Towards Run-Time Adaptation of Test Cases for Self-Adaptive Systems in the Face of Uncertainty

Erik M. Fredericks
freder99@cse.msu.edu

Byron DeVries
devri117@cse.msu.edu

Betty H. C. Cheng
chengb@cse.msu.edu

Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, 48823, USA

ABSTRACT

Self-adaptive systems (SAS) may be subjected to conditions for which they were not explicitly designed. For those high-assurance SAS applications that must deliver critical services, techniques are needed to ensure that only acceptable behavior is provided. While testing an SAS at design time can validate its expected behaviors in known circumstances, testing at run time provides assurance that the SAS will continue to behave as expected in uncertain situations. This paper introduces Veritas, an approach for using utility functions to guide the test adaptation process as part of a run-time testing framework. Specifically, Veritas adapts test cases for an SAS at run time to ensure that the SAS continues to execute in a safe and correct manner when adapting to handle changing environmental conditions.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements / Specifications—*tools*; D.2.4 [**Software Engineering**]: Software / Program Verification—*reliability, validation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*

General Terms

Measurement, performance, reliability, verification

Keywords

Search-based software engineering, software assurance, evolutionary algorithms, software testing

1. INTRODUCTION

In order to handle unanticipated changing system and environmental conditions [7], a self-adaptive system (SAS) adapts at run time to new configurations [6, 19, 21]. Testing the SAS throughout the different phases of software design and development provides assurance that the system will execute as specified in the software design and requirements specifications. System and environmental conditions may change throughout system execution beyond what was known at design time, thereby potentially making previously specified test cases irrelevant to the current operating context. System and environmental uncertainty manifests itself further in partially informed decisions and assumptions regarding the SAS's requirements, as the execution environment may not have been fully understood during requirements elicitation and system development [34]. Furthermore, the reconfiguration capabilities of an SAS can cause previously derived test cases to no longer be applicable to the current execution environment. This paper presents Veritas, a run-time evolutionary approach for adapting an online testing framework for an SAS to ensure at run time that the test cases are relevant to the current operating conditions.

Typically, it is difficult to anticipate all possible operating conditions that an SAS may face throughout execution [6, 7, 21]. Existing design-time [5, 12, 25, 27] and run-time [15, 16, 25, 26] techniques have been developed for testing an SAS. These techniques are typically restricted to evaluating requirements satisfaction given specific operational contexts comprising system and environmental configurations and may not consider unanticipated changes within the system or environment. Search-based heuristics, including evolutionary algorithms, have been previously used for generating test cases and test suites [1, 13, 20]. However, techniques are needed to ensure that an SAS continually delivers acceptable behavior throughout execution even as its operating context changes.

This paper introduces Veritas,[1] an approach that uses evolutionary computation to adapt requirements-based test cases at run time to handle different system and environmental conditions. Veritas monitors its environment for evidence of change and then automatically adapts individual test cases as necessary to ensure testing relevance, while protecting against adapting test cases to automatically pass under invalid conditions. Test adaptation can be triggered by significant environmental changes, system reconfiguration, or invalid test cases.

Veritas uses utility functions [33] that measure how well high-level system goals are being satisfied to assess relevance

[1] In Roman mythology, Veritas was known as the goddess of truth.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
SEAMS '14, June 2-3, 2014, Hyderabad, India
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2864-7/14/05$15.00.
http://dx.doi.org/10.1145/2593929.2593937.

of test cases. While utility functions can quantify high-level intent of a system (e.g., maintain a safe distance between vehicles), test cases can provide finer-grained assessments as to whether individual features and functions are behaving properly (e.g., radar is detecting obstacles within a specific range). Veritas also provides regression analysis [28, 35], where existing test cases are re-evaluated in the event of operational context change (e.g., significant environmental conditions change and/or SAS reconfiguration). By correlating utility values with test cases, Veritas can identify test cases that are valid and/or invalid at run time, and if invalid, can provide a frame of reference for adapting the test case parameters to be realigned with the current operational context. Once Veritas determines that a test case requires adaptation, an online evolutionary algorithm is executed to facilitate run-time adaptation of test case parameters.

We demonstrate Veritas by applying it to an intelligent robotic vacuum that is tasked with safely and efficiently cleaning a given environment. Experimental results indicate that Veritas can significantly reduce the amount of test case failures due to the mismatch between test cases and operational context caused by uncertainty within the system and environment. The remainder of this paper is organized as follows. Section 2 provides background information on the smart vacuum system (SVS) application, goal modeling, genetic algorithms, and software testing. Section 3 describes the Veritas approach with the SVS as its motivating example. Next, Section 4 presents our experimental results. Following, Section 5 overviews related work. Finally, Section 6 discusses our findings and presents future directions.

2. BACKGROUND

This section presents background material on SVSs, goal-oriented requirements modeling, genetic algorithms, and software testing.

2.1 Smart Vacuum Systems

SVSs, most notably the iRobot Roomba,[2] are currently available in the consumer market. An SVS cleans a desired area by monitoring sensor data to balance path planning, power conservation, and safety concerns. Sensors may include bumper sensors, cliff sensors, motor sensors, and object sensors. Bumper sensors indicate when the robot has collided with an object, cliff sensors detect empty space and prevent the robot from falling down stairs, motor sensors provide feedback on individual velocities and power modes for the wheels and suction units, and object sensors can be used to detect and identify objects near the robot. A controller analyzes the incoming sensor data to ensure that the proper path, power saving mode, and failsafe modes are selected as necessary.

An SVS can be modeled as an adaptive system [2], and as such, can self-reconfigure at run time by dynamically performing mode changes [24]. Mode changes enable the SVS to mitigate uncertainty by selecting an optimal configuration. Uncertainties can include unexpected power drain, distribution of dirt within the room, and unexpected obstacles. Mode changes to mitigate these uncertainties can include reduced power consumption modes, different pathfinding algorithms, and obstacle avoidance measures.

[2]See http://www.irobot.com/

2.2 Goal-Oriented Requirements Modeling

Goal-oriented requirements engineering (GORE) graphically captures high-level objectives and constraints that a system must satisfy, and can be used to guide the elicitation and analysis of requirements, incorporating assumptions and expectations of the executing environment. Furthermore, a *functional* goal declares a service that the system-to-be must provide to its stakeholders, a *non-functional* goal imposes a quality constraint or criterion upon the delivery of those services, a *safety* goal declares a critical safety objective that mitigates dangerous situations and must always be satisfied, and a *failsafe* goal specifies a goal that provides a safe fallback in case of system failure. Functional goals can be classified as invariant (denoted by the keyword "Maintain"), requiring that they are always satisfied by the system, or non-invariant (denoted by the keyword "Achieve"), indicating that they may be temporarily unsatisfied at run time. Safety and failsafe goals may also be designated as either functional or non-functional. Moreover, goals can be *satisficed*, or satisfied to a certain degree, possibly based on subjective preference [8].

The GORE process gradually decomposes high-level goals into finer-grained subgoals [31] using a directed acyclic graph, where each node represents a goal and an edge represents the corresponding goal refinement. KAOS [31] provides a framework for systematically performing goal refinement via AND or OR refinements. An AND-refined goal is satisfied only if all its subgoals have also been satisfied. OR-refined goals are satisfied if at least one subgoal has also been satisfied. The refinement process continues until each leaf-level goal has been assigned to an agent that is responsible for that goal's satisfaction, and is then considered a requirement.

The KAOS goal model in Figure 1 captures the functional requirements of the SVS application. The SVS must (A) successfully clean at least 50% of the dirt within the room. To accomplish this task, the SVS must (B) operate efficiently to conserve battery power (F) while providing movement (E) and suction (G). The SVS can then operate in either normal power modes for speed (L) and suction (N) or reduced power modes for speed (K) and suction (M) to conserve battery power (F). Furthermore, the SVS must also (C) clean the room effectively by either selecting a 10 second (H) random (O) or straight (P) path, or 20 second (I) spiral (Q) path plan until the simulation completes. The SVS must also consider safety (D) as a high-level system goal. Moreover, if a safety violation occurs then the SVS must activate (J) a failsafe mode, with safety violations comprising a collision with specific obstacles (R) (such as pets or children) and avoiding liquid spills, cliffs, and objects that would otherwise damage the SVS itself (S). Goal (A) demonstrates an AND-decomposition where the room can only be successfully cleaned if the battery retains enough power (B), the desired area has been covered via path planning (C), and no safety concerns have been violated (D). Conversely, Goal (H) provides an example of an OR-refinement in that an area can be cleaned for 10 seconds if the SVS follows either a *RANDOM* path (O) or follows a *STRAIGHT* path (P).

2.3 MAPE-T Feedback Loop

The MAPE-T feedback loop [14] is a run-time testing loop that assists an SAS in continually satisfying its requirements. Similar to the MAPE-K feedback loop that governs proper SAS design and execution [19], MAPE-T provides a

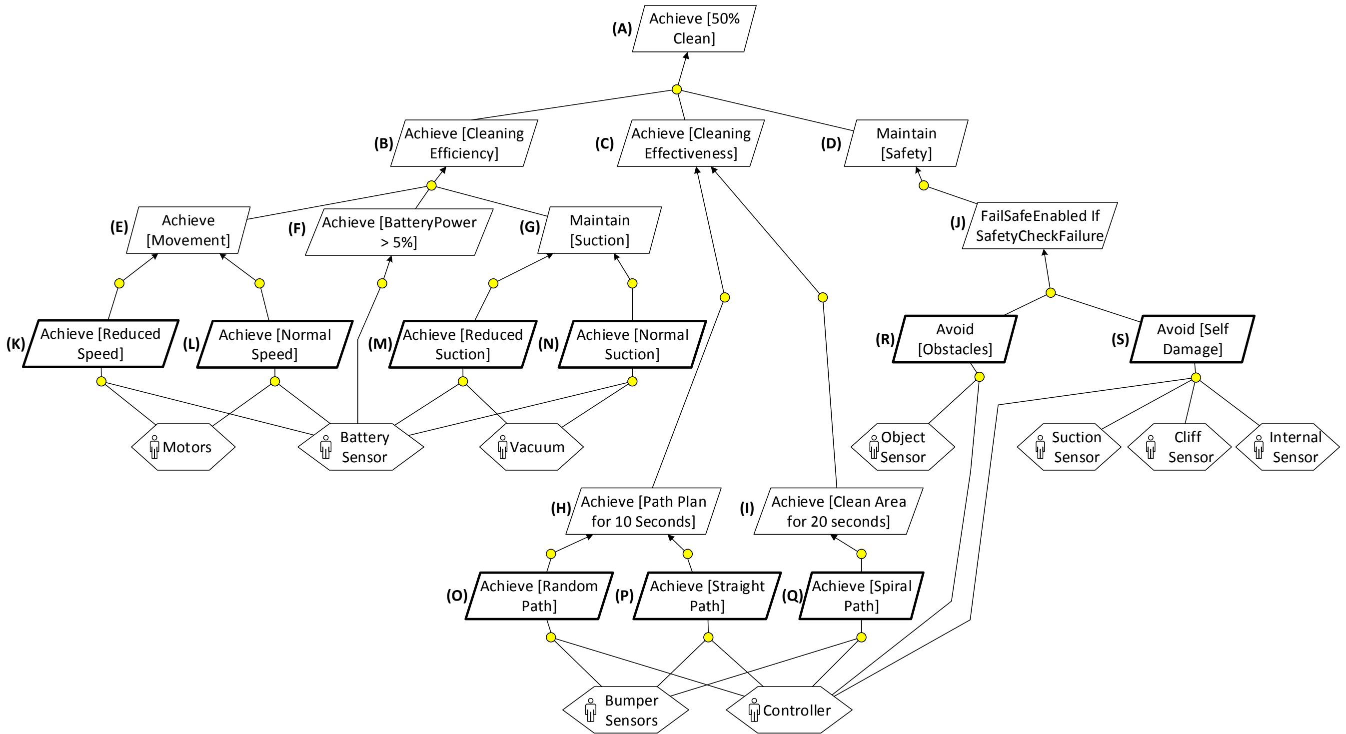

Figure 1: **KAOS goal model of the SVS application.**

framework for *Monitoring*, *Analyzing*, *Planning*, and *Executing* run-time test adaptation. *Monitoring* is facilitated by observing key elements in the system and environmental configurations and providing relevant testing information to the SAS in case of contextual change. *Analyzing* consumes the current operational context and selects a proper subset of test cases from an existing test specification. *Planning* determines the proper time during execution as to when the test plan will be evaluated during SAS execution. Finally, *Executing* runs the test cases alongside the SAS and analyzes the results. Furthermore, *Executing* adapts the test cases as necessary to ensure that they are applicable and/or relevant to their current environment, while maintaining overall test specification validity. Veritas is one approach for realizing the MAPE-T feedback loop.

2.4 Genetic Algorithms

A genetic algorithm [18] is a stochastic, search-based approach used to efficiently solve complex optimization problems. Candidate solutions are represented as a set of individuals within a population, and a fitness function evaluates the performance of each individual, thereby guiding the search process towards promising areas within the available solution space. New individuals are generated with evolutionary operators such as crossover and mutation. Crossover combines portions of existing individuals to form new individuals that, ideally, attain higher fitness values. Mutation randomly modifies an individual to maintain diversity within the population. The evolution process continues until the maximum number of specified generations is completed.

(1+1)-ONLINE Evolutionary Algorithm. The (1+1)-ONLINE Evolutionary Algorithm (EA) [4], based on prior work in evolutionary strategies [9, 29], facilitates run-time evolution by adapting genomes in a low-impact manner during program execution. Generally, EAs must run offline due to the large amount of time required for a full evolutionary run. The (1+1)-ONLINE EA sacrifices searching power for execution speed. Within the (1+1)-ONLINE EA, there is only a single parent individual and single child individual in the population, with the creation of the child accomplished by mutating the parent. A mutation value σ provides local or global search of the solution space depending on stagnation of the fitness value, where stagnation is defined as a minor or no change to the fitness value, indicating a local optima within the search space. Each individual is provided a set amount of time for evaluation during execution. The individual with the higher fitness is allowed to survive, and the individual with the lower fitness value is discarded. A new child is then created from the current parent individual, and the process repeats throughout execution. Given that mutation is driving the evolutionary process with a small population size, the ability to exhaustively search all areas within the solution space is limited, however (1+1)-ONLINE EA is instead able to provide lightweight evolutionary search during execution.

2.5 Software Testing

Software testing provides assurance that a system is operating according to its requirements specification, which can be accomplished with a number of different strategies [3, 17], including structural, functional, and regression testing. Structural testing comprises coverage metrics such as branch and data flow coverage. Functional testing validates a system by testing against a specification. Finally, regression testing validates a system using an existing test specification following a major change that would necessitate retest-

ing, such as a new software release or a change in operating context [28, 35]. This paper adapts functional test cases and moreover uses regression testing to validate test case execution following system and/or environmental changes.

3. APPROACH

Veritas is an online assurance technique for adapting test cases at run time in response to changing system and environmental conditions. For discussion purposes, we define a *test specification* to be a collection of all possible test cases relevant to the SAS, and a *test plan* to be a subset of test cases from the test specification that may be adapted and executed at run time. Veritas monitors the operating context for evidence of change, and then determines which test cases are relevant to current conditions. Next, Veritas executes the test plan and analyzes the results to determine if each executed test case is valid and if adaptations are necessary. Finally, if adaptation is required, then Veritas executes the (1+1)-ONLINE EA to generate a set of new test cases. In this section, the inputs, assumptions, and expected outputs are described. Then, the Veritas approach is presented with each of its key elements described in turn.

3.1 Inputs, Assumptions, and Expected Outputs

Veritas requires five elements as input: a goal model of the system, a set of utility functions for high-level requirements monitoring [33], an executable specification or prototype of the SAS, a set of monitoring elements, and a test specification comprising all possible test cases. Each of these elements is next described in detail.

Inputs and Assumptions. First, the system goal model is needed to capture the high-level functionality of the SAS (Figure 1 depicts the SVS goal model). The goal model is assumed to represent the intent specified within the software requirements.

Next, utility functions must be derived from the goal model to quantify each goal's performance throughout the SAS execution. An example of a utility function that measures the satisfaction of power conservation with respect to SVS movement (Goal (F)) is presented as follows in Equations 1 and 2:

$$utility_{Goal_F} = BatteryDecay, \tag{1}$$

where

$$BatteryDecay = \begin{cases} 1.00 & \text{if } BatteryCharge >= 75\%, \\ 0.75 & \text{if } BatteryCharge >= 50\%, \\ 0.50 & \text{if } BatteryCharge >= 25\%, \\ 0.25 & \text{else.} \end{cases} \tag{2}$$

The returned values are then used to determine if an SAS self-reconfiguration is necessary. For example, the above equations reveal the current state of the battery. If the remaining charge falls below a specified percentage, for instance 50%, then the SVS self-reconfigures to reduce power to the wheel motors, effectively conserving battery power while still achieving motion to satisfy upper-level goals (A) and (B). Furthermore, Veritas uses the utility values for run-time validation of its test results. Continuing with our prior example, a test case that monitors the current power mode of the SVS would be considered valid if the utility value ($utility_{goal_E}$) is violated *and* the test case determines that the SVS has not successfully entered a reduced power mode (this result implies that there is a problem with the self-reconfiguration capabilities of the SVS). For the purposes of this paper, the utility functions are assumed to have been derived correctly and, moreover, provide an accurate quantification of system behavior throughout execution in all provided environments.

An executable specification of the SAS that captures the system and environmental contexts must be provided. The executable specification is responsible for executing the SAS behavior in a simulation environment that can generate all possible operating contexts that the SAS may encounter.

A requirements engineer specifies a set of elements to properly monitor and quantify operating conditions. First, the ENV element describes a specific object or behavior to be analyzed. Second, MON specifies the variable that monitors the ENV element. Lastly, REL describes the relationship between ENV and MON. For example, in power conservation mode, the SVS can reduce power to its wheels (ENV). To accomplish this task, the SVS queries the wheel motor sensors (MON) to determine the current amount of torque being applied to the wheels. Upon entering power conservation mode, the SVS then commands the wheel motors to reduce the applied torque (REL), effectively reducing the power consumption by the wheel motors and extending the time that the SVS can operate before running out of battery power.[3]

A test specification must also be provided for Veritas to analyze. The test specification includes test cases that provide full coverage of a requirements specification, as Veritas can only perform testing at run time by selectively executing test cases and mutating test case parameters, and is not intended to dynamically add or remove test cases during execution. Furthermore, each test case must be specified as invariant or non-invariant. Invariant test cases are precluded from adaptation, and are generally associated with critical testing such as safety or failsafe concerns. Non-invariant test cases relate to functional requirements and behaviors, and as such are reconfigurable at run time. For run-time validation, each test case must also be associated with one or more of the previously derived utility values for the goal model. Example SVS test cases can be found in Table 1. For instance, Test Case 3 ensures that the SVS's object sensor (MON) can detect large dirt particles (ENV) within a radius of $0.5m$ (REL). Therefore, the expected value of an internal variable (i.e., $ObjectSensor.DetectRadius$) must equal $0.5m$. If this test case fails, then adaptation may be warranted, as operating conditions may have changed such that the SVS can no longer detect objects within the specified radius. As Test Case 4 is defined to be non-invariant, adaptation is allowed. The expected value can be mutated within a pre-defined tolerable range of $[0.25m, 0.75m]$. Furthermore, satisfaction of this particular test case contributes to satisfying testing of Goals D and R.

Expected Outputs. Upon completion of the simulation, Veritas outputs a set of tuples comprising the *environmental configuration*, *system configuration*, and *test specification configuration*, as well an *adaptation trace*. The tuple provides a snapshot of the overall SAS state at each point of test execution and test adaptation. This information can be analyzed offline by an SAS engineer or test engineer to gain insights into the types of system and environmental config-

[3]For this paper, we consider the REL element to be the *expected value* for each test case.

Table 1: **Examples of SVS test cases.**

	Test Case (ENV)	Agent (MON)	Expected Value (REL)	Type	Acceptable Value	Goal Constraint
1	Test suction tank for large objects	$InternalSensor$	$InternalSensor.NoLargeObjects = TRUE$	Invariant	$TRUE$	$utility_{Goal_D}$, $utility_{Goal_J}$
2	If SVS falls off cliff, ensure that failsafe mode was enabled and all power was disabled	$InternalSensor$, $Controller$	$Controller.FailSafeActive = TRUE$	Invariant	$TRUE$	$utility_{Goal_D}$, $utility_{Goal_J}$
3	Verify large dirt particle detection within $0.5m$	$ObjectSensor$	$ObjectSensor.DetectRadius = 0.5m$	Non-invariant	$[0.25m, 0.75m]$	$utility_{Goal_D}$, $utility_{Goal_R}$
4	Ensure that a random angle on $[0.0\ rad, (\pi/2.0)\ rad]$ was selected while following the *RANDOM* path plan	$InternalSensor$, $Controller$	$InternalSensor.TurnAngle \in [0.0\ rad, (\pi/2.0)\ rad]$	Non-invariant	$[-(\pi/2.0)\ rad, \pi\ rad]$	$utility_{Goal_O}$

urations the SAS is experiencing. Furthermore, the adaptation path provides information regarding the contextual changes that triggered execution of the test specification, the test results, and the adaptation path for each affected test case.

3.2 Veritas Process

The data flow diagram (DFD) in Figure 2 provides an overview of the Veritas process as it executes in parallel to the SAS. Each step is next described in detail.

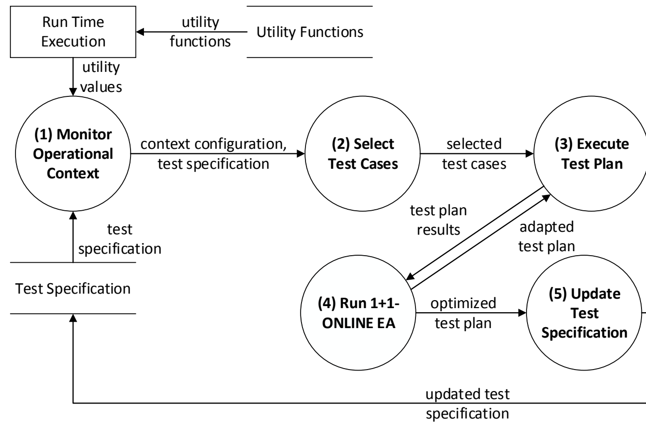

Figure 2: **DFD diagram of Veritas process.**

(1) Monitor Operational Context. Veritas first monitors the operational context for evidence of change. Monitoring can be accomplished by passively monitoring the utility values (e.g., the SVS's battery power has fallen below 5% and therefore Goal (F) is violated) and actively monitoring the environment via instrumented code (e.g., monitoring the object sensor to detect obstacles such as pets or children).

(2) Select Test Cases. Once a contextual change has been detected, Veritas then analyzes the current configuration of the system and environment to determine if an adaptation of the current test plan is necessary. Veritas selects a set of appropriate test cases based on parameters from the current operating context, such as system mode and observed features within the environment. A test case is applicable if it can be measured within the current context. For instance, testing that the SVS has successfully completed the *SPIRAL* path plan is invalid if the SVS is currently executing the *STRAIGHT* path plan. Following test case selection, Veritas then generates a test plan comprising all selected test cases.

(3) Execute Test Plan. Veritas then executes the test plan and returns a *measured fitness subfunction value* and *test result* for each test case. While test cases typically return a boolean *pass* or *fail*, providing a fitness value instead yields an extra metric for comparison of test case usefulness. A specific fitness subfunction is defined for each test case based on its type and designation as invariant or non-invariant (see Table 2).

Table 2: **Individual test case fitness subfunctions.**

Test type	REL	Fitness
Invariant	Exact Value	if ($value_{measured} == value_{expected}$) then $fitness_{measured} = 1.0$ else $fitness_{measured} = 0.0$
Invariant	Range of Values	if ($value_{measured} \in [value_{low_boundary}, value_{high_boundary}]$) then $fitness_{measured} = 1.0$ else $fitness_{measured} = 0.0$
Non-invariant	Exact Value	$fitness_{measured} = 1.0 - \frac{\lvert value_{measured} - value_{expected}\rvert}{\lvert value_{expected}\rvert}$
Non-invariant	Range of Values	if ($value_{measured} \in [value_{low_boundary}, value_{high_boundary}]$) then $fitness_{measured} = 1.0$ else $fitness_{measured} = 1.0 - \lvert\frac{value_{measured} - value_{optimal}}{value_{optimal}}\rvert$

Different types of fitness calculations are used based on the type of variable measured by the test case. For those test cases that are monitoring an *exact* value, the test's measured value ($value_{measured}$) is compared to the expected value ($value_{expected}$). Variables defined as a *range*, conversely, expect the measured value to fall within pre-determined boundaries (i.e., $[value_{low_boundary}, value_{high_boundary}]$). Fur-

thermore, if a *range* variable falls outside of those boundaries, then its optimal value (i.e., $value_{optimal}$) is defined as the nearest boundary to the measured value, as defined in Equations 3 and 4.

$$\begin{aligned} d_{low} &= |value_{measured} - value_{low_boundary}| \\ d_{high} &= |value_{high_boundary} - value_{measured}|, \end{aligned} \tag{3}$$

where

$$value_{optimal} = \begin{cases} value_{low_boundary} & \text{if } (d_{low} < d_{high}), \\ value_{high_boundary} & \text{else.} \end{cases} \tag{4}$$

Furthermore, the test case fitness calculation also considers the inclusion within the low and high boundaries as an impetus for providing a fitness boost for the evolutionary process. A measured value (i.e., $value_{measured}$) that is not equal to the expected value ($value_{expected}$), but is still within the specified range of acceptable values (i.e., $[value_{low_boundary}, value_{high_boundary}]$) is rewarded for still being considered a valid result. The aggregate fitness for each test case is then calculated as a weighted sum comprising the measured fitness subfunction and the aforementioned fitness boost for validity, with weights $\alpha_{measured}$ and α_{valid} defining the relative importance of each fitness subfunction. Test case fitness is presented as follows in Equations 5 and 6.

$$\begin{aligned} fitness_{test_case} = \ & \alpha_{measured} * fitness_{measured} + \\ & \alpha_{valid} * ValidResult, \end{aligned} \tag{5}$$

where

$$ValidResult = \begin{cases} 1.0 & \text{if } value_{measured} \text{ is valid,} \\ 0.0 & \text{else.} \end{cases} \tag{6}$$

As invariant test cases cannot be adapted, they may only pass or fail. This result is converted to a fitness value as 1.0 and 0.0, respectively. Non-invariant test cases must provide a measure of performance and represent their fitness as the distance to an optimal value. For test cases designated as exact, the fitness is calculated as the normalized distance between the measured value ($value_{measured}$) and the expected value ($value_{expected}$). Test cases designated as ranges define fitness as the normalized distance between the measured value and the *closest* boundary value (either $value_{low_boundary}$ or $value_{high_boundary}$). Test cases that are not relevant to the operating context cannot be accurately executed, and therefore have a fitness value of 0.0. Overall fitness of the test specification is calculated as the average fitness of all test cases relevant to the current operating context. Furthermore, a test case may also be designated as having passed or failed. For the purposes of this paper, a test case is considered passed if it has a fitness of 0.75 or greater, where 0.75 was selected as a baseline to trigger the adaptation process more often during execution based on observed test case fitness values.

(4) Run 1+1-ONLINE EA. Following execution of the test plan, Veritas then runs the (1+1)-ONLINE EA on test cases marked for adaptation. The (1+1)-ONLINE EA comprises a population of two individuals: a parent and a child, where the child is created by mutating the parent. The parent is first evaluated and then placed into a parent archive following completion of its evaluation period (e.g., a set amount of time has expired), and then the child is evaluated in turn. The individual with the higher fitness value is retained for the next iteration. Veritas implements the (1+1)-ONLINE EA by considering test cases to be individuals and evaluation to be a single test case execution. The initial composition of the parent archive is populated by the first evaluation of each test case. A DFD of the (1+1)-ONLINE EA process as used by Veritas is presented in Figure 3 and detailed in the following subsections.

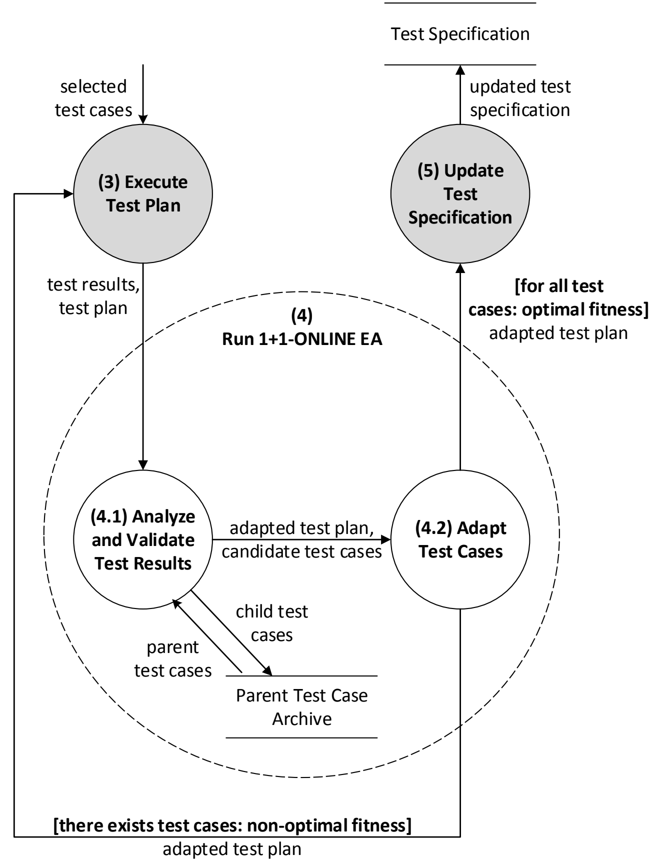

Figure 3: **DFD diagram of 1+1-ONLINE EA.**

(4.1) Analyze and Validate Test Results. Veritas analyzes the non-invariant test results to determine if a test case adaptation is warranted. Test cases that have passed are analyzed based on their calculated fitness values. Failing test cases are automatically marked for adaptation. Both passing and failing test cases undergo validation. Validity of a test case is determined by the relationship between a utility function and a test case, with the following four possible cases:

C1. Utility function is satisfied, and test case has passed (i.e., SAS behaves as expected)

C2. Utility function is not satisfied, and test case has failed (i.e., SAS requires reconfiguration)

C3. Utility function is satisfied, and test case has failed (i.e., false negative, test case requires adaptation)

C4. Utility function is not satisfied, and test case has passed (i.e., false positive, test case requires adaptation)

C1 and C2 indicate that the test case concurs with the results of each associated utility value, indicating that the

test is valid. Moreover, C2 can indicate a problem with the codebase, potentially requiring code adaptation. C3 and C4 imply that the test case requires adaptation. As the utility value in C3 indicates that the system is performing as it should, then a failing test case implies that the test case is invalid and requires adaptation. Conversely, C4 also requires adaptation as the test case has passed, and yet the utility value is indicating that the system is operating incorrectly. This behavior implies that the test case is no longer applicable to the operating context.

For instance, a test case that monitors the health of the cliff sensors is considered valid if the corresponding utility value(s) indicate that the SVS has not enabled its failsafe mode (Goal (J)) and has not fallen down a set of stairs (Goal (D)). However, if the same test case is passing (implying the cliff sensors are fully functional) and yet the SVS has fallen off a step (Goals (J) and (D) were violated), then the test case is *in disagreement* with its utility functions and therefore considered invalid. Moreover, if the test case fails and the utility value(s) indicate goal violation, then the test case is *in agreement* with the utility value(s), and either an SAS self-reconfiguration or sensor replacement is necessary. Test cases that are invalid (i.e., disagree with their corresponding utility functions) are marked for adaptation.

Next, the test case results from the child test case are evaluated against its parent. Prior fitness values for each test case selected for adaptation are retrieved from the parent test case archive for comparison. If the calculated fitness determined in Step (3) is higher than the existing version in the archive, then the child replaces the parent in the archive. If the parent's fitness was higher, then the child test case is replaced by the parent in the current test plan. Furthermore, each selected test case is annotated with information regarding which test case (i.e., parent or child) has been chosen, as this information will determine the amount of mutation possible in Step 4.2. A test case that has been executed for the first time (e.g., at the start of SAS execution) is copied into the parent test case archive to be used as a baseline. The resulting test case is then adapted as follows:

(4.2) Adapt Test Cases. Following the validity check, Veritas now searches for a new subset of test cases to include in the test plan. First, a mutation value σ is determined based upon performance of the test case relative to its parent, as determined in Step 4.1. If the parent test case performed better than its child, then σ is set to search the global search space in an effort to find a more representative test case. Conversely, if the child performed better, then σ is set to explore the local search space relative to the child. The boundary elements of each selected test case are then mutated by σ, and the REL value is randomly generated within those bounds. Each value is required to stay within the previously defined tolerance to ensure that safety or failsafe concerns are not violated by adaptation. All active test cases that have been determined to require adaptation are subjected to the adaptation procedure.

To illustrate the adaptation process, consider a camera sensor that has a scratched lens. Initially the camera could detect objects at $0.5m$ according to its design, however the damage has reduced its ability to only sense objects at a distance of $0.3m$. An associated test case states that the SVS must use the camera sensor to detect objects within $0.5m$ as is defined within its requirements, however the test engineer has previously specified a safety tolerance (i.e., range of values with which Veritas can adapt test case parameters) that states that as long as an object is detected within a range of $[0.25m, 0.75m]$, the SVS can continue to execute as normal. Given that the new sensing distance of $0.3m$ is within the tolerated range, a Veritas-adapted test case that expects a sensing distance of $0.28m$ will be considered more fit than the original distance of $0.5m$, as it is considered both valid and closer to the measured value.[4]

Following adaptation, Veritas updates the current test plan with either the adapted, child test cases (for testing at the next possible opportunity) or with the previously tested parent test case (as long as the parent's fitness was higher than the child's fitness). Test case fitness is considered *optimal* if the current fitness has surpassed the parent's fitness. If any test case fitness is *non-optimal*, then the (1+1)-ONLINE EA is restarted with test cases considered non-optimal. If all test case fitnesses in the current test plan are optimal, then no further test case adaptation is required until a new context change is detected.

(5) Update Test Specification. Veritas next updates the test specification with the current state of the test plan, where the test specification comprises all defined test cases, and the test plan comprises the set of applicable test cases to the current operational context, with each test case having either been adapted by Veritas or retrieved from the parent test case archive. The updated test specification is again used as a repository of possible test cases for run-time testing the next time that Veritas is executed during SAS execution.

4. EXPERIMENTAL RESULTS

This section describes the experimental setup and discusses the results from applying Veritas to the SVS application.

4.1 Experimental Setup

For this paper, we implemented the SVS application as a completely autonomous robot that receives input from sensors and responds accordingly. The SVS comprises a set of bumper sensors to determine if a wall has been encountered, an object sensor to measure distance between objects within the room and the robot, a cliff sensor to prevent the robot from falling, an internal sensor that monitors the internal health of the robot, and wheel and suction sensors that monitor the health of the wheels and suction unit, respectively. Each sensor has a specified probability of failure, and a probability of occlusion. Failure causes the sensor to cease functioning, and occlusion causes the sensor to receive partial or corrupted input. The SVS also has a controller that responds to sensor input and self-reconfigures as necessary. Example conditions that could cause a reconfiguration are cliff detection, low power, or an activated failsafe mode. Furthermore, the environment comprises four walls and a set of objects that can be instantiated within the room. These objects include dirt that can be safely vacuumed, large dirt that cannot be safely vacuumed, a downward step that the robot must avoid, collideable objects (e.g., walls or pillars) that the robot may safely contact, and non-collideable objects (e.g., pets or children) that the robot must avoid.

[4]The occlusion of the sensor will still be noted by the utility function as an issue that may require self-reconfiguration or replacement for safety reasons as part of normal SAS operation.

The test specification comprised 26 separate test cases, 10 of which tested safety cases, and 4 that tested failsafe cases. The safety and failsafe test cases were considered to be invariants, and therefore exempted from adaptation. The remaining 12 cases tested system functionality and were considered non-invariant and allowed for adaptation. The fitness functions applied to each test case are presented in Table 2. Furthermore, Veritas was configured to consider any test result value below 0.75 to be a failure and therefore a target for adaptation. The SVS simulation was executed for 120 timesteps and required to vacuum at least 50% of the available dirt within the room. Furthermore, the fitness function weights $\alpha_{measured}$ and α_{valid} were set to 0.4 and 0.6 respectively to maximize the amount of test results that measured *valid* results.

Environmental uncertainties, such as amount, location, and distribution of dirt; instantiation of objects that can damage the robot (e.g., liquid spills); and instantiation of objects that the robot must navigate around (e.g., stationary pets) may be applied to the SVS at the start of the simulation. Furthermore, cliffs, varying room sizes, and different floor frictions further contribute to the environmental uncertainty that the SVS must mitigate. System uncertainty manifested in the form of sensor failure or occlusion, each of which may manifest at any given point during the simulation. To simulate these uncertainties, 15 unique operational contexts were generated by Loki [27], a novelty search-based approach for generating unique system and environmental configurations. The SVS was tested within each Loki environment sequentially to determine the impact of environmental uncertainty on run-time testing.

In order to validate our approach, we compared and evaluated the resulting test cases generated by Veritas with a set of manually-derived test cases (hereafter the "Control") that had no test case adaptations applied at run time. In order to test all the mode changes that might occur at run time, the Control comprises all of the test cases from the test specification where test cases are configured to test the original environmental conditions. For statistical purposes, we conducted 50 trials of this experiment and, where applicable, plotted or reported the mean values with corresponding error bars or deviations.

4.2 Run-Time Testing

For this experiment, we define the null hypothesis H_0 to state that "there is no difference between Veritas test cases and unoptimized test cases." Furthermore, we define an alternate hypothesis H_1 to state that "there is a difference between Veritas test cases and unoptimized test cases." Figure 4 presents boxplots of the mean fitness values observed throughout each experiment for Veritas test cases and a Control that did not adapt test cases at run time. As this figure demonstrates, Veritas test cases achieve statistically significant higher fitness values than the static, unoptimized test cases (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).[5] Moreover, these results demonstrate that Veritas test cases adapt to be more representative of their environment (Step 4.1, C1), enabling us to reject our null hypothesis H_0 that no difference exists between the two sets of test cases.

[5]For each reported result in this section, the Wilcoxon-Mann-Whitney U-test was selected to measure statistical significance, due to the non-normal distribution of each dataset.

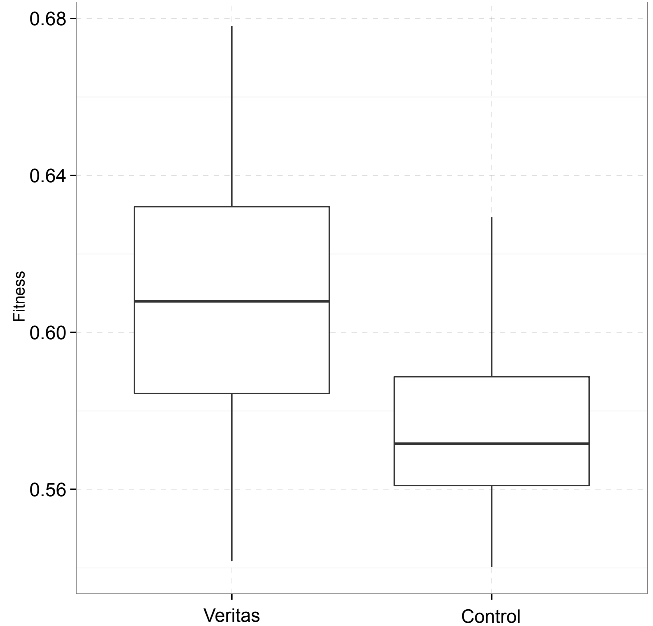

Figure 4: **Comparison of fitness between Veritas and Control.**

Figure 5 presents the mean number of test case failures experienced throughout the SVS simulation (i.e., moving through several Loki-generated environments and system reconfigurations to represent environment condition change and system reconfiguration) between Veritas test cases and the Control test cases. The utility functions for each failure in this figure, however, were satisfied, implying that Veritas can minimize the amount of *false negatives* (Step 4.1, C3) at run time.

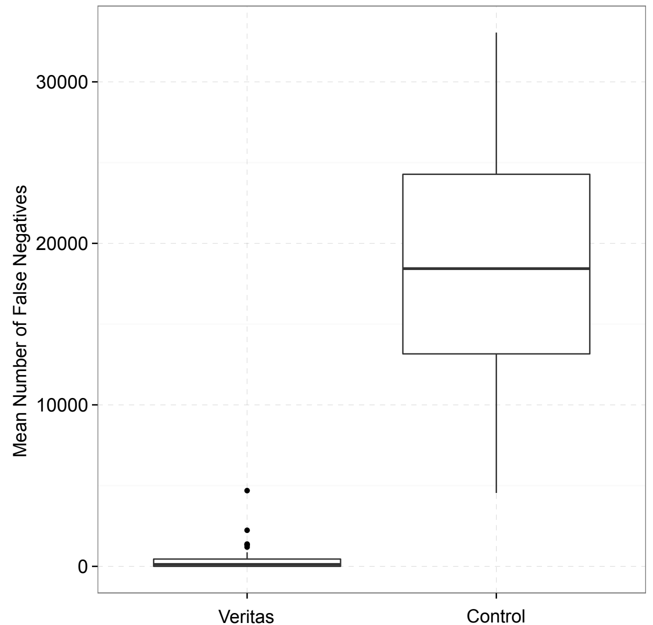

Figure 5: **Comparison between Veritas and Control test cases.**

As demonstrated by this plot, Veritas test cases fail significantly less than those that are unoptimized. Due to chang-

ing system and environmental conditions, Veritas was able to adapt its test cases to fit the environmental conditions within the specified safety tolerance, whereas the unoptimized test cases could not handle the unexpected uncertainties, and could become invalid following a context change.

As Veritas has been shown to maximize fitness (Figure 4) and minimize false negatives (Figure 5), we can safely reject H_0 and accept H_1, and furthermore state that Veritas can optimize run-time test cases to fail less often in uncertain environments while maintaining a valid representation of the operational context.

Threats to Validity. This research was intended as a proof of concept study to determine the feasibility of the MAPE-T feedback loop in run-time optimization of test cases for varying environments. We applied the Veritas approach to optimize test cases at run time by harnessing a (1+1)-ONLINE EA. As a point of reference, we compared the resulting Veritas-generated test specifications with those that were unoptimized throughout execution. The SVS was modeled as an adaptive system by describing mode changes as the impetus for self-reconfiguration. As such, one threat to validity is if Veritas will achieve similar results in other adaptive system domains not involving mode changes. Another threat involves the validity of the utility functions as derived by the requirements engineer, given their importance to validating the adapted test cases. Furthermore, as Veritas was intended as a proof of concept, we did not fully study the impact of run-time testing on a live system (e.g., performance impact). Sensitivity of test cases is another concern, however we did not intentionally derive them as such. Finally, the representativeness of the initially derived test cases with respect to the encountered system and environmental configurations throughout execution is another threat to validity.

5. RELATED WORK

This section overviews related work on search-based software testing and run-time testing.

Search-Based Software Testing. The field of search-based software testing explores how software testing can be augmented with search-based techniques, such as genetic algorithms or hill climbing. These search-based techniques have been applied to many branches of software testing, including structural, model-based, mutation, and regression testing [17, 22], as the exploratory nature of these techniques can automatically provide a representative set of test cases. EvoSuite [13] and Nighthawk [1] are frameworks that leverage evolutionary computation for test suite generation and unit test case instantiation, respectively. While search-based techniques can guide the creation of a diverse and representative set of tests automatically, these techniques tend to be design-time approaches, whereas Veritas searches for optimal test case parameter combinations at run-time.

Run-Time Testing. Testing software at run time provides useful insights into the behavior and correctness of a system as it executes. Run-time testing has been previously implemented as model-based [16] and an adaptation of reinforcement learning [32]. Unit testing has also been incorporated into run-time verification via extensions to an assertion language [23]. Previously, run-time software testing of real-time systems has been facilitated by recording live execution traces to a secondary computer and replaying the data in parallel, examining the traces for faults [30]. Filieri *et al.* [11] used Markov models to handle uncertainty within run-time monitoring. Run-time testing has also been applied using agent-based approaches [25]. Each of these techniques has facilitated run-time testing, however, Veritas combines evolutionary search for optimal test cases with goal-based validation.

Regression Testing. Regression testing provides assurance that software is valid following major changes to the system [28, 35]. As such, the selection of test cases is an important consideration in determining what must be revalidated. An excellent analysis of different test case selection techniques [28] describes different approaches to test case selection, including analysis of linear equations, symbolic executions, and program dependence graphs. Moreover, test case prioritization can reduce the cost of regression testing to ensure that important and relevant test cases are executed first [10]. The Veritas framework actively monitors operating conditions via pre-defined agents to determine which test cases are selected for regression testing. Currently, Veritas does not prioritize test case execution, opting instead to execute each in parallel.

6. CONCLUSION

In this paper we presented Veritas, a run-time testing framework that harnesses a (1+1)-ONLINE EA to search for optimal test case parameters as uncertainty within the system and environment manifests. Veritas is an example realization of the MAPE-T feedback loop, comprising the *Monitoring*, *Analyzing*, *Planning*, and *Executing* states. Specifically, Veritas monitors an SAS for contextual change, generates an appropriate test plan, analyzes the test results, and adapts test cases as necessary. Veritas then validates each test case against a set of corresponding utility functions to ensure that each test case is representative of the intended system design. We demonstrated the applicability of Veritas on an SVS that was tasked to clean a room while maintaining functional and safety concerns. The SVS was subjected to uncertainties in the form of randomly placed objects, steps that must be avoided, and objects that may otherwise damage the SVS. Experimental results confirmed that Veritas-adapted test cases performed better and reduced the amount of false negative results more often than non-adapted test cases across varying environments. Future directions for this work include investigating other techniques for planning run-time test execution, exploring other evolutionary techniques such as hill climbing and simulated annealing for run-time adaptation, and providing feedback to the MAPE-K loop to trigger a system reconfiguration.

7. ACKNOWLEDGMENTS

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, or other research sponsors.

8. REFERENCES

[1] J. H. Andrews, T. Menzies, and F. C. Li. Genetic algorithms for randomized unit testing. *IEEE Trans. on Software Engineering*, 37(1):80–94, January 2011.

[2] N. Bencomo and A. Belaggoun. Supporting decision-making for self-adaptive systems: from goal models to dynamic decision networks. In *Requirements Engineering: Foundation for Software Quality*, pages 221–236. Springer, 2013.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE '07*, pages 85–103, 2007.

[4] N. Bredeche, E. Haasdijk, and A. Eiben. On-line, on-board evolution of robot controllers. In P. Collet, N. Monmarché, P. Legrand, M. Schoenauer, and E. Lutton, editors, *Artifical Evolution*, volume 5975 of *Lecture Notes in Computer Science*, pages 110–121. Springer Berlin Heidelberg, 2010.

[5] 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.*, pages 53 –62, june 2012.

[6] B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, and et al. Software engineering for self-adaptive systems. In B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, editors, *Software Engineering for Self-Adaptive Systems*, chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[7] 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*, pages 468–483, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. Non-functional requirements. *Software Engineering*, 2000.

[9] S. Droste, T. Jansen, and I. Wegener. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1):51–81, 2002.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[11] A. Filieri, C. Ghezzi, and G. Tamburrelli. Run-time efficient probabilistic model checking. In *Proc. of the 33rd International Conference on Software Engineering*, pages 341–350, Waikiki, Honolulu, Hawaii, USA, 2011. ACM.

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

[13] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 416–419, Szeged, Hungary, 2011. ACM.

[14] 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*, SEAMS '13, pages 169–174. IEEE Press, 2013.

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

[16] H. J. Goldsby, B. H. C. Cheng, and J. Zhang. Models in software engineering. In H. Giese, editor, *Models in Software Engineering*, chapter AMOEBA-RT: Run-Time Verification of Adaptive Software, pages 212–224. Springer-Verlag, Berlin, Heidelberg, 2008.

[17] M. Harman, P. McMinn, J. Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In B. Meyer and M. Nordio, editors, *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 1–59. Springer Berlin Heidelberg, 2012.

[18] J. H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA, 1992.

[19] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, jan 2003.

[20] M. Lajolo, L. Lavagno, and M. Rebaudengo. Automatic test bench generation for simulation-based validation. In *Proc. of the Eighth International Workshop on Hardware/Software Codesign*, pages 136–140, San Diego, California, United States, 2000. ACM.

[21] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56 – 64, july 2004.

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

[23] E. Mera, P. Lopez-García, and M. Hermenegildo. Integrating software testing and run-time checking in an assertion verification framework. In *Logic Programming*, pages 281–295. Springer, 2009.

[24] S. Neema, T. Bapty, and J. Scott. Development environment for dynamically reconfigurable embedded systems. In *Proc. of the International Conference on Signal Processing Applications and Technology. Orlando, FL*, 1999.

[25] 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.

[26] 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*, pages 16 –22, aug. 2011.

[27] A. Ramirez, A. Jensen, B. H. C. Cheng, and D. Knoester. Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 568 –571, nov. 2011.

[28] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *Software Engineering, IEEE Transactions on*, 22(8):529–551, 1996.

[29] H.-P. Schwefel. *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

[30] 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*, 16(8):897–916, 1990.

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

[32] M. Veanes, P. Roy, and C. Campbell. Online testing with reinforcement learning. In *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253. Springer, 2006.

[33] 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*, pages 70–77, New York, NY, USA, 2004. IEEE Computer Society.

[34] K. Welsh and P. Sawyer. Understanding the scope of uncertainty in dynamically adaptive systems. In *Proc. of the Sixteenth International Working Conference on Requirements Engineering: Foundation for Software Quality*, volume 6182, pages 2–16. Springer, 2010.

[35] 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*, pages 264–274. IEEE, 1997.