A Black Box Validation Strategy for Self-adaptive Systems

Georg Püschel, Christian Piechnick, Sebastian Götz, Christoph Seidl, Sebastian Richly, Uwe Aßmann
Software Technology Group, Technische Universität Dresden
Nöthnitzer Str. 46, 01062 Dresden, Germany
Email: {georg.pueschel, christian.piechnick, sebastian.goetz1, christoph.seidl, sebastian.richly, uwe.assmann}@tu-dresden.de

***Abstract*—Self-adaptive systems are able to operate autonomously by reconfiguring themselves for changing context conditions and tasks. This capability requires a process of decision making that can only be partially hard-coded. Some parts of the logic are the result of reasoning and, thus, implicit to the system designer or user. In consequence, the quality of the systems functionality has to be extensively validated before delivery. During the validation, firstly, the response of adaptation decisions as a result of environment change has to be examined. Secondly, it is necessary to check the interaction of adaptation and non-adaptation-related behavior. The management of all this information is expensive. Therefore, we propose an approach that separates environment change, functionality and adaptation concerns using expressive models. The models are executed by a simulator and validated against the real behavior of the system under test. We illustrate the complete approach using an example SAS operating a domestic service robot. Our design process and the proposed modeling principles equip engineers with a toolset that allows them to face the challenging complexity of self-adaptive system validation.**

Keywords*—*self-adaptive systems; service robots; model-based testing; simulation; feedback loops

I. INTRODUCTION

A Self-adaptive System (SAS) [1] adapts itself according to changes in its environment. The continuous execution of sensor monitoring, decision making, planning, and adaptation execution is organized in feedback loops [2]. Due to the use of intelligent reasoning strategies, the 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.

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 is that a SAS can be adapted from external or reason about unanticipated events can never be tested comprehensively in this phase of the life cycle.

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 [3] 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 validate the outcome. Thus, 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.

A validation method for specification-based black boxes is model-based testing [4]. In this approach, a test model is specified and test cases are generated from it. Additionally, a further problem is that SAS can be deployed in complex environments where not every detailed situation can be enforced. For instance, some entities the system is interacting with like hardware controllers or physical objects are difficult to be formalized. Instead, the test model designer may specify some future decisions depending on run-time state that is observed from these entities at runtime. As sequential test cases cannot support such decisions, the model has to be executed at run-time. Therefore, we propose using simulation and capturing the discussed non-specifiable parts of the system or test environment *"in-the-loop"*.

The challenge in simulating a SAS is to provide a meta-model that is expressive enough for compactly specifying all behavioral and adaptation-related aspects. These aspects are given by several requirements that we derived from failure scenarios in our previous work [5]:

(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. However, 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 [2]). In order to match the requirements, the model has to provide means for defining in which situations an adaptation has to be initiated (goal 2), how the system has to adapt (goal 3), how the adaptation has to be scheduled with non-adaptation-related behavior (goal 4), and how the end result of the adaptation is expected to look like (goal 5).

In order to match these requirements, we contribute a methodology to separate their different aspects in 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 concretely validated. We illustrate the complete modeling methodology using our HomeTurtle domestic robot. In the HomeTurtle scenario, a robot is deployed in a flat of a handicapped person and is

 ISBN: 978-1-61208-341-4

capable of delivering various items, which are stored in a software-controlled cabinet.

The remainder of this paper is structured as follows: In Section II, we start with our example adaptive system. In Section III, we present our approach based on this example. In Section IV, we illustrate an example simulation run. In Section V, we present out implementation and experimental environment. Afterwards, in Section VI, we discuss related work. In Section VII, conclusion and future work are discussed.

II. Example Application: Home Turtle

In this section, we present an illustrative example of a SAS that supports a handicapped person at home. The scenario is depicted in Figure 1. A service robot "HomeTurtle" (an extended version of the TurtleBot platform [6]) is initially deployed in the flat. The task of the robot is to find and deliver a desired item to the user (i.e., the inhabitant). Those items can be dropped from a *cabinet* into a basket mounted on top of the robot. Therefore, the cabinet contains several boxes with magnetically clamped flaps. The magnets are triggered from a WiFi-connected embedded device.

In the beginning, a 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, available cabinets including their positions and contents. After this information has been gathered, the robot is able to inform the user whether the desired item is available. If the item has been found, 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. Additionally, 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 to obstruct the 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, obstacles, and the cabinets.
- **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.
- **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.

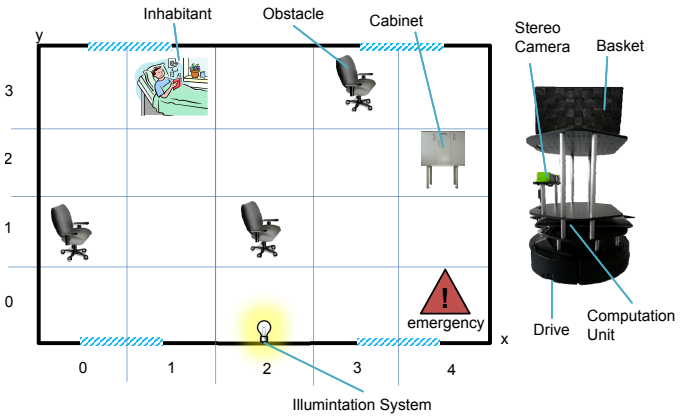

Fig. 1. Scenario: HomeTurtle operating in a flat.

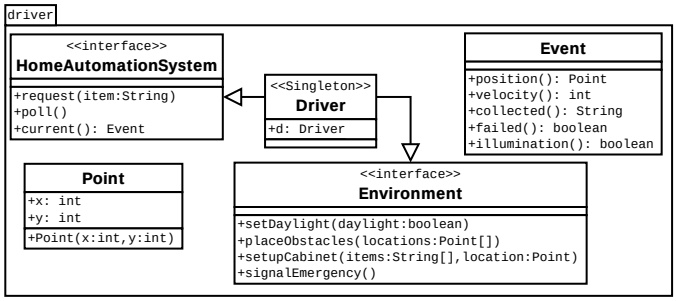

Fig. 2. Test driver interface.

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.
- **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 validate its output, the simulation has to communicate with the system using a test driver. For our example, we implemented such a driver whose interface is depicted in Figure 2. The `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 from the robot or to retrieve events that can be validated during simulation. Each `Event` captures information about the current position, velocity, and illumination. It also informs whether an item was collected or the search has failed.

 ISBN: 978-1-61208-341-4

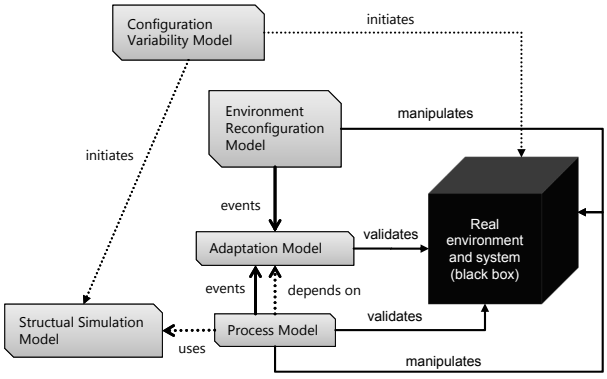

Fig. 3. Concern-separated components of the simulation model.

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 model is as much as possible based on Unified Modeling Language (UML) 2 [7], Object Constraint Language (OCL) [8] and a special version of equivalence class trees [9].

The recent state of the performed scenario is reflected by the *Process Model* that is based on state charts. 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. In order to work with more detailed state-defining information, the *Structural Simulation Model* (i.e., a UML class model) is used. 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. The details of the individual model components are explained in the following.

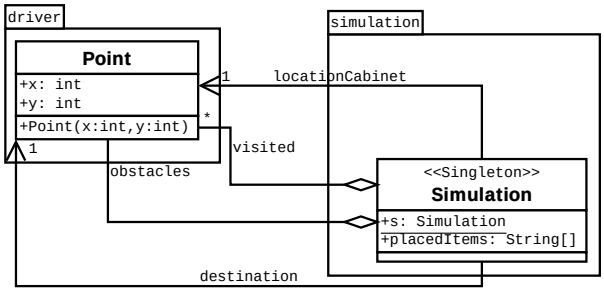

Fig. 4. Structural simulation model.

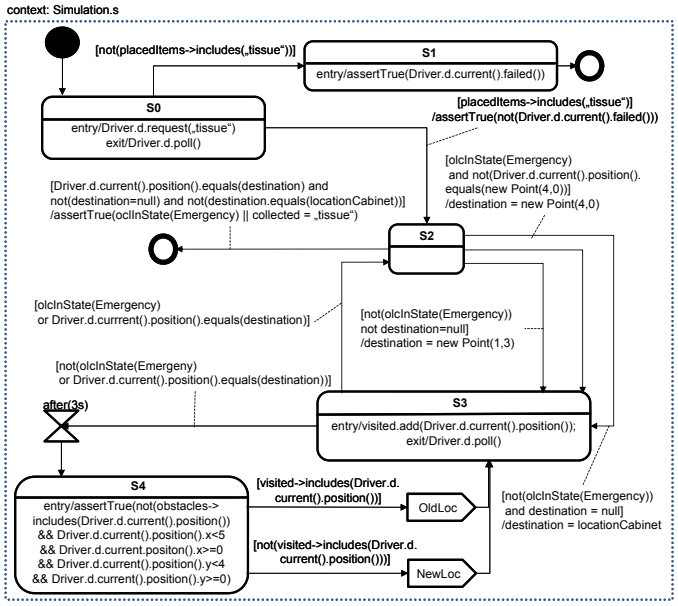

Fig. 5. System process model.

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 System Process Model.

B. System Process Model

The *System 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 UML State Chart as depicted in Figure 5. It 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

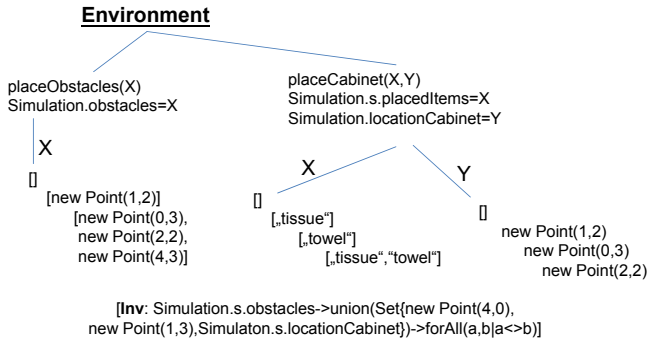

Fig. 6. Environment configuration variability model.

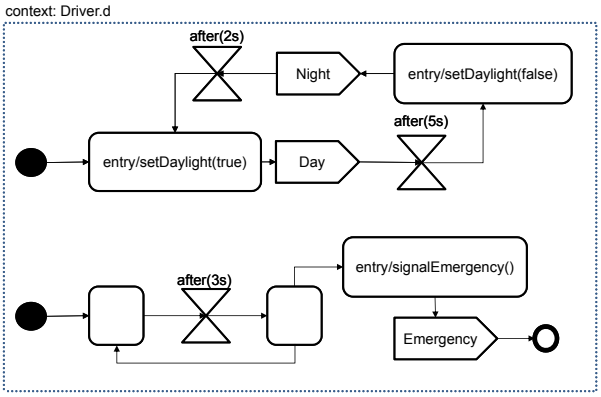

Fig. 7. Environment reconfiguration model.

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. [7]). 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.

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 special 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 [10]) 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 to system adapts correctly the changes of this configuration. Therefore, in the next section, the configurations dynamics are defined.

D. Environment Reconfiguration Models

Figure 7 depicts a simple 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 signalled 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 have been produced by either the Environment Reconfiguration Models or by the Process Model in order to notify about a condition that may cause an adaptation. Figure 8 depicts three state charts for the velocity, illumination, and emergency adaptations.

 ISBN: 978-1-61208-341-4

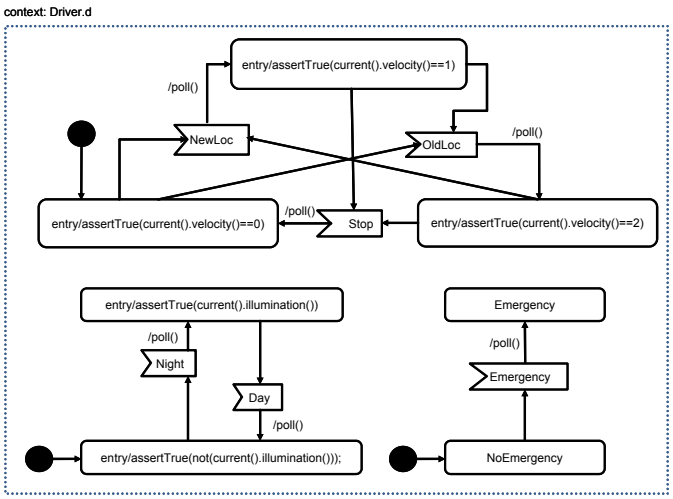

Fig. 8. Adaptation models.

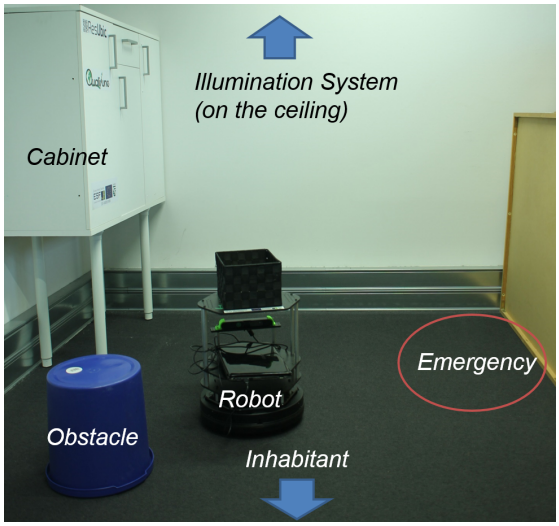

Fig. 10. The HomeTurtle lab.

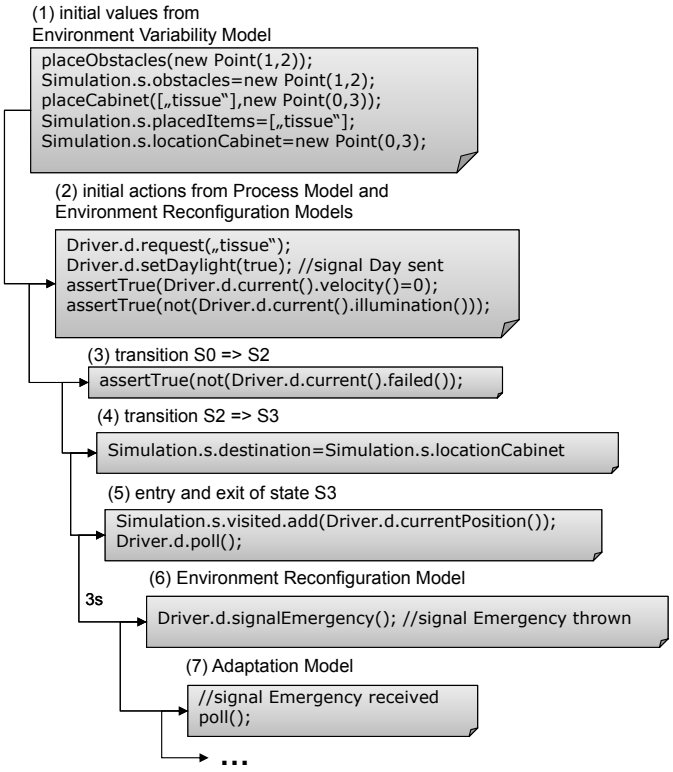

Fig. 9. Excerpt of an example simulation run.

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. Simulation

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 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 up 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.

V. Implementation and Experimental Environment

Syntax and semantics of all used models were implemented in our *Model-driven Adaptivity Test Environment* (MATE). It provides an EMF (Eclipse Modeling Framework [11]) based metamodel and a simulator that can be used to execute the model automatically or—in order to debug it—step-wise.

In our previous work, we developed the Smart Application Grid (SMAG) framework that can be used for architectural run-time adaptation [12]. Based on SMAG, we created the self-adaptive HomeTurtle software. An impression of the physical

 ISBN: 978-1-61208-341-4

experimental environment is given in Figure 10. 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 the automatable phases, the validation directly benefits from the model-driven nature of our approach, its advantage in manually performed action is given by the reproducability of simulation paths. If any path fails during a test, it can be recorded, analyzed and later even be re-executed.

VI. Related Work

Validation approaches for self-adaptive systems are still rare in literature. An advanced strategy was proposed within the DiVA project [13]. 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. The DiVA validation methods neither consider any system/adaptation interaction, nor do they propose specific test models.

Nehring and Liggesmeyer proposed in [14] a process for testing the reconfiguration of adaptive systems. The validation is performed in six iterations: In the beginning, 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 Liggemeyer 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.

VII. Conclusions and Future Work

In this paper, we presented a concept to build black box simulation models for validating SAS. Our models are based on UML class models, state charts plus equivalence class trees with invariants. Automatons communicate by events such that the different concerns of the system 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 requirements (2)-(5) as stated in Section III. Requirement (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. Requirement (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, requirement (5)-*Correct adapation 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, for instance, 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.

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] 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.

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

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

[4] M. Utting and B. Legeard, Practical model-based testing: a tools approach. Morgan Kaufmann, 2010.

[5] G. Püschel, S. Götz, C. Wilke, and U. Aßmann, "Towards Systematic Model-based Testing of Self-adaptive Software," in Adaptive, 2013, pp. 65–70.

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

[7] Object Management Group (OMG), "UML Specification, Version 2.4.1," http://www.omg.org/spec/UML/2.4.1/, visited 04/01/2014.

[8] Object Management Group (OMG), "Object Constraint Lanugage, Version 2.3.1," http://www.omg.org/spec/OCL/2.3.1/, visited 04/01/2014.

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

[10] 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.

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

[12] C. Piechnick, S. Richly, and S. Götz, "Using Role-Based Composition to Support Unanticipated , Dynamic Adaptation - Smart Application Grids," in Adaptive, 2012, pp. 93–102.

[13] 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.

[14] K. Nehring and P. Liggesmeyer, "Testing the Reconfiguration of Adaptive Systems," in Adaptive, 2013, pp. 14–19.

 ISBN: 978-1-61208-341-4