Automated Generation of Adaptive Test Plans for Self-Adaptive Systems

Erik M. Fredericks and Betty H. C. Cheng
Department of Computer Science and Engineering
Michigan State University, East Lansing, Michigan, 48824, USA
Email: {freder99, chengb}@cse.msu.edu

***Abstract*—Self-adaptive systems (SAS) can reconfigure at run-time to mitigate uncertainties posed by environments for which they may not have been explicitly designed. High-assurance SAS applications must continually deliver acceptable behavior for critical services, enabling the need for run-time validation techniques. To this end, run-time testing can provide additional assurance that an SAS will continue to behave as expected while executing under unknown conditions. This paper introduces Proteus, a framework for adaptive run-time testing on an SAS. Proteus facilitates both execution and adaptation of run-time testing activities to ensure that the SAS continues to execute according to its requirements and that both test plans and test cases continually remain relevant to changing operating conditions. We demonstrate our approach by applying it to a simulated self-adaptive remote data mirroring network that must efficiently diffuse data while experiencing adverse operating conditions. Experimental results suggest that Proteus can reduce the number of executed irrelevant, false positive, and false negative test cases at run time to ensure that online testing activities remain relevant as the SAS encounters uncertainty.**

I. Introduction

Self-adaptive systems (SAS) are often exposed to a number of environmental conditions that may prevent the system from satisfying its requirements. As such, SASs can self-reconfigure at run time to accommodate changing environmental conditions [1], [2], [3]. Testing an SAS should provide assurance that the new configurations satisfy requirements at design time and run time. Changing environmental conditions, however, may necessitate updates to both test suites and test cases developed at design time [4] to ensure testing relevance at run time [5], [6], where relevance indicates the applicability of a test case to its environment. This paper presents Proteus,[1] a requirements-driven approach for managing and dynamically adapting run-time testing activities based upon changing system and environmental conditions.

Despite the numerous techniques that have been developed for assessing the assurance of an SAS at both design time [7], [8], [9] and run time [8], [10], [11], [12], monitoring and adapting a test suite at run time to maintain relevance to changing system and environmental conditions has been largely unexplored. To this end, the MAPE-T feedback loop [13] was proposed to identify key elements of a testing framework for adaptive systems, where the testing elements and process should be adaptive to mitigate run-time uncertainty. A key challenge in performing run-time testing is *how* to adapt testing to the SAS as it reconfigures. Since test plan generation [14], [15], [16] is an essential aspect of the testing process, they should be explicitly defined for online testing [13] when assessing run-time assurance of an SAS.

Proteus is a framework for enabling online adaptive testing to address assurance at run time in the face of uncertainty. Proteus adapts both *test suites* and *test cases* at run time to ensure that both remain relevant to changing operating conditions, where a test suite comprises a collection of test cases. Moreover, Proteus defines an *adaptive test plan* for each operating context that comprises all possible test suites directly related to that operating context, that is, a configuration of system and environmental parameters. For instance, a test case that measures a specific value for the expected capacity of a data mirror may not be applicable if an unexpected incident causes damage to that data mirror's hard drive, thereby reducing its available capacity for data storage. If the damage to the hard drive is tolerable, then Proteus can adapt the expected value of the test case to reflect the new capacity. However, if the damage is sufficiently extensive that the hard drive fails, then Proteus can update test suites within the adaptive test plan to avoid testing the hard drive until it has been replaced. Adaptive testing can provide an approach for ensuring that test cases are relevant to their environment as conditions change, thereby increasing testing effectiveness. Within Proteus, test cases that monitor invariant, safety, and failsafe conditions are precluded from adaptation and always executed to ensure that the SAS is executing both safely and according to its invariant requirements. By managing the adaptation and execution of run-time testing for an SAS as it self-adapts, Proteus therefore provides a realization of the SAS MAPE-T run-time testing feedback loop [13].

Proteus enables both coarse-grained and fine-grained adaptation of test suites and test cases, respectively. Proteus provides coarse-grained testing adaptation by deriving test suites for each adaptive test plan at run time, where each adaptive test plan corresponds to a particular operating context. Each adaptive test plan comprises a default test suite and a set of automatically generated test suites, where the default test suite comprises all test cases relevant to the associated operating context. Proteus performs a *testing cycle* upon invocation of a new SAS configuration. Throughout a testing cycle, Proteus derives and executes test suites, monitors test

[1] In Greek mythology, Proteus was a deity that could change his form at will.

results, and invokes Veritas [4] to perform fine-grained test case parameter adaptation as necessary. Veritas is a run-time, online evolutionary approach for evolving test case parameter values to ensure that test cases remain relevant to changing conditions. A testing cycle is completed upon invocation of a new SAS configuration or determination that all executed test cases have successfully passed.

We demonstrate the use of Proteus by applying it to a simulation of a remote data mirroring (RDM) network that must replicate and distribute data to all mirrors within the network while experiencing network link failures and dropped or delayed messages. Experimental results indicate that testing a system with Proteus can significantly decrease the amount of irrelevant, false positive, and false negative test cases executed at run time when compared with manually-derived test suites that performed no run-time adaptation. The remainder of this paper is organized as follows. Section II provides background information on the RDM, goal-oriented requirements engineering, and software testing. Section III describes the Proteus approach using the RDM as a motivating example. Section IV then describes our experimental setup and our experimental results. Following, Section V overviews related work. Lastly, Section VI discusses our findings and presents future directions.

II. Background and Application

This section provides relevant background information on the RDM, goal-oriented requirements engineering, and software testing.

A. Remote Data Mirroring

Remote data mirroring (RDM) is a data protection technique for maintaining data availability and preventing data loss by storing copies (i.e., replicates) on servers (i.e., data mirrors) in physically remote locations [17], [18]. By replicating data on remote data mirrors, an RDM network can provide continuous access to data and moreover ensure that data is not lost or damaged. In the event of an error or failure, data recovery can be facilitated by either requesting or reconstructing the lost or damaged data from another active data mirror. Additionally, the RDM network must replicate and distribute data in an efficient manner by minimizing consumed bandwidth and providing assurance that distributed data is not lost or corrupted.

The RDM can reconfigure at run time in response to uncertainty, including dropped or delayed messages and network link failures. Furthermore, each network link incurs an operational cost that directly impacts a controlling budget and also has a measurable throughput, latency, and loss rate. Collectively, these metrics determine the overall performance and reliability of the RDM. To mitigate unforeseen issues, the RDM can reconfigure in terms of its network topology and data mirroring protocols. Specifically, the RDM can selectively activate and deactivate network links to change its overall topology. Furthermore, each data mirror can select a remote data mirroring protocol, defined as either synchronous or asynchronous propagation. Synchronous propagation ensures that the receiving or secondary data mirror both receives and writes incoming data before completion at the primary or sending site. Batched asynchronous propagation collects updates at the primary site that are periodically transmitted to the secondary site. Given its complex and adaptive nature, the RDM application can be modeled and implemented as an SAS [19].

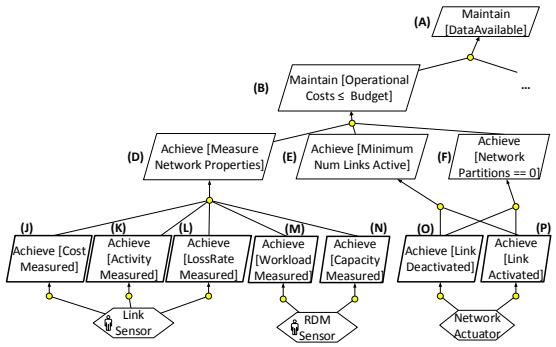
B. Goal-Oriented Requirements Engineering

Goal-oriented requirements engineering (GORE) is an approach that graphically captures high-level objectives and constraints of a system and can be used to guide both the elicitation and analysis of requirements. Different types of goals may be specified in the GORE process. *Functional* goals define a service that the system must provide to its stakeholders. *Non-functional* goals define a quality constraint or criterion upon delivery of those services. *Safety* goals must always be satisfied, as they declare a critical service objective often associated with safety concerns. *Failsafe* goals define a safe fallback in case of critical system failure [20]. Furthermore, functional goals may be classified as invariant (denoted by the keyword "Maintain" or "Avoid") or non-invariant (denoted by the keyword "Achieve"). Invariant goals must always be satisfied by the system, and non-invariant goals may be temporarily unsatisfied at run time.

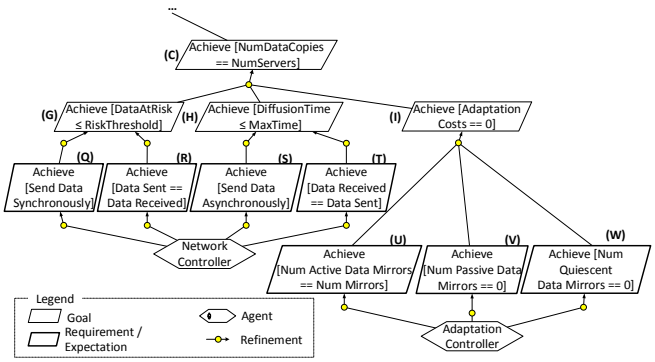
GORE decomposes high-level goals into fine-grained subgoals [21] by using a directed, acyclic graph. Each node in the graph represents a goal and each edge represents a goal refinement. KAOS [21], [22] provides one such approach for systematically refining goals with AND or OR refinements. An AND-refined goal is considered satisfied if all its subgoals have also been satisfied, and an OR-refined goal is satisfied if at least one subgoal has been satisfied. Goal refinement continues until an agent has been assigned responsibility for the satisfaction of each leaf-level goal (i.e., requirement).

Figure 1 provides a KAOS goal model of the RDM application. Specifically, the RDM must maintain remotely stored copies of data (A). To satisfy this goal, the RDM must maintain operational costs within a fixed budget (B) while ensuring that the number of disseminated data copies matches the number of available servers (C). To satisfy Goal (B), the RDM must be able to measure all network properties (D) while ensuring that both the minimum number of network links are active (E) and that the network is unpartitioned (F). To satisfy Goal (C), the RDM must ensure that risk (G) and time for data diffusion (H) each remain within pre-defined constraints, and moreover, the cost of network adaptation must be minimized (I). To satisfy Goals (D) - (I), RDM agents, such as sensors and actuators, must be able to measure and affect all available network properties, respectively.

Utility functions. Utility functions can be used to quantify the degree or level of satisfaction of system requirements or behaviors in autonomic computing systems during execution [23], [24]. Moreover, utility functions can be derived for KAOS goals and can then be used to determine the



(a) Left half of RDM goal model.



(b) Right half of RDM goal model.

Fig. 1. Goal model for RDM application.

run-time satisfaction of each individual goal [25]. Typically, a utility value of 0.0 indicates a requirements violation, a utility value of 1.0 indicates complete satisfaction, and any value in between represents the degree of satisfaction for that particular requirement. For instance, a utility function can measure the satisfaction of Goal (B), "Maintain [Operational Cost $\leq$ Budget]," by monitoring the current cost of operating the network.

Equation (1) presents an example utility function that monitors the current cost of operating the network with respect to the allocated budget:

$$utility_{Goal\_B} = 1.0 - \frac{operational\_cost}{budget} \quad (1)$$

C. Software Testing

Software testing can be used to provide assurance that a system under test is satisfying its requirements. Many approaches exist for testing [26], [27], [28], each with a different focus, such as structural, functional, and regression testing. Structural testing focuses on coverage, such as branch or data flow coverage [29]. Functional testing verifies a system against a test or design specification [29]. Finally, regression testing provides assurance that a system satisfies its requirements following a major change to the system, such as a new software release or change in configuration [30], [31].[2]

[2]The methods of testing implemented within this paper are predominantly functional and regression testing.

Software testing is historically a design-time process. However, testing at run time can provide assurance that a system continually satisfies its requirements during execution [27], [32]. Run-time testing is an approach for performing test cases as the system under test executes. As such, run-time testing can incur a performance cost, as the system must perform testing activities in parallel with normal system execution. Approaches exist for mitigating the performance issue, including multi-agent testing [8] and record-and-replay testing [33]. However, test cases can lose *relevance* during execution as uncertainty manifests within the operating context, necessitating the need for online test case adaptation to ensure that test cases remain applicable to the new operating context [4]. Run-time testing differs from monitoring utility functions in that utility functions measure the satisfaction of key objectives or high-level requirements at run time, whereas testing enables a finer-grained approach for assessing requirements satisfaction as to whether individual features or functions are behaving correctly.

For this paper, we use the definitions of test plans and test cases as defined by the IEEE [29]. In particular, a test case comprises an expected value and the conditions necessary for execution. A test plan describes the scope and schedule of testing activities. An adaptive test plan, therefore, describes how a set of test suites are used to test a particular operating context, where a test suite comprises a collection of test cases to be executed.

III. Approach

Proteus is a requirements-driven approach for managing run-time testing. Specifically, Proteus is a framework for performing testing activities at run time, including test execution and online adaptation. To this end, Proteus provides two levels of test adaptation: test suite adaptation and test case parameter value adaptation. We next describe the Proteus framework and online test adaptation in turn.

A. Proteus Framework

Proteus is a framework for managing run-time testing activities, including adaptation and execution of test suites and test cases. To facilitate this approach, Proteus performs two main tasks. First, an *adaptive test plan* is defined for each SAS configuration at design time, where each SAS configuration corresponds to a particular operating context and each adaptive test plan comprises multiple test suites. Second, Proteus performs a *testing cycle* during the execution of each new SAS configuration, where a testing cycle may comprise multiple iterations that each execute a different test suite. Each task is next described in turn.

Adaptive test plan. An adaptive test plan comprises all possible test suites that can be derived for a particular operating context, or combination of system and environmental parameters. As such, Proteus defines an adaptive test plan for each SAS configuration, where each adaptive test plan comprises a default test suite and a set of intermediate, automatically-derived test suites. We next describe how both

the default test suites are derived at design time and how the intermediate test suites are derived at run time. For illustration purposes, consider that an SAS test engineer is configuring adaptive test plan ATP_i for a particular SAS configuration C_i in response to a particular set of operating conditions OC_i.

At design time, an SAS test engineer defines a *default test suite* $TS_{i.0}$ for an adaptive test plan ATP_i, where a default test suite specifies all test cases that are relevant to the current operating context OC_i and SAS configuration C_i. As such, an activation state is defined for each test case within $TS_{i.0}$. Particularly, a test case can be labeled as:

- **ACTIVE**: An ACTIVE test case to be executed when the current test suite is performed. Moreover, all test cases that monitor *safety*, *failsafe*, or *invariant* conditions are required to always be labeled as ACTIVE to ensure that the SAS continually executes safely and according to its invariant requirements.
- **INACTIVE**: An INACTIVE test case is not executed when the current test suite is performed. A test case is labeled INACTIVE at run time if it passed successfully during the previous testing iteration for the current testing cycle.
- **N/A**: An N/A test case is never executed within the current adaptive test suite, as it is not relevant to the SAS configuration and operating context. N/A test cases cannot become ACTIVE or INACTIVE within the current adaptive test plan.

As such, the SAS engineer designates all relevant test cases as ACTIVE for $TS_{i.0}$ and all test cases that are not relevant as N/A. For instance, a data mirror within the RDM network may be using *asynchronous propagation* to distribute data. In this case, test cases that validate the *synchronous propagation* approach would be designated as N/A as they are not relevant to the current situation. Collectively, the set of test cases with their associated status form a unique test suite within ATP_i.

Proteus dynamically generates test suites at run time to provide *coarse-grained* test adaptation, where a new test suite is generated based upon monitored test results. For example, following execution of $TS_{i.0}$, the test cases designated ACTIVE are executed and their results are analyzed, resulting in a combination of passed and failed test cases. As such, Proteus can derive a new test suite $TS_{i.j}$, based on the default test suite $TS_{i.0}$, to reflect the combination of test results. The application of a test case status (i.e., ACTIVE or INACTIVE) with respect to testing results is discussed in the following section.

Figure 2 illustrates the difference between default test suite $TS_{1.0}$ and dynamically-generated test suite $TS_{1.1}$ for a particular adaptive test plan ATP_1. Specifically, the SAS engineer designated all test cases to be ACTIVE with the exception of TC_3, as it is considered irrelevant and therefore designated N/A. Following execution of $TS_{1.0}$, Proteus determined that TC_4 was satisfied and therefore was designated as INACTIVE, thereby generating $TS_{1.1}$. Upon execution of $TS_{1.1}$, all test cases will be executed with the exception of TC_3 and TC_4. There exists the danger that a fault will occur during the testing cycle and not be caught by a test case because it was previously designated as INACTIVE. However, if such a fault occurs, the SAS monitoring infrastructure should trigger a reconfiguration, thereby starting a new testing cycle in which the INACTIVE test case is reactivated. The run-time testing cycle is next described in detail.

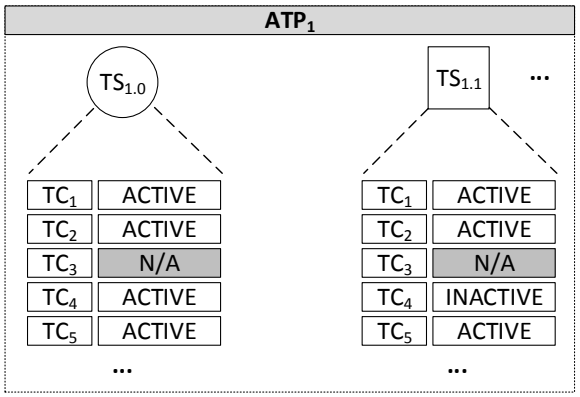


Fig. 2. Example of test case configuration for $TS_{1.0}$ and $TS_{1.1}$.

Testing cycle. Proteus performs a *testing cycle* at each step of SAS execution, where a testing cycle comprises several iterative steps. For illustration purposes, consider that an SAS has invoked configuration C_i in response to operating context OC_i, thereby activating adaptive test plan ATP_i. The steps that define a testing cycle are next described in turn:

(1) Execution of default test suite. A testing cycle begins upon invocation of SAS configuration C_i. Proteus requires that the default test suite $TS_{i.0}$ always be executed at the beginning of a testing cycle for two reasons. First, $TS_{i.0}$ provides an overview of the current state of C_i and OC_i, as minor or transient changes to OC_i may cause test cases to react differently. For example, a minor or transient change to OC_i may not necessitate an SAS reconfiguration, however it is detectable at the testing level, thereby providing a finer-grained view of SAS behavior at run time than can be provided by higher-level utility functions. Second, $TS_{i.0}$ provides regression analysis of SAS configuration C_i in response to operating context OC_i. For instance, a test case that passed in a previous testing cycle may fail in the current testing cycle, indicating an error in C_i in response to minor changes to OC_i. To perform the default test suite, Proteus executes each test case that has been labeled as ACTIVE.

(2) Analysis of test results. Next, Proteus analyzes the results of each executed test case. To provide a quantitative measure of testing and to enable online test adaptation, we define *test case relevance* to measure how relevant a test case is to its current operating context. A test case may lose relevance if minor, yet tolerable, changes to the operating context cause the parameter values for existing test cases to no longer be valid. A typical relevance calculation is presented in Equation 2:

$$relevance_{TC_i} = 1.0 - \frac{|value_{measured} - value_{expected}|}{|value_{expected} + value_{variance}|} \quad (2)$$

where a value of 0.0 indicates no relevance and 1.0 indicates full relevance. Moreover, $value_{measured}$ is the calculated value of the test case, $value_{expected}$ is the expected value of the test case, and $value_{variance}$ is defined by the SAS test engineer as the maximum value that $value_{measured}$ may take to ensure that $relevance_{TC_i}$ is normalized on $[0.0, 1.0]$. Furthermore, a test result may also be designated as having passed or failed, based on a threshold for test case relevance defined by the SAS test engineer. A higher threshold will result in more failed test cases and incur more test adaptations while a lower threshold will yield more passed test cases and a lower amount of test adaptations. For the purposes of this work, we selected a relevance threshold of 0.75 to trigger the test adaptation process more often throughout SAS execution to measure the effects of Proteus.

Test case relevance must also be validated following testing adaptation. To this end, Proteus uses the utility functions defined for the system goal model (c.f., Section II-B) to ensure that test cases have not been adapted incorrectly. Each test case must be correlated to at least one utility function for comparison. Proteus uses utility values for two purposes. First, a utility value may indicate that a particular goal is unsatisfied and therefore the SAS may require reconfiguration. Second, the utility values act as a point of reference to which test case results are validated at run time. For reference, a utility function is considered violated if the calculated utility value is 0.0 and otherwise is considered satisfied to some degree.

The relevance value of each test case is then compared to its correlated utility function(s) to determine if the test result is a:

- **True positive**: Test case relevance is within $[Threshold, 1.0]$ and its correlated utility value is within $(0.0, 1.0]$, indicating that the test is valid and has passed. No extra action is required by the SAS reconfiguration engine or the Proteus test adaptation framework.
- **True negative**: Test case relevance is within $[0.0, Threshold)$ and its correlated utility value equals 0.0, indicating that an error has occurred and the test has failed. Presence of a true negative implies that the SAS requires reconfiguration, halting further testing until the issue has been resolved.
- **False positive**: Test case relevance is within $[Threshold, 1.0]$ and its correlated utility value equals 0.0, requiring both SAS reconfiguration and test adaptation. This result indicates that the test case has passed, however its correlated goal is not satisfied. In this case, the test case and utility value are in *disagreement*, indicating that the test case and/or the utility function are incorrect. Test cases that are false positive require adaptation to become relevant. As the SAS will perform an immediate reconfiguration as a result of the violated utility function, validation of an adapted false positive will not occur until a future testing cycle when the test case in question is ACTIVE. Moreover, an SAS test engineer can analyze the false positive and associated execution trace to determine the source of the error, particularly if the test case or utility function is in error.[3]
- **False negative**: Test case relevance is within $[0.0, Threshold)$ and its correlated utility value is within $(0.0, 1.0]$, indicating that the test case requires adaptation. In this case, the test case and utility value are again in *disagreement*. Given that the utility function is satisfied, the test case requires adaptation to become relevant again. Moreover, an SAS test engineer can analyze the state of the SAS and its environment at this point in time to determine the reason(s) that the test case became irrelevant.[3]

(3) Fine-grained test case parameter value adaptation. Following the analysis of test results, Proteus determines if fine-grained adaptation is necessary to realign test case parameter values with OC_i. To this end, test cases that have resulted in a *false positive* or *false negative* are selected for fine-grained adaptation. For each false negative or false positive test case, Proteus invokes Veritas [4], an online, evolutionary computation-based approach for optimizing test case parameters at run time, to explore the search space of possible test case parameter values that better match current operating conditions. Veritas yields an optimized expected value for each provided test case that better reflects the environment. Moreover, the optimized test case is used in all existing test suites $TS_{i.j}$, including the default test suite $TS_{i.0}$, within ATP_i.

However, test cases that monitor *safety*, *failsafe*, or *invariant* requirements or conditions (hereafter termed "invariant test cases") are precluded from adaptation to ensure that all safety and invariant concerns are continuously satisfied. In the event that an invariant test case results in a false positive or false negative, an SAS engineer is notified to localize the cause of the error, as such an error would be considered catastrophic.

(4) Coarse-grained test plan adaptation. Next, Proteus performs coarse-grained adaptation based upon test results. Coarse-grained adaptation is used to reduce the amount of test cases that are unnecessarily executed at run time. For instance, as testing is performed at each step of SAS execution, measures are required to reduce the overall impact of testing. To this end, Proteus now labels test cases as ACTIVE or INACTIVE based on the results of Step (3):

- *Invariant* test cases are always labeled as ACTIVE to ensure that they are continually re-validated and precluded from fine-grained adaptation.
- *True positive* test cases are labeled as INACTIVE, as they do not need to be re-validated during the current testing cycle.

[3]For this work, we assume that all utility functions have been derived correctly, and therefore a false positive indicates an error in a test case parameter value.

- *False positive* and *false negative* test cases are labeled as ACTIVE, as they require additional validation following adaptation.

The combination of test case statuses forms a new test suite $TS_{i.j}$. If another iteration of the testing cycle is required (See Step (5) as follows), then the process iterates back to Step (1), where $TS_{i.j}$ replaces $TS_{i.0}$ for execution.

(5) End of testing cycle. A testing cycle terminates for two reasons. First, a new SAS configuration is invoked based upon a major change to the current operating context (e.g., SAS configuration C_k is invoked based upon identification of operating context OC_k, at which point ATP_k is activated). Second, all test cases have resulted in *true positives*, thereby resulting in each test case having a status of either INACTIVE or N/A. In this case, run-time testing is halted until a change in operating context is detected by the SAS.

B. Proteus Overview

Figure 3 presents a graphical overview of an SAS within the Proteus framework. Specifically, this figure presents the logical connections between SAS configurations, operating contexts, adaptive test plans, and the test suites contained within the adaptive test plans. For example, SAS configuration C_1 is triggered by operational context OC_1. When testing begins, Proteus selects the associated adaptive test plan ATP_1. ATP_1 comprises a collection of test suites $TS_{1.j}$ that each define a particular configuration of test cases to be executed. Moreover, $TS_{1.0}$ is considered to be the default test suite for ATP_1, and as such, is executed initially each time ATP_1 is selected for testing. The test case parameter values for all test cases within ATP_1 are stored in its associated data store $Params_1$. Finally, each derived test suite $TS_{1.j}$ comprises a collection of test cases that each specify an activation state. Specifically, Proteus selectively activates or deactivates test cases based on test results, and therefore the configuration of test case states (i.e., ACTIVE, INACTIVE, or N/A) define a unique test suite.

Figure 3 also demonstrates how different testing cycles derive new test suites at run time. Specifically, the results from two iterations through a testing cycle are shown in Figure 3 by different patterns in each $TS_{i.j}$. The first iteration is represented by the darker, right-slanted shading, and the second iteration is represented by the lighter, left-slanted shading. $TS_{1.0}$, as the default test suite, was executed at the start of both testing cycles for regression purposes. Following execution of the default test suite, both testing cycles generated $TS_{1.1}$ based on the configuration of test cases. Following execution of $TS_{1.1}$, the first testing cycle then generated $TS_{1.2}$, $TS_{1.3}$, and $TS_{1.4}$, whereas the second testing cycle generated $TS_{1.5}$, $TS_{1.6}$, and $TS_{1.7}$.

Regression Testing. To continually ensure that the SAS is satisfying its requirements even as the environment changes, Proteus performs different levels of regression testing. For instance, test cases that monitor invariant, safety, or failsafe conditions (i.e., Figure 1, Goals (A) and (B)) must always remain ACTIVE to ensure that they are re-validated when each test suite is executed. Test cases that monitor non-invariant goals must also be re-validated in the event of a contextual change or SAS self-reconfiguration. For instance, following reconfiguration of the SAS from C_1 to C_2, the default test suite $TS_{i.0}$ is executed with all relevant test cases set to ACTIVE. Assuming that a particular test case has been previously validated in a prior testing cycle, re-executing that test case in the current state provides functional regression assurance.

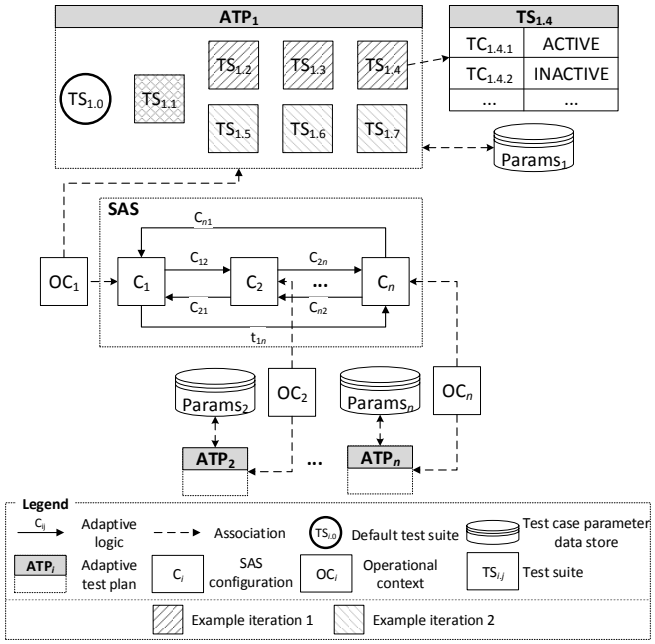


Fig. 3. Example of SAS within Proteus framework.

IV. Experimental Results

This section presents our experimental setup and discusses the results obtained by applying Proteus to the RDM. We also present preliminary results from studying the impact of our run-time testing framework on the RDM.

A. Experimental Setup

For this paper, the RDM application has been implemented as a completely connected graph, where each node represents an RDM and each edge represents a network link. For each experimental treatment, the RDM network comprised between 15 and 30 data mirrors and was required to disseminate between 100 and 200 messages over 300 timesteps.

Uncertainty was simulated within both the environment and within the RDM application itself. In particular, the RDM can experience unpredictable network link failures, randomly dropped or delayed messages, and noise within both data mirror sensors and network links at each timestep of simulation. The RDM network could then self-reconfigure in response to these adverse conditions to continue execution. Possible reconfigurations include updates to the network topology or data mirror propagation parameters.

The input test specification comprised 36 test cases, where 7 test cases were considered invariant and therefore pre-

cluded from adaptation. Moreover, invariant test cases were re-executed each testing cycle to ensure that system constraints are continually satisfied. The remaining 29 test cases were considered non-invariant and were therefore targets for adaptation. The RDM simulation was also instrumented to enable run-time monitoring of system and environmental conditions that are not typically available to RDM sensors. Examples include monitored variables that store data regarding the RDM decision-making logic and data structures that maintain the state of all objects within the simulation environment over time.

We compared and evaluated adaptive test plans generated by Proteus for each SAS configuration with a manually-derived test plan (hereafter the "Control") that did not provide run-time adaptation capabilities. The Control test plan contains a single test suite that comprises all test cases from the input test specification and only executes the test cases that satisfy their execution requirements (i.e., conditions necessary for execution were met). According to the IEEE standard, a test case must define the conditions that must be true for the test to be successfully executed [29]. The intent of the Control test plan is to provide coverage of all possible reconfigurations that may be performed at run time by the RDM. Moreover, Proteus invoked Veritas as necessary for fine-grained test case parameter adaptation. For statistical purposes, we conducted 50 experimental treatments, and, where applicable, plotted mean values with corresponding error bars or deviations. Moreover, we use the Wilcoxon-Mann-Whitney U-test to determine if statistical significance exists between two data samples, as we do not assume normality of data.

B. Run-Time Test Plan Adaptation

For this experiment, we define the null hypothesis H_0 to state that "*there is no difference between a Proteus adaptive test plan and a manually-derived test plan*." Moreover, we define the alternate hypothesis H_1 to state that "*there is a difference between a Proteus adaptive test plan and a manually-derived test plan*."

Figure 4 presents boxplots of the number of test cases that should not have been executed (hereafter termed "irrelevant") between a Proteus adaptive test plan and a manually-derived test plan. A test case is considered to be irrelevant if its relevance value (see Equation 2) equals 0.0. For those test cases designated as irrelevant, the difference between the measured and expected values is large, indicating that the test case is no longer relevant to its operating context. Figure 4 shows that Proteus significantly reduces the amount of irrelevant test cases executed in comparison to those executed under a manually-derived test plan ($p < 0.05$).

Testing activities were further analyzed to monitor the amount of *false positive* test cases, or instances where test case relevance falls within $[Threshold, 1.0]$ and its correlated utility value equals 0.0 (see Section III-A). Figure 5 illustrates how Proteus significantly reduces the amount of false positive test results as compared to testing with the manually-derived test plan ($p < 0.05$). These results indicate that testing with adaptive test plans can reduce the amount of false positive test results, thus reducing the need for spurious test adaptation and, moreover, reducing the burden of unnecessary analysis by the SAS test engineer.

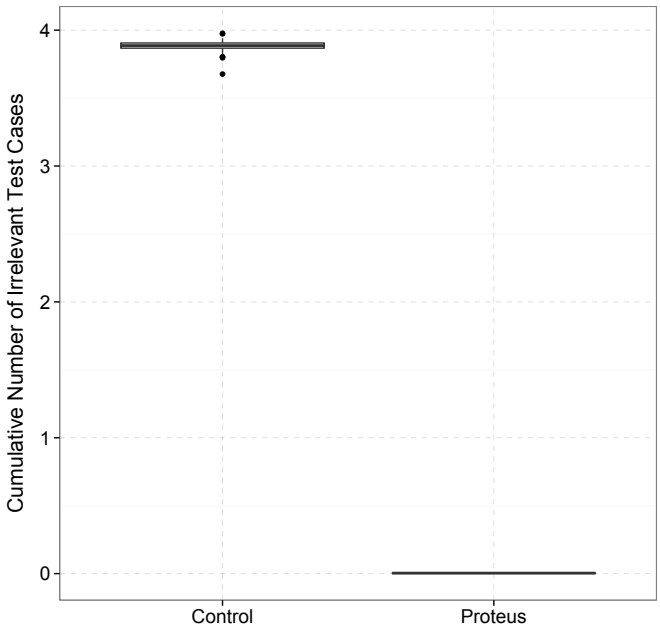


Fig. 4. Cumulative number of irrelevant test cases executed for each experiment.

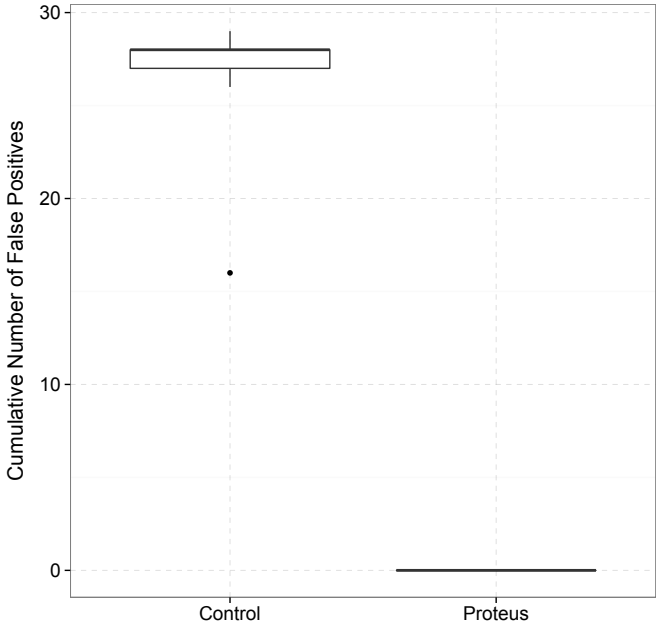


Fig. 5. Cumulative number of false positive test cases for each experiment.

Figure 6 shows how Proteus also significantly reduces the amount of *false negative* results, or instances when test case relevance is calculated to be within $[0.0, Threshold)$ while its correlated utility value is greater than 0.0 (see Section III-A), that were encountered during testing ($p < 0.05$). This result indicates that Proteus adaptive test plans assist in reducing the amount of run-time testing adaptations required for the

testing framework, thereby reducing the overall cost of testing the SAS and the amount of analysis required by an SAS test engineer.

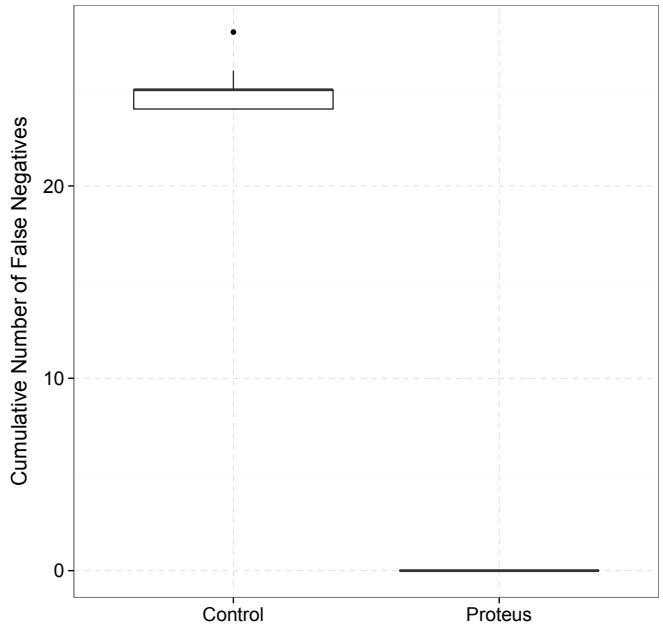


Fig. 6. Cumulative number of false negative test cases for each experiment.

Lastly, the total number of executed test cases were recorded to provide a measure of the overall impact of run-time testing to an SAS. Particularly, we demonstrate that Proteus significantly reduces the amount of executed test cases per testing cycle by ensuring that only relevant test cases are executed, as shown in Figure 7. This figure illustrates that Proteus can reduce the amount of required effort by a testing framework at run time ($p < 0.05$).

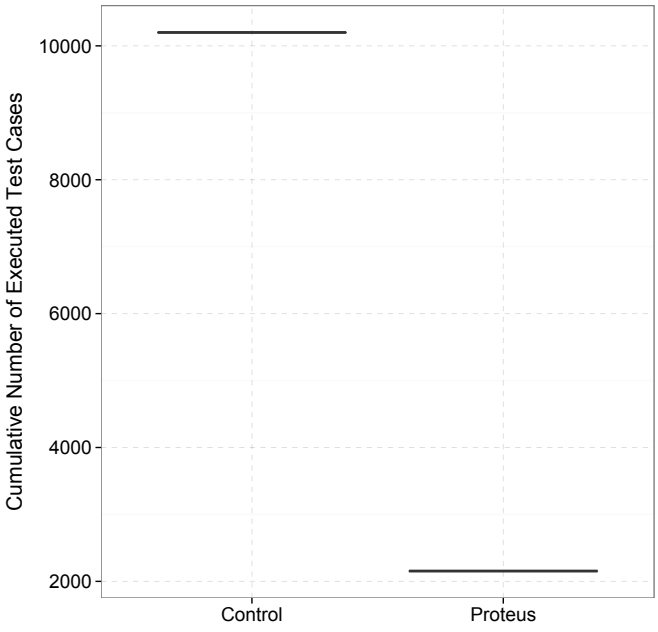


Fig. 7. Cumulative number of executed test cases for each experiment.

The results presented in Figures 4 – 7 enable us to reject the null hypothesis H_0 and determine that a clear difference exists between Proteus adaptive test plans and a manually-derived test plan. Moreover, these results enable us to accept the alternate hypothesis H_1, based on the assumption that the manually-derived test plan was based on operating conditions that the RDM generally experiences. Therefore, these results suggest that Proteus adaptive test plans can provide assurance that test cases remain relevant to changing operating conditions due to uncertainty.

Threats to validity. The research presented in this paper is intended as a proof of concept to determine the feasibility of using Proteus for managing and adapting run-time testing activities. One threat to validity is if Proteus will achieve similar results in a different problem domain involving self-reconfigurations. Another threat to validity occurs in the validity of the input test specification, as it must be fully comprehensive to enable complete coverage of the system requirements for testing assurance to be provided. Finally, another threat to validity is that the experimental results presented within this paper focus on increasing testing relevance rather than focusing on test case failures to uncover faults in the system.

C. Impact of Run-Time Testing

Run-time testing provides a valuable layer of assurance for an SAS. However, the relative impact that a testing framework imparts upon the SAS must also be considered when performing run-time testing. While research has been performed with providing assurance at run time beyond or complementary to testing techniques [34], [35], [36], [37], [38], the direct impact that testing at run time has on an SAS has yet to be explored. The addition of testing activities can require additional processing time, extra memory overhead, and unexpected changes to SAS behavior. To this end, we analyzed how our run-time testing framework impacts an SAS at run time relative for the following conditions:

- **(S1)**: All run-time testing activities enabled (i.e., Proteus and Veritas enabled)
- **(S2)**: Run-time testing disabled (i.e., Proteus and Veritas disabled)
- **(S3)**: Run-time testing removed (i.e., Proteus and Veritas data structures and functions removed from SAS codebase)

(S1) executes the RDM with all run-time testing activities enabled, including test case execution, Proteus adaptive test plan adaptation, and Veritas test case parameter value adaptation. (S2) does not perform run-time testing, however the data structures and functions required by the framework are still instantiated by the SAS. Lastly, (S3) completely removes the run-time testing framework from the SAS.

SAS performance can be quantified based upon two key metrics: *total execution time* and *memory footprint*. We next provide the method for which we measure each metric, as well as the observed result, where the presented result is the mean of 50 experimental treatments.

Total execution time: In simulation, the SAS executes for a set amount of timesteps and therefore the total execution time can be measured. To this end, we measure the total execution time of the function that is responsible for executing the complete SAS simulation. Particularly, we use the `cProfile` Python package to measure the *cumulative* time that the simulation execution function requires. Measuring the execution times of a deterministic SAS instrumented with run-time testing and with testing disabled can then provide a point of comparison for any extra time used to perform run-time testing.

(S1) required a mean execution time of 23.03 seconds, (S2) required 13.901 seconds, and (S3) required 13.785 seconds. A significant different exists between (S1) and both (S2) and (S3), indicating that performing run-time testing requires significantly more time for the simulation to complete than either disabling or removing run-time testing from the RDM application ($p < 0.05$).

Memory footprint: Extra memory may be consumed when instrumenting an SAS with a run-time testing framework. Depending on the hardware used to support the SAS, the extra memory cost may be prohibitive, particularly in embedded systems where memory is limited. Given that the RDM application has been implemented in Python, we use the `resource` package to examine the total amount of memory consumed throughout execution.

With respect to memory footprint, (S1) required 65.324 mb, (S2) required 65.332 mb, and (S3) required 65.020 mb. As such, no significant difference in memory overhead was incurred between each testing state ($p > 0.05$).

While execution time and incurred memory costs are relatively straightforward to quantify, examining differences in SAS behavior is less obvious. To this end, we define two key metrics for quantifying behavior: *requirements satisficement* and *behavioral function calls*. Each of these metrics are next described in turn.

Requirements satisficement: We first examine the extent to which software requirements are satisficed during execution to examine the impact that testing may impose upon an SAS. Particularly, we monitor utility values calculated based upon the RDM goal model (c.f., Figure 1) to determine if any difference occurs when run-time testing is performed on an SAS. Given that a utility value is required to be within a range of $[0.0, 1.0]$, any difference in behavior can be identified based on a comparison of the utility values that quantify goal satisficement.

Here, we average the calculated utility values across all timesteps of RDM execution into a single value to represent overall goal satisficement. On average, (S1) yielded a mean utility value of 0.7717, (S2) yielded 0.7656, and (S3) yielded 0.7656. These results, while exhibiting a difference in their respective means, are not statistically significantly different from each other overall ($p > 0.05$), resulting in the conclusion that our testing framework does not significantly impact SAS behavior.

Next, we present the average number of utility violations throughout RDM execution. (S1) incurred 830.6 violations, (S2) incurred 875.0 violations, and (S3) incurred 869.7 violations. Statistically, performing run-time testing does not incur significantly more utility violations as compared to disabling or removing run-time testing ($p > 0.05$).

While the presented results pertaining to requirements satisficement are not statistically significant, a clear difference exists in the presented mean values. As such, we examined RDM execution traces to determine why this difference occurs. To this end, we found that, while no difference exists in the operating context between all three testing states, a minor difference in execution behavior does occur. Specifically, utility functions sample data provided by the RDM each timestep. When testing is enabled or available, the real time at which utility values are calculated will be slightly delayed by the extra time required to perform testing activities, resulting in a sampling of different RDM environmental states. As such, a difference in calculated utility values can occur as the operating context is slightly different.

Behavioral function calls: We also quantify behavioral performance based upon the number of behavioral function calls invoked. We define a behavioral function call as a function identified by the SAS engineer as having an integral impact on SAS behavior. For the purposes of this research, we monitor the number of self-reconfigurations performed by the SAS, as a self-reconfiguration can create a major divergence in system behavior.

(S1) performed 23.0 reconfigurations, (S2) performed 17.28 reconfigurations, and (S3) performed 19.16 reconfigurations. Again, run-time testing does not incur significantly more adaptations than were found by disabling or removing testing ($p > 0.05$). These results, coupled with the results calculated based on utility values and utility violations, indicate that our testing framework does not introduce significant behavioral change to the RDM application.

The experimental results presented in this section suggest that run-time testing only significantly impacts an SAS in the amount of execution time required, whereas memory overhead and behavior are not significantly impacted by run-time testing. We further analyzed our run-time testing framework and determined that the extra execution time is a largely a result of overhead introduced by our testing framework rather than run-time test execution and adaptation. As such, we are exploring optimization strategies, such as parallelization of testing activities, alongside this research.

V. Related Work

This section describes related work in run-time testing, search-based software testing, test plan generation, and test case selection.

A. Run-Time Testing

While testing software at design time provides assurance that a system satisfies its requirements specification, testing at run time can provide assurance that the system continually

satisfies requirements in unexpected situations. Run-time testing has been successfully implemented by recording traces of the production system to a secondary computer. The traces are then examined in parallel for faults [33]. Agent-based approaches have also been leveraged as a means for testing a system at run time [8]. Run-time testing has been proposed as a proactive approach to facilitate run-time adaptation of service-based systems [39]. Moreover, a roadmap of traits and properties that are desirable for testing adaptive systems at run time has been previously enumerated [40]. While each of these approaches implement or propose some form of run-time testing, Proteus focuses on maintaining test plan and test case relevance at run time as system and environmental conditions change in order to minimize the number of executed irrelevant test cases for SASs.

B. Search-Based Software Testing

Search-based software testing applies search-based techniques, such as simulated annealing and evolutionary computation, to the field of software testing. As search-based techniques explore the available solution space, they are a natural fit in areas such as automated test case generation, with examples including application to model-based testing, mutation testing, regression testing, and structural testing [28], [41], [42]. Veritas leverages a run-time evolutionary algorithm to provide online evolution and search-based software testing capabilities [4], whereas the previous approaches search for solutions at design time. Online evolution provides the distinct advantage of facilitating adaptation under live conditions throughout execution, rather than requiring offline bug fixes or code optimizations to mitigate unforeseen circumstances.

C. Test Plan Generation

Automated generation of test plans has been extensively studied. One such approach uses requirements and formal grammars to automatically define a test plan [15]. In this approach, a requirements specification is converted to a finite state automata (FSA), a grammar is derived from the FSA, and then a set of test plans are generated from the grammar. The test plans are then executed throughout the software testing cycle. In contrast, Proteus-generated test plans can be generated at run time. Automated planning has also been applied to graphical user interface (GUI) testing [43]. In this approach, artificial intelligence techniques are used to anticipate actions taken by the users of a GUI, resulting in a set of testing goals. Upon completion, the controlling algorithm generates a partially-ordered plan for realizing the testing goals, and then generates a set of related test cases. In comparison, our approach automatically generates a test plan at run time by leveraging SAS configuration information to determine which test cases are relevant to each operating context, thereby bypassing expensive decision or learning mechanisms.

D. Test Case Selection

Automated techniques for selecting and prioritizing test cases have been previously surveyed within the search-based software engineering domain [28], where selection is concerned with selecting a representative set of test cases, and prioritization involves optimizing the order of test case execution. While the surveyed techniques tend to focus on design-time optimizations, Proteus instead selects test cases at run time. Tropos [8] is an implementation of a multi-agent system that provides a testing agent to continuously execute test cases. In this approach, test cases are randomly selected based upon current operating conditions. Conversely, Proteus generates test plans targeted towards each SAS configuration, rather than relying on random test case selection.

VI. Conclusion

In this paper, we have presented Proteus, a framework for adaptive run-time testing. Particularly, Proteus manages test suite and test case adaptation. Test suite adaptation is enabled through the use of adaptive test plans, where an adaptive test plan comprises a default test suite and automatically generated suites derived from the default test suite. Proteus selects a particular test suite based upon identified operating conditions for the execution of run-time test cases. Moreover, Proteus manages test case adaptation through invocation of Veritas, a run-time evolutionary approach for evolving test case parameter values. Both types of adaptation are used to ensure that both test plans and test cases remain *relevant* to their current operating context. We have demonstrated Proteus on an RDM application that was required to replicate data across a network. The RDM network was subjected to uncertainty in the form of random network link failures, data mirror failures, and dropped or delayed messages. Experimental results demonstrate that adaptive test plans provide a higher degree of relevance than a manually-derived test plan. Future work includes application of Proteus to other SAS application domains, investigation into using search-based techniques to derive a collection of adaptive test plans at design time, and automatically defining default test suites based on SAS requirements specifications.

Acknowledgment

This work has been supported in part by NSF grants CCF-0820220, DBI-0939454, CNS-0854931, CNS-1305358, Ford Motor Company, and General Motors. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Ford, General Motors, or other research sponsors.

References

[1] B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, and et al., "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Berlin, Heidelberg: Springer-Verlag, 2009, ch. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26.

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

[3] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer*, vol. 37, no. 7, pp. 56 – 64, july 2004.

[4] E. M. Fredericks, B. DeVries, and B. H. C. Cheng, "Towards run-time adaptation of test cases for self-adaptive systems in the face of uncertainty," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '14, 2014.

[5] B. H. C. Cheng, P. Sawyer, N. Bencomo, and J. Whittle, "A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty," in *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 468–483.

[6] N. Esfahani, E. Kouroshfar, and S. Malek, "Taming uncertainty in self-adaptive software," in *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 234–244. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025147

[7] C. Ghezzi, "Adaptive software needs continuous verification," in *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*, Sept. 2010, pp. 3 –4.

[8] C. D. Nguyen, A. Perini, P. Tonella, and F. B. Kessler, "Automated continuous testing of multiagent systems," in *The Fifth European Workshop on Multi-Agent Systems (EUMAS)*, 2007.

[9] N. Qureshi, S. Liaskos, and A. Perini, "Reasoning about adaptive requirements for self-adaptive systems at runtime," in *Proc. of the 2011 International Workshop on Requirements at Run Time*, aug. 2011, pp. 16 –22.

[10] J. Camara and R. de Lemos, "Evaluation of resilience in self-adaptive systems using probabilistic model-checking," in *Software Engineering for Adaptive and Self-Managing Systems.*, june 2012, pp. 53 –62.

[11] Y. Falcone, M. Jaber, T.-H. Nguyen, M. Bozga, and S. Bensalem, "Runtime verification of component-based systems," in *Proc. of the 9th international conference on Software engineering and formal methods*. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 204–220.

[12] A. Filieri, C. Ghezzi, and G. Tamburrelli, "A formal approach to adaptive software: continuous assurance of non-functional requirements," *Formal Aspects of Computing*, vol. 24, pp. 163–186, 2012.

[13] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng, "Towards run-time testing of dynamic adaptive systems," in *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '13. IEEE Press, 2013, pp. 169–174.

[14] P. Ammann and J. Offutt, "Introduction to software testing," *Cambridge University Press*, 2008.

[15] J. A. Bauer and A. B. Finger, "Test plan generation using formal grammars," *Proceedings of the 4th International Conference on Software Engineering*, pp. 425–432, 1979.

[16] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton, "The combinatorial design approach to automatic test generation," *IEEE Software*, vol. 13, no. 5, pp. 83–88, 1996.

[17] M. Ji, A. Veitch, and J. Wilkes, "Seneca: Remote mirroring done write," in *USENIX 2003 Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, June 2003, pp. 253–268.

[18] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes, "Designing for disasters," in *Proc. of the 3rd USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2004, pp. 59–62.

[19] A. J. Ramirez, D. B. Knoester, B. H. Cheng, and P. K. McKinley, "Applying genetic algorithms to decision making in autonomic computing systems," in *Proceedings of the 6th international conference on Autonomic computing*, 2009, pp. 97–106.

[20] ISO, "Iso 26262: Road vehicles – functional safety," *International Standard ISO/FDIS 26262*, 2011.

[21] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[22] A. Dardenne, A. Van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of computer programming*, vol. 20, no. 1, pp. 3–50, 1993.

[23] P. deGrandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *Proc. of the 6th International Conference on Autonomic Computing*, ser. ICAC '09. ACM, 2009, pp. 107–116.

[24] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Proc. of the First IEEE International Conference on Autonomic Computing*. IEEE Computer Society, 2004, pp. 70–77.

[25] A. J. Ramirez and B. H. C. Cheng, "Automatically deriving utility functions for monitoring software requirements," in *Proc. of the 2011 International Conference on Model Driven Engineering Languages and Systems Conference*, Wellington, New Zealand, 2011, pp. 501–516.

[26] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[27] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *Future of Software Engineering, 2007. FOSE '07*, 2007, pp. 85–103.

[28] M. Harman, S. A. Mansouri, and Y. Zhang, "Search based software engineering: A comprehensive analysis and review of trends techniques and applications," *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[29] IEEE, "Systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, Dec 2010.

[30] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *Software Engineering, IEEE Transactions on*, vol. 22, no. 8, pp. 529–551, 1996.

[31] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. The Eighth International Symposium On Software Reliability Engineering*. IEEE, 1997, pp. 264–274.

[32] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[33] J.-P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, "A noninterference monitoring and replay mechanism for real-time software testing and debugging," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 897–916, 1990.

[34] A. Mukherjee and D. Siewiorek, "Measuring software dependability by robustness benchmarking," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 366–378, 1997.

[35] R. Almeida and M. Vieira, "Benchmarking the resilience of self-adaptive software systems: Perspectives and challenges," in *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011, p. 6.

[36] J. Cámara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Testing the robustness of controllers for self-adaptive systems," *Journal of the Brazilian Computer Society*, vol. 20, no. 1, 2014.

[37] B. H. C. Cheng, K. I. Eder, M. Gogolla, L. Grunske, M. Litoiu, H. A. Müller, P. Pelliccione, A. Perini, N. A. Qureshi, B. Rumpe, D. Schneider, F. Trollmann, and N. M. Villegas, "Using models at runtime to address assurance for self-adaptive systems," in *Models@run.time*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8378, pp. 101–136.

[38] M. U. Iftikhar and D. Weyns, "Activforms: Active formal models for self-adaptation," in *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2014, pp. 125–134.

[39] J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore, "A framework for proactive self-adaptation of service-based applications based on online testing," in *Towards a Service-Based Internet*, ser. Lecture Notes in Computer Science, P. Mähönen, K. Pohl, and T. Priol, Eds. Springer Berlin Heidelberg, 2008, vol. 5377, pp. 122–133. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89897-9_11

[40] R. de Lemos, H. Giese, H. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. Smith, J. Sousa, L. Tahvildari, K. Wong, and J. Wuttke, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*, ser. Lecture Notes in Computer Science, R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, Eds. Springer Berlin Heidelberg, 2013, vol. 7475, pp. 1–32.

[41] M. Harman, P. McMinn, J. T. Souza, and S. Yoo, "Search based software engineering: Techniques, taxonomy, tutorial," in *Empirical Software Engineering and Verification*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7007, pp. 1–59.

[42] P. McMinn, "Search-based software testing: Past, present and future," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 153–163.

[43] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Hierarchical gui test case generation using automated planning," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 144–155, 2001.