. For this purpose, an infrastructure is proposed that unifies both approaches.

2) **Details on implementation:** All concepts have been implemented in our integrated test environment MATE. The components of this tool are presented.

3) **Extended related work:** We extend the discussion on the body of knowledge in SAS testing.

The remainder of this paper is structured as follows: In Section II, we introduce our example adaptive system. In Section III, we present our approach based on this example. In Section V, we illustrate an example simulation run. In Section IV, we describe how the necessary infrastructure for simulation and generation can be unified. In Section VI, we present our implementation and experimental environment. In Section VII, we discuss related work. In Section VIII, we discuss conclusion and future work.

II. Example Application: Home Turtle

In this section, we present an illustrative example of an SAS controlling a robot that is instructed to support a handicapped person at home. The scenario is depicted in Figure 1. A service robot "HomeTurtle" (an extended version of the TurtleBot platform [8]) is initially deployed in the flat. The task of the robot is to locate and deliver a desired item to the user (i.e., the inhabitant). Respective items can be dropped from a *cabinet* into a basket mounted on top of the robot. For this purpose, the cabinet contains several boxes with magnetically clamped flaps. The magnets are triggered from a WiFi-enabled embedded device.

Initially, an user instructs the robot by entering the desired item (e.g., "towel") using a Tablet PC that is accessible nearby. Using a wireless network, the robot can query the flat's map,

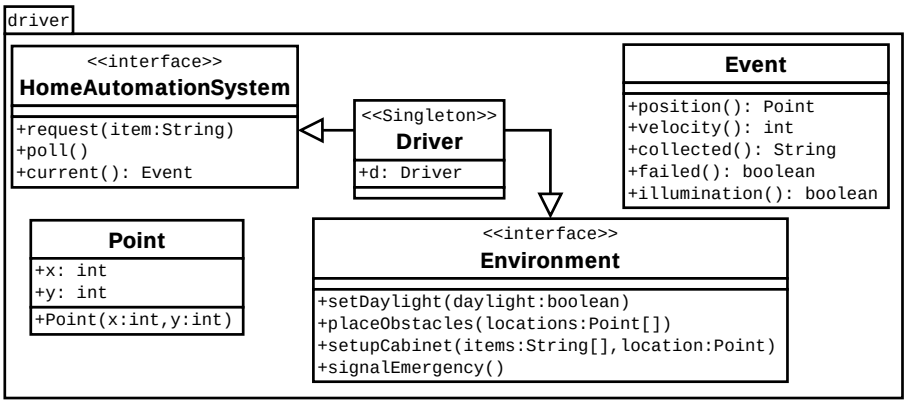

Fig. 2. Test driver interface.

available cabinets including their positions and contents. After this information was gathered, the robot is able to inform the user whether the desired item is available. Once the item was located, a route is planned and the robot starts driving. In this process, the robot has to avoid collisions with walls and other obstacles (symbolized by office chairs). After approaching a cabinet and parking in a predefined position underneath it, the robot signals the cabinet to drop the requested item. Afterwards, it drives back to the user. During the complete process, the environment may signal an emergency (e.g., a fire or medical emergency). In this situation, the robot is expected to drive to its emergency position as labeled in our illustration. Thus, it avoids obstructing access of human helpers to the inhabitant.

The following sensors and actuators are used to accomplish the robot's task:

- **Robot drive:** The robot drive has three modes for stopping (0=stop) and driving in arbitrary directions with two different velocities (1=slow, 2=fast).
- **Stereo camera:** Can be used to recognize walls and obstacles.
- **On-board computation unit:** The robot runs its operations on-board using a fix-installed netbook that connects to all the hardware on the robot.
- **Smart illumination system:** The flat is equipped with room lights that can be operated by the software system to improve the flat's illumination on demand. In this way, the object recognition performed using the stereo camera is supported.
- **Local WiFi:** The robot as well as the cabinet are connected to a wireless network. Thus, the flat's map and information about the cabinet's position and contents can be shared.

Furthermore, to improve its behavior, assure safety and minimize operation time, the following *adaptations* are possible:

- **Improve illumination:** If the robot enters a room and daylight from the windows is not sufficient for object recognition, the robot connects to the illumination system and activates it. After delivery, the supporting illumination is switched off again.
- **Location-dependent velocity:** While driving at fast mode velocity, the robot is not able to stop in time if an obstacle is detected. As the obstacles' positions may change, the robot is expected to run in slow mode during the current request as long as the current position was not explored during this request.

In order to send input data to the real system and to verify its output during simulation or test case execution, a *test driver* is required. For our example, we implemented such a driver whose interface is depicted as UML Class Diagram in Figure 2. The class `Driver` holds a static instance `Driver.d` and implements two interfaces: Firstly, `Environment` provides methods to enforce an emergency signal, mock a light state, and setup obstacles and a cabinet. In order to reduce the scenario's complexity, we assume that the positions of the inhabitant and emergency locations as well as the room's layout are static.

Secondly, the interface `HomeAutomationSystem` can be used to request a new item for delivery or to retrieve events that can be verified during simulation. The driver's event-based architecture allows verifying changes of the system without surveying it actively during the whole test execution. Changes in the environment can be tracked by investigating multiple events within one verification action. Therefore, events only have to be produced when the environment changes. Each instance of class `Event` captures information about the current position, velocity, and illumination. It also informs whether an item was collected or the search has failed.

Only a subset of the driver's functionality can be automated. Especially, for obstacle placement and cabinet setup, a dialog is shown to the test engineer, which lists instructions on necessary manual manipulations. All other functions are implemented using the system's sensors (brightness, velocity, etc.) and a wireless-switchable light bulb for change of illumination.

III. Validating SAS by Using an Adaptive Simulation Model

In this section, we present our methodology. The briefly discussed challenges are tackled in different components of a black box simulation model. These components, as well as their dependencies, are depicted in Figure 3. Each component matches a set of specific concerns that were separated in order to decouple the responsibilities during the design process. The

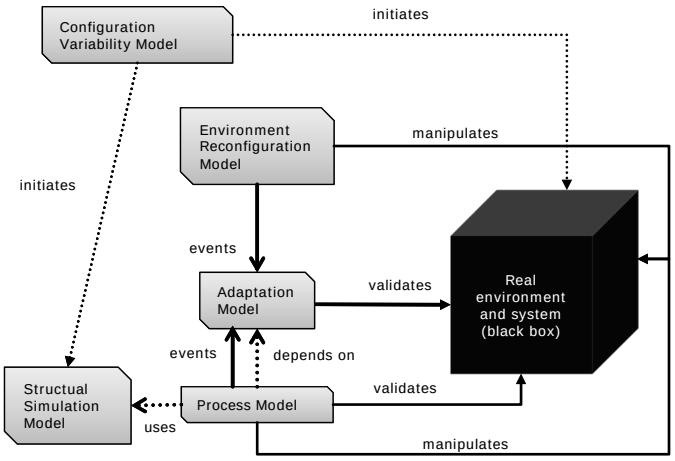

Fig. 3. Concern-separated components of the simulation model.

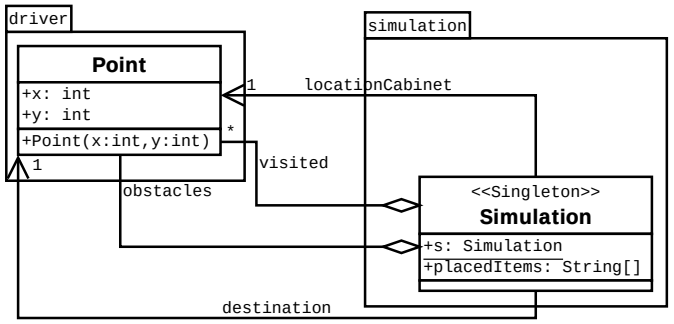

Fig. 4. Structural simulation model.

model is as much as possible based on Unified Modeling Language (UML) 2 [9], Object Constraint Language (OCL) [10] and a special version of equivalence class trees [11]. Actions are formulated as Java method calls.

The current state of the test execution is represented by the *Structural Simulation Model* (i.e., a UML class model). Based this state, the operational model of the running test scenario is given by the *Process Model* that is represented as state chart. The actions performed during execution are, firstly, the requests that are sent to the test driver and, secondly, *assertions* that determine whether the received events are correct in the current state. Thus, the state of the simulation model represents assumptions on the state of the real system. During the initiation of the system, the environment is set up and, synchronously, the Structural Simulation Model is configured with information that reflects this initial environment setting. As there may be different variants of initial configurations, the *Environment Variability Models* defines an equivalence class tree that allows to derive such configurations. The *Environment Reconfiguration Model* contains state charts with actions that define environment manipulations in order to trigger adaptation in the real system. As it defines an operational order of manipulations, requirement (3)–*correct adaptation planning*— can be dealt with. Regarding the requirement (2) (cf. Section I), it has to be validated whether system correctly adapts to these changes. Therefore, the Environment Reconfiguration Model produces events that are consumed by an *Adaptation Model* that reflects adaptation modes and validates them using assertions (requirement (5)–*consistent adaptation execution*). This Adaptation Model is a state charts as well. Events can also be produced by the Process Model and its behavior can be tailored to the Adaptation Model's state. Thus, requirement goal (4)–*consistent adaptation/system interaction*–is matched.

Basically, data flow between all model components follows the Counter Feedback Loop (CFL) that we claimed to be a central requirement to SAS test approaches in [12]. CFL proposes that a test system has to work vice versa to SAS feedback loops: Instead of monitoring the environment and deducing adaption decisions, an CFL-based based test workflow triggers actively manipulations on environment properties and monitors the SUT's reaction. CFL separates the task of a test system into four periodically-executed steps that are all matched within the proposed components:

1) **Change:** Environment configuration variability model and environment reconfiguration model explicitly define how environment properties can be changed in order to stress to system.

2) **Causal connection:** By exchanging symptom events between environment reconfiguration model and adaption models, the causal connection between change and self-adaption is modeled such that state-dependent adaptations can be verified.

3) **Adaptation plans:** Within the adaption models, accepted events can trigger multiple actions that can be used to describe what parts of the system are expected to be adapted and how. The adaptation outcome must be able to be monitored using the test interface.

4) **Service specification:** In the process model it has to be specified how the performed services of the SUT behaves from a black box perspective according to the reached adaption mode.

In summary, all of our components are designed along the CFL. The details of the individual model components are explained in the following.

A. Structural Simulation Model

During the simulation, several assumptions on the real system have to be managed that are represented by a simulation state. For our example application, the locations of obstacles and the cabinet has be remembered as well as the locations that were already visited. This state is captured by a structural model as depicted in Figure 4. The singleton object `SimulationState.s` holds attributes and aggregates objects that can be manipulated or evaluated by the central Process Model. All (two-dimensional) positions are stored in form of instances of class `Point`.

B. Process Model

The *Process Model* defines the task-specific behavior of the system and how it interacts with its adaptation feedback loops. For our example, we defined these aspects in an

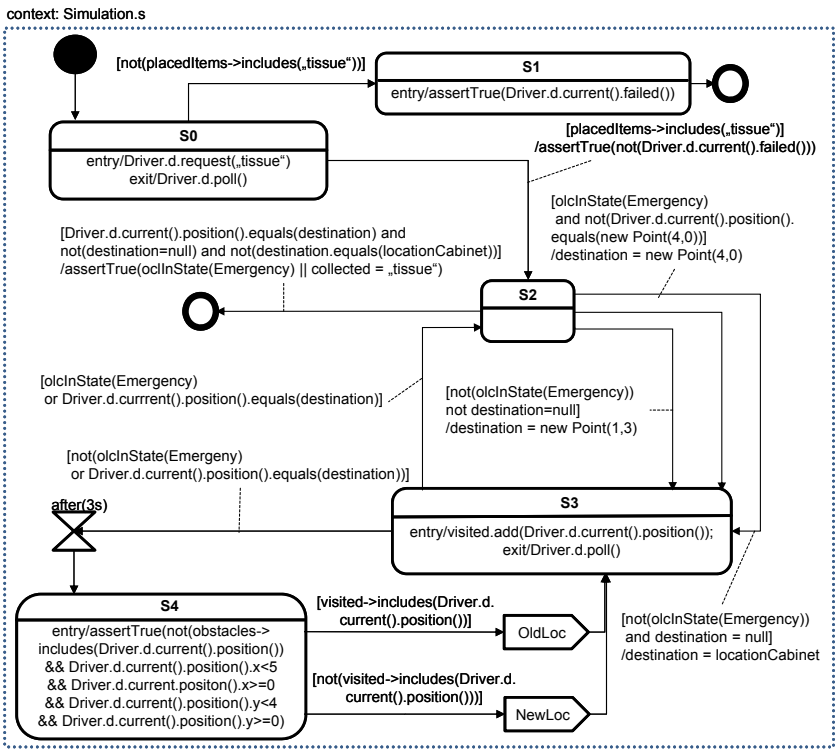

Fig. 5. Process model.

UML State Chart as depicted in Figure 5. The representation uses OCL constraints whose context is the static instance `Simulation.s`. In state `S0`, a request for a towel is initiated and the first event is polled. If the initial configuration set up the cabinet with the desired item, `S1` is reached, otherwise `S2`. The action of the latter transition (i.e., the entry action of `S1`) performs an assertion on whether the real system has either failed or not. If any assertion in the models fails, the simulation is cancelled and an error is signaled. Starting from state `S2`, the robot's destination is determined by evaluating the previous destination value (either null, the start place, the cabinet's place or the emergency position).

States `S3` and `S4` form a feedback loop. When entering `S3`, the current position is appended to the list of visited locations and the next event is polled. In the next step, the loop sleeps three seconds (indicated by the `AcceptTimeAction`, cf. UML spec. [9]). Thus, the Adaptation Models are expected to enforce changes to the environment that are interleaved with the process. Subsequently, in `S4` an assertion is performed in order to ensure no obstacle has been hit and the robot did not leave the boundaries of the scenario. Depending on whether the current position is contained in the `visited` collection, a signal `OldLoc` or `NewLoc` is produced. Therefore, we use the `SendSignalAction` UML element. These signal events are later used to synchronize with the adaptation models. At this point, the feedback loop is restarted. As soon as the destination is reached, the transition to state `S2` is triggered. Another exit possibility from the loop is triggered when the `Emergency` adaptation mode is active. This information can be queried by the `oclInState(...)` function, which is applied to the Adaptation Models. In this way, an interaction between the task-related process and the adaptation mode of the SAS can be modeled. The final state is enabled if the robot reaches a destination that is not the location of the cabinet. The respective transition checks an assertion whether either an emergency was signaled or the correct item was collected.

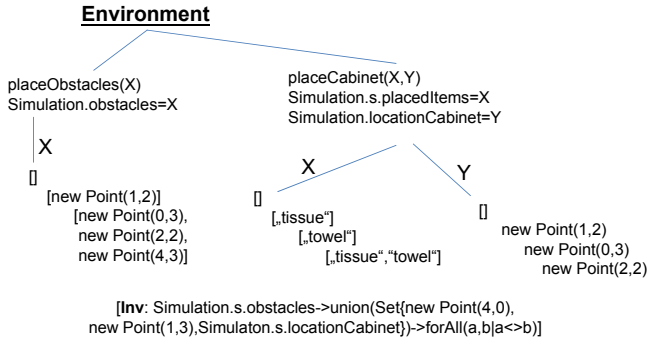

Fig. 6. Environment configuration variability model.

C. Environment Configuration Variability Model

The state space of an environment situation can be enormously large. In testing, this problem is usually dealt by using classification. For instance, data ranges of the system's input parameters are split into equivalence classes and only representatives are tested. All representatives of an equivalence class are assumed to produce the same output. For our example, we designed a dedicated model as depicted in Figure 6. The hierarchical structure serves as a decision tree for determining under which initial conditions a simulation can be started.

Each one of the `Environment` child nodes performs multiple operations: Firstly, the real system is initiated (e.g., the robot is set up in its initial location) and secondly, the simulation state is manipulated such that it reflects this initial configuration. The operations are parameterized with one or two substitution variables. Each variable can be replaced by one of the concrete values in its leaf nodes. The latter ones are the equivalence class representatives. Furthermore, the model contains an invariant to prohibit configurations where the robot's start position, obstacles, or the cabinet are put in the same location.

Basically, this model represents the variability of possible environment settings. Thus, more sophisticated models of variability (e.g., attributed feature models [13]) can also be used for the same purpose. Inherent invariants of such models can restrict the configuration variability space to a manageable size. However, a specific challenge of SAS is to validate whether the system adapts correctly to changes of this configuration. Therefore, in the next section, the configurations dynamics are defined.

D. Environment Reconfiguration Models

The left part of Figure 7 depicts a model of environment reconfiguration. In the upper chart, the entry point of the first state sets the environment daylight to `true`. The driver is now in charge of mocking the brightness sensor's input data and, thus ,enforces the system to adapt. In order to reflect the expected adaptation in the simulation model, a signal `Day` is produced that later will be received by the Adaptation Model. After five seconds, the daylight setting is inverted and the `Night` signal is sent. After additional two seconds, the reconfiguration loop restarts. The lower chart performs a loop that every three seconds demands the simulation to decide of an emergency is signaled or not. This decision can, for instance, be determined randomly or by the user.

Using such environment reconfiguration models, scenarios with different operational orders can be generated. Based on these scenarios, the SUT is stressed and its reactions are exhaustively validated. Using timing, the variety of interleaving possibilities with actions from the Process Model can be reduced.

E. Adaptation Model

Adaptation models define how a configuration has to be altered in response to a received signal. Signals are produced by either the Environment Reconfiguration Models or by the Process Model in order to notify about a condition that may cause an adaptation. The left part of Figure 7 depicts three state charts for the velocity, illumination, and emergency adaptations.

States of an adaptation state chart may contain an entry operation, which performs a validation on the system's adaptation mode. Using UML `AcceptEventActions`, the automaton is designed to wait for the signals. After a signal was received, a new system event is retrieved (`poll()`) such that the assertion is performed on a fresh information basis. Each Adaptation Model stores a specific aspect of the SUT's adaptation mode. Behavioral adaptations are defined using constraints on the Adaptation Models' states.

IV. A Common Infrastructure for Simulation and Test Case Generation

The constructed model can be used for both test case generation and simulation. For test case generation, generators perform a *reachability analysis*, which produces a reachability tree. The tree's root node represents the initial system's state; child states can be discovered by state-changing actions. Each path from the tree's root to a terminal leaf (i.e., where no new actions can be performed) forms a unique test case. The depth of this tree can not only be enormously large but also potentially infinite due to loops and actions without conditions within the modeled control flow. Furthermore, the tree's breadth grows with the degree of indeterminism in each state (i.e., the number of child states respectively applicable actions). Thus, an adequacy criterion is applied that restricts the number of deducible test cases by certain kinds of *adequacy criteria* in form of quantitative measures on the model's elements (e.g., number of states or transitions) or the resulting test cases (e.g., number or length). The benefit of the generation-based approach is that the generated test suite can be re-run for new versions of the examined SAS in the sense of regression testing. As a result, test coverage and test results can be compared quantitatively.

Alternatively, in simulation, no fixed test suite is maintained. In comparison to the generation approach, only one path through the reachability tree is traversed. Therefore, the model is directly executed by an interpreter. If multiple actions can be performed in a single state (a path branches), this indeterminism can be resolved by a human tester or an automatic mechanism (e.g., a heuristic based on a coverage criterion). The major advantage of simulation over generation is the ability the react to change of sensed environment properties. This capability can be used for elements, whose behavior cannot be modeled as precisely as necessary for predicting output. For instance, the movement of physical objects through space is such dynamic that steering it according to generated test data is too expensive or even not possible at all. If the tested system monitors the object's location and the expected reaction to this property has to be tested, it is more effective to deploy the real object, monitor its location and use this data as input for the prediction of expected reaction. In this manner, the real object is taken "in the loop". In generated test cases, a reaction to such properties is not possible as assertions cannot be defined depending on past observations.

For simulation, the metamodel only requires a small set of additional concepts. The model must allow computing test input from monitored data of the in-the-loop objects and for verifying test output based on this run-time information. Therefore, access to query operations of the test interface has to be provided. The rest of the model's capabilities is equivalent for simulation.

As presented, both approaches have their pros and cons. Hence, we constructed an infrastructure that enables test engineers to make use of both methods. Figure 8 depicts involved artifacts, data flow and processing actions in this infrastructure. SUT and environment are accessible via a test interface as presented in Section II. The SUT was built from a design model, which was constructed according to a set of requirements. The requirements document is also the foundation for the validation model (i.e., the test or simulation model). If certain environment objects have to be tested in the loop, query calls to the interface have to be embedded to the validation

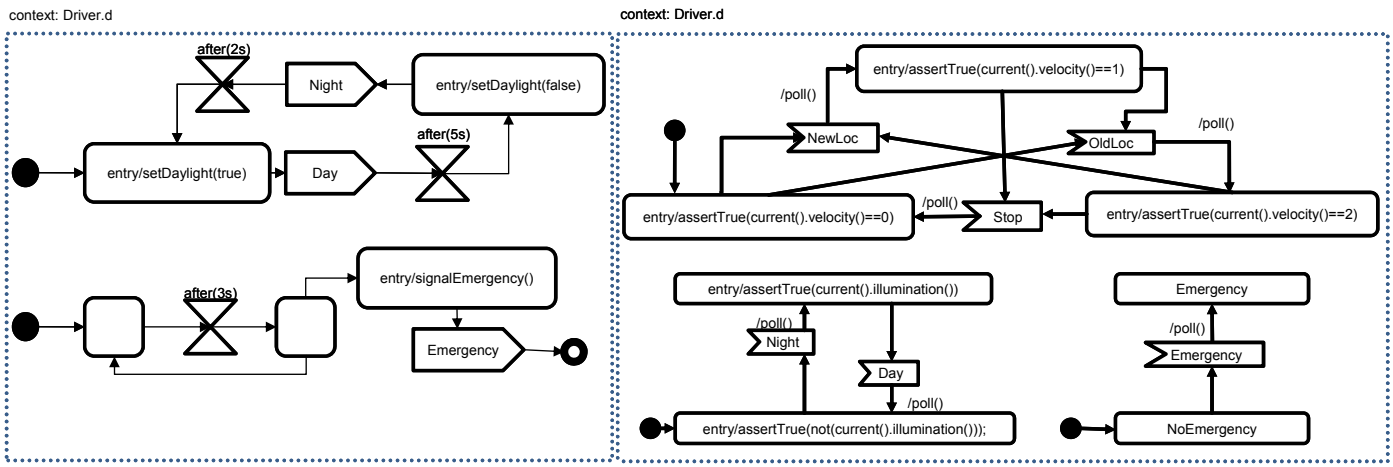

Fig. 7. Left: Environment reconfiguration model. Right: Adaptation models.

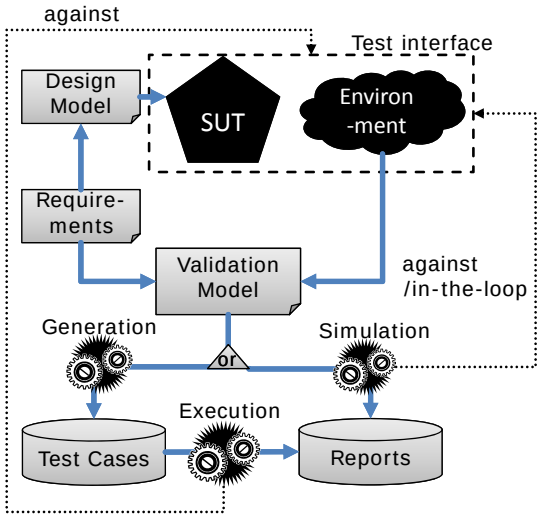

Fig. 8. Simulation and generation infrastructure.

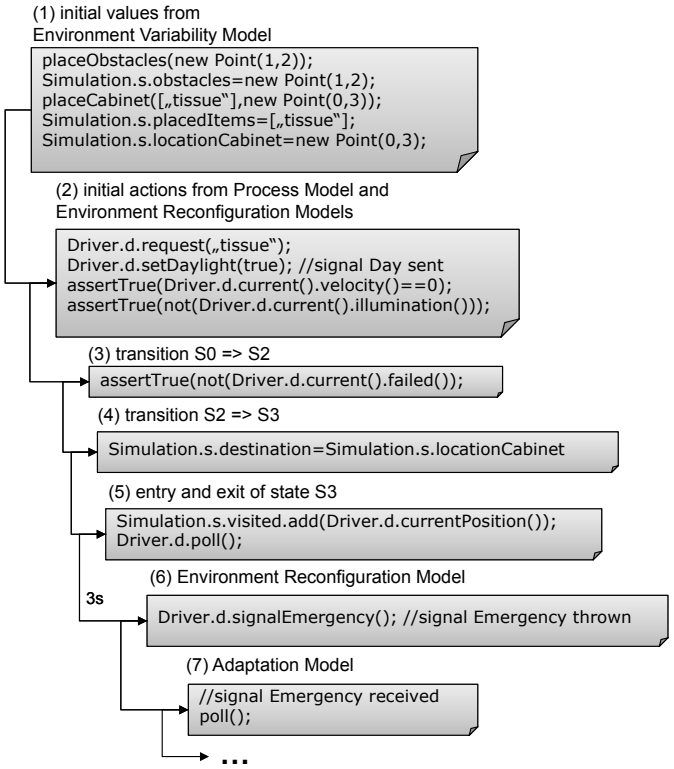

Fig. 9. Excerpt of an example simulation run.

model as well. After specifying the model, it is given as input to either the generator or the simulation engine. The generator produces a set of test cases, which can be run manually or automatically against the exposed test interface. The simulator instead only traverses one trajectory through the state space. Both types of execution results contain the list of performed steps as well as verdicts (e.g., PASS, ERROR, INCONCLUSIVE). In this way, the output of both approaches can be utilized in the following quality improvement.

V. Simulation Example Run

To clarify the models' interactions, we illustrate an excerpt of an example simulation run in Figure 9. The simulation is indeterministic as there can be several execution paths. Sequence (1) of operations is generated by the Environment Variability Model. The simulator automatically selects a solution of the model's invariant such that no obstacle position equals the positions of the inhabitant, cabinet, or emergency stop. When the different state charts are initiated, operations sequence (2) is performed as defined in the initial states. When the Environment Reconfiguration Model sets the daylight property, a signal `Day` is produced. However, as the respective Adaptation Model has no matching outgoing transitions in its initial state, this signal is ignored in this specific state. Sequences (3) and (4) are generated when the transitions `S0->S2` and `S2->S3` are triggered. `S0->S1` cannot be executed as the tissue item was placed in the cabinet during operation of sequence (1). Subsequently, in sequence (5) the entry and exit action of `S3` are executed. After this point, the Process Model waits for three seconds as defined and, consequently, there is an indeterministic decision point in the Environment Reconfiguration Model where either an emergency is signaled or not. We assume that the

simulation determines to generate the emergency such that in sequence (6), the driver is called and the respective signal is produced. In sequence (7), the Adaptation Model receives this signal and switches to the emergency mode after polling a new event. Afterwards, the simulation starts validating whether the robot correctly drives to the emergency stop.

For test generation, all possible trajectories through the state space would be searched and saved as test cases. The number of these test cases is then restricted by a test adequacy criterion such as state or transition coverage. The model of our example is not appropriate for test case generation as it introduces decisions based on runtime information. In particular, the position of the robot is not predicted but constantly queried as it cannot be modeled with an adequate precision. Therefore, the location property has been taken ”in-the-loop” here.

VI. Implementation and Experimental Environment

Syntax and semantics of all used models were implemented in our *Model-driven Adaptivity Test Environment* (MATE). Figure 10 shows a screenshot of its graphical user interface. MATE is an integrated test environment including the following components:

Metamodel implementation: All models proposed here require a metamodel containing concepts to be instantiated. For this purpose, the Eclipse Modeling Framework (EMF, [14]) was used. Besides UML and OCL, a metamodel for creating instances of environment variability models was required.

Model editors: Model construction is enabled by a set of graphical editors. These editors not only support drawing UML and the variability model but also include parsing of the textual elements into their abstract syntax (cf. Figure 10, marking ①).

Test case generator: Using the created models, test cases can be generated. Therefore, an interpreter implements the metamodel's semantics, traverses through the state space and produces one test case for each termination reached. In order to restrict the generation time, the maximal number of test cases can be specified as well as different adequacy criteria.

Interactive simulator: In order to run simulations, the interpreter can be run synchronously with the system instead of generation. Decisions can be either delegated to a heuristic algorithm or performed manually. Heuristics can be implemented project-specific using an appropriate interface. During simulation, the user can visualize the current execution state within the model editors and inspect the state's constituting variable assignments. The simulator's user interface shows a reachability tree, which can be inspected as well (cf. Figure 10, marking ②). Thus, the tester can interact with the interpreter and find situations where variability multiplies the number of sub-paths.

Automation framework: As our approach and tooling is designed to be generic for SAS, it must be reusable. Due to this reason, a framework is provided that allows mapping of actions and queries to a concrete SAS platform. This framework can be used for both executing generated test cases and simulation.

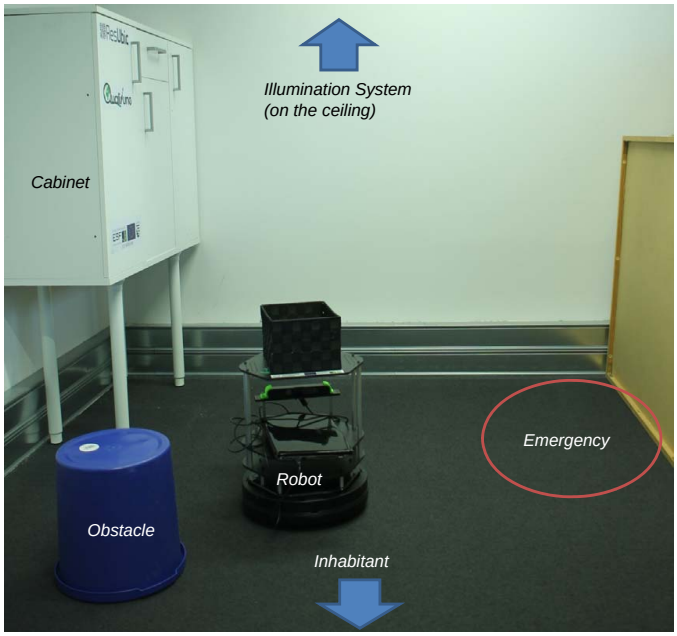

Fig. 11. The HomeTurtle lab.

Reporting tools: The system's quality can be evaluated statistically by reports exported from either executed test cases or simulation runs. Reports themselves are model-based and include verdicts as PASS, FAIL, ERROR, or INCONCLUSIVE. The relation of these verdicts among a set of test case reports can be visualized in appropriate bar diagrams (cf. Figure 10, marking ③).

All these tooling components allow engineers to perform the complete test modeling, execution, and reporting process within a single integrated test environment.

In our previous work, we developed the Smart Application Grid (SMAG) framework that can be used for architectural run-time adaptation [15]. Based on SMAG, we created the self-adaptive HomeTurtle software. An impression of the physical experimental environment is given in Figure 11. In order to show the feasibility of our validation approach, a platform-specific HomeTurtle test driver was developed as well. It directs the operation calls produced by the model to the real system and, vice versa, generates events from the system's observed behavior. However, not every modeled operation can be performed automatically. The initial configuration of the environment (setting up the cabinet's content, placing obstacles, etc.) and the validation whether the correct item was collected are performed manually by the test engineer. During phases, in which test execution had to be automated, the validation directly benefits from the model-driven nature of our approach. Its advantage in manually performed action is given by the reproducibility of simulation paths. If any path fails during a test, it can be recorded, analyzed and even be re-executed later on.

VII. Related Work

Comprehensive validation approaches for self-adaptive systems are still rare in literature. According to Salehie et

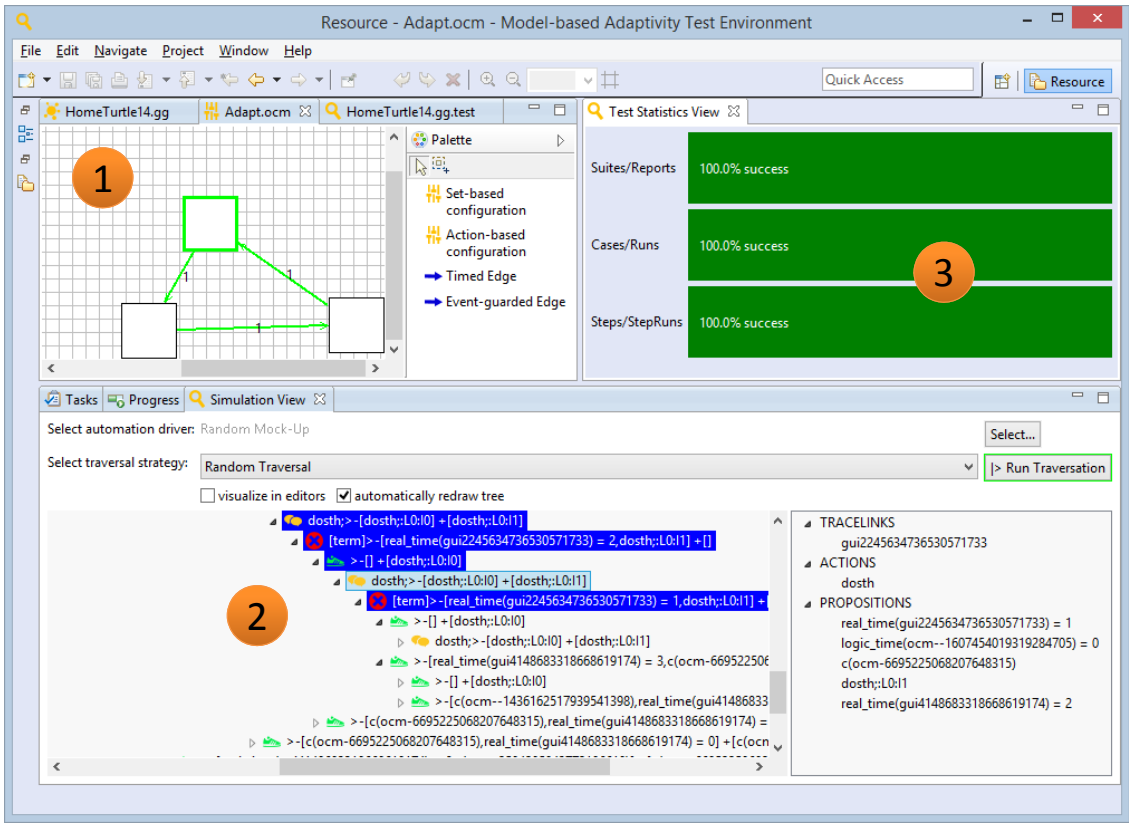

Fig. 10. Screenshot of MATE.

al., SAS leverage both self awareness and context awareness as primitive concepts for higher-level self-adaptive behavior [3]. For context-aware systems, several proposals targeting testing were made as well. The most basic approach in this sense was proposed by Kakousis within the MUSIC project [16]. In order to test a mobile application, a domain specific language was designed that allows for defining timed changes of context information, logging, and evaluation of memory usage. As already discussed, complex systems cannot be covered appropriately with manually defined test cases. However, in the set of inspected works, the MUSIC methodology is the only one that includes means for dealing with time.

Wang et al. discussed in [17] how a context-aware system can be tested by observing how certain context changes trigger adaptation. The basic assumption in their work is that the SUT is based on a context middleware and calls on this middleware's interface are enriched by calls to a dedicated monitoring framework. The points where these calls are made are called context-aware program points (capps). From the monitored execution, a control flow model of capps can be deduced. The context is then manipulated and manipulations are correlated with capps. The resulting information on which context changes trigger which capps can be utilized to generate appropriate test cases. Wang et al.'s method is helpful for stressing the system with good coverage. However, there is no oracle included in the approach, i.e., it cannot be automatically decided whether the triggering of capps was correct. Thus, this approach is less-powerful then our proposal. Another disadvantage is that the developer has to change the SUT's code as the monitoring framework has to be called. In consequence, the approach cannot be considered a pure black box method.

A simulation-based approach was proposed Abeywickrama et al. [18][19]. For SAS design, the authors created a modeling environment SOTA (State of the Affairs) where feedback/control loops can be specified directly in form of hierarchical state charts. In order the examine such systems' correctness, an interactive simulator was added, which visualizes the execution of these models. Thus, the system engineer can observe incorrect states and give manual input where context information is expected. As this approach again does not include any automatic oracle, it can rather be compared to debuggers that execute a program step-wise in order to analyze it manually.

Fleurey et al. showed in [20] how an SAS can be built based on variant models, context variables, and adaptation rules. They also recognized the need for simulation when such systems have to be validated. Thus, they derived a simulation graph and validated it against invariants. In contrast to our black box approach, their validation method is based on design models, which makes it hard for testers to decide whether failures stem from design or implementation.

An advanced strategy was proposed within the DiVA project [21]. The validation of DiVA-based implementations can be performed in two phases: (1) In the early phase, instances of the context model are generated and associated with partial solutions. Those describe how parts of the systems have to be configured after a certain context instance was applied and the corresponding adaptation was performed. (2) In an operational

validation phase, the system's behavior is investigated during a sequence of contextual changes. Sequences are automatically built by a heuristic (so called Multi-Dimensional Coverage Arrays, MDCA). The DiVA validation methods neither consider any system/adaptation interaction, nor do they propose specific test models for constructing an automatic test oracle. Furthermore, in [22], the DiVA authors Munoz and Baudry alternatively propose using statistical models for generating sequences of context changes. They also consider the possibility to define formalized oracles based on direct associations between a change history and an expected adaptation. For a rather complex systems, the management of such associations is much more expensive than with our model (i.e., each possible adaptation has to be modeled separately).

Nehring and Liggesmeyer proposed in [23] a process for testing the reconfiguration of adaptive systems. The validation is performed in six iterations: In the first iteration, a system model is derived and representative workload is prepared by a domain expert and later executed by developers or system engineers. In the second iteration, a system architect checks if structural changes are performed correctly. Thereby, the reconfiguration actions have to be in the correct order such that the system ends in a valid state and the quality of service is only affected minimally during reconfiguration. The third iteration considers data integrity while stressing the system with increasing load. In the fourth iteration step, state transfer between replaced components is investigated. An interaction issue between system transactions and the adaptation is tested in the fifth iteration. The last iteration considers the identity of components and component types before and after adaptation. In comparison to our approach, Nehring and Liggesmeyer assume the adaptive system to be component based and the validation can be sufficiently investigated by a debugger-like tool chain. Thus, their approach is exploratory and hard to use for integration and system testing.

Furthermore, in the area of self-organizing systems, Eberhardinger et al. proposed an approach called *Corridor Enforcing Inrastructure (CEI)* [24]. As MATE, the CEI concepts rely on running a feedback loop that monitors the test object and checks its validity continuously. In contrast to MATE, CEI is more focused on non-functional qualities of the SUT as the definition of validity is based on constraints. The constraints indicate a *corridor of correct behavior (CCB)*, which the SUT is expected never to leave. By extracting *environmental variation scenarios (EVS)* from the SUT's specification, the system is examined in different situations. EVS extraction can also be automated by the use of a model checker that finds sequences of interactions within communication protocols. Eberhardinger et al.'s approach is well-suited for this kind of system and especially for checking non-functional properties of self-organizing systems. In comparison to MATE, it is not yet clear, how the functional complexity of SAS can be tackled and concisely modeled within the CEI concepts.

VIII. Conclusions and Future Work

In this paper, we presented a concept to build black box simulation models for validation of SAS. The models are separated in different components, each matching a certain step inside the proposed Counter Feedback Loop principle. We showed how these interacting model artifacts can be employed for both simulation and test case generation. For this purpose, an enabling infrastructure was outlined. Both test case and simulation execution can be reported in the same manner. Furthermore, we illustrated an excerpt of a simulation run along our example model. The system was implemented and is deployed in our lab.

Our models are based on UML class models, state charts, and equivalence class trees with invariants. Automata communicate by events such that the different concerns of the scenario process and adaptation can be separated. Our approach does not rely on any design model such that engineers are able to build discrete simulation models of arbitrary self-adaptive systems. The methodology comprises a process of classifying environment variability and defining an explicit model on its change. Using this toolset, we match the goals (2)-(5) as stated in Section III. Goal (2)–*Correct adaptation initiation* is considered by letting Adaptation Models receive signal events from the Environment Reconfiguration Models. Thus, the change in context can be causally connected with an adaptation of the system. As Adaptation Models define an operational order of adaptation actions, goal (3)-*Correct adaptation planning* is dealt with. Goal (4)-*Consistent adaptation/system interaction* can be validated as the Process Model accesses the state of the Adaptation Models and defines conditions on this state. Thus, the system's adaptive behavior can be defined. As Adaptation Models can also check an adaptation's outcome by assertions, goal (5)-*Correct adaptation execution* is addressed.

In our future work, we are going to enrich the employed formalism (i.e., state charts, equivalence class trees, etc.) for more compact definitions and experiment with more complex scenarios in order to expand the evaluation. Concerning the improvement of formalism, e.g., we consider using Petri nets as they are more flexible in describing parallelism and synchronization, which is especially important when multiple widely-independent system parts interact.

Furthermore, we considered that it may be beneficial to provide alternative environment reconfiguration model types. While state charts can only model very less-complex and explicitly specified states, data graphs, movement profiles, or event differential formulas could provide a more dynamic representation. For instance, with graphs and differential formulas, data changes can be correlated with discrete simulation time precisely. Instead, changing locations of objects that effect the SUT could be modeled using pre-defined paths that are triggered by simulation time as well.

In summary, we aim at providing a complete test generation and simulation environment that can be employed for almost arbitrary SAS. Our central assumption is that all considerable SAS comply with the MAPE-K loop principle. In order to evaluate this proposition, further case studies will be performed in future work as well. Different scenarios with autonomous systems are considered for this purpose, e.g., SAS-controlled drones and automotive systems.

Acknowledgment

This work is funded within the projects #100084131 and #100098171 (VICCI) by the European Social Fund as well as CRC 912 (HAEC) and the Center for Advancing Electronics Dresden (cfaed) by Deutsche Forschungsgemeinschaft.

REFERENCES

[1] G. Püschel, C. Piechnick, S. Götz, C. Seidl, S. Richly, and U. Assmann, "A black box validation strategy for self-adaptive systems," in Proceedings of The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE). XPS Press, 2014, pp. 111–116.

[2] B. H. C. Cheng et al., "Software engineering for self-adaptive systems: A research roadmap," in Dagstuhl Seminar 08031 on Software Engineering for Self-Adaptive Systems, 2008, pp. 1–26.

[3] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," ACM Transactions on Autonomous and Adaptive Systems, vol. 4, no. 2, May 2009, pp. 14:1–14:42.

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

[5] IABG, "V-Modell XT 1.4," http://v-modell.iabg.de, visited 04/01/2014, 2012.

[6] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, 2010.

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

[8] "TurtleBot 2," http://turtlebot.com, visited 04/01/2014.

[9] Object Management Group (OMG), "Unified Modeling Language (UML) specification, version 2.4.1," http://www.omg.org/spec/UML/2.4.1/, visited 04/01/2014.

[10] Object Management Group (OMG), "Object Constraint Lanugage (OCL), version 2.3.1."

[11] M. Grochtmann, "Test case design using classification trees," Proceedings of STAR, vol. 94, 1994, pp. 93–117.

[12] G. Püschel, S. Götz, C. Wilke, C. Piechnick, and U. Aßmann, "Testing self-adaptive software: Requirement analysis and solution scheme," International Journal on Advances in Software, ISSN 1942-2628, no. vol. 7, no. 1 & 2, year 2014, 2014, pp. 88–100.

[13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," DTIC Document, Tech. Rep., 1990.

[14] "Eclipse Modeling Framework Project," http://www.eclipse.org/modeling/emf/, visited 04/01/2014.

[15] C. Piechnick, S. Richly, and S. Götz, "Using role-based composition to support unanticipated, dynamic adaptation - smart application grids," in Proceedings of The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE). XPS Press, 2012, pp. 93–102.

[16] K. Kakousis, N. Paspallis, G. A. Papadopoulos, and P. A. Ruiz, "Testing self-adaptive applications with simulation of context events," Electronic Communications of the EASST, vol. 28, 2010.

[17] 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.

[18] D. B. Abeywickrama, N. Bicocchi, and F. Zambonelli, "SOTA: Towards a general model for self-adaptive systems," in Enabling Technologies: , IEEE 21st International Workshop on Infrastructure for Collaborative Enterprises (WETICE). IEEE, 2012, pp. 48–53.

[19] 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. New York, NY, USA: ACM, 2013, pp. 67–76.

[20] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel, "Modeling and Validating Dynamic Adaptation," in Models in Software Engineering. Springer, 2009, pp. 97–108.

[21] 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.

[22] F. Munoz and B. Baudry, "Artificial table testing dynamically adaptive systems," CoRR, vol. abs/0903.0914, 2009.

[23] K. Nehring and P. Liggesmeyer, "Testing the reconfiguration of adaptive systems," in Proceedings of The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE). XPS Press, 2013, pp. 14–19.

[24] B. Eberhardinger, H. Seebach, A. Knapp, and W. Reif, "Towards testing self-organizing, adaptive systems," in Testing Software and Systems. Springer, 2014, pp. 180–185.