An Empirical Analysis of the Mutation Operator for Run-Time Adaptive Testing in Self-Adaptive Systems

Erik M. Fredericks
Oakland University
Rochester, Michigan
fredericks@oakland.edu

ABSTRACT

A self-adaptive system (SAS) can reconfigure at run time in response to uncertainty and/or adversity to continually deliver an acceptable level of service. An SAS can experience uncertainty during execution in terms of environmental conditions for which it was not explicitly designed as well as unanticipated combinations of system parameters that result from a self-reconfiguration or misunderstood requirements. Run-time testing provides assurance that an SAS continually behaves as it was designed even as the system reconfigures and the environment changes. Moreover, introducing adaptive capabilities via lightweight evolutionary algorithms into a run-time testing framework can enable an SAS to effectively update its test cases in response to uncertainty alongside the SAS's adaptation engine while still maintaining assurance that requirements are being satisfied. However, the impact of the evolutionary parameters that configure the search process for run-time testing may have a significant impact on test results. Therefore, this paper provides an empirical study that focuses on the mutation parameter that guides online evolution as applied to a run-time testing framework, in the context of an SAS.

CCS CONCEPTS

• **Social and professional topics** → **Software selection and adaptation**; • **Theory of computation** → **Online algorithms**; • **Mathematics of computing** → **Evolutionary algorithms**; • **Software and its engineering** → **Software testing and debugging**; *Empirical software validation*;

KEYWORDS

search-based software testing, run-time testing, evolutionary algorithms, self-adaptive systems

ACM Reference Format:
Erik M. Fredericks. 2018. An Empirical Analysis of the Mutation Operator for Run-Time Adaptive Testing in Self-Adaptive Systems. In *SBST'18: SBST'18:IEEE/ACM 11th International Workshop on Search-Based Software Testing , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3194718.3194726

SBST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5741-8/18/05…$15.00
https://doi.org/10.1145/3194718.3194726

1 INTRODUCTION

Self-adaptive systems (SAS) can self-reconfigure at run time to ensure continuing requirements satisficement (i.e. satisfied to some degree) [7, 31, 33] in response to uncertainty. Uncertainty can manifest in many different forms, including aleatory and epistemic [1], as well as known-unknowns and emergent behaviors [14, 15]. Such uncertainties can cause an SAS to react in possibly incorrect or unforeseen manners, therefore, providing assurance at run time is paramount to a system's success. An SAS must ensure that not only its high-level requirements are continually satisfied, but that any test cases performing *fine-grained* validation also are continually passing. In the context of an SAS, test cases must be adaptive in that they can reconfigure alongside the SAS [18, 19], possibly using lightweight evolutionary techniques [4, 9]. However, the parameters that guide such a search procedure, specifically the σ parameter that guides the mutation operator of an online evolutionary algorithm (EA), can significantly impact the outcome of the search procedure. This paper presents an empirical study that focuses on the impact that this key configurable value can have on run-time evolutionary adaptation of test cases in an SAS.

The number of possible states and configurations that an SAS may experience is generally impossible to fully enumerate by an engineer [7, 8, 33, 41], leading to techniques for automatically validating and verifying an SAS at design time [6, 16, 17, 36, 37, 39] and run time [22, 23, 36, 38, 43, 47]. Additionally, techniques have been developed for anticipating unexpected SAS and environmental configurations [39], however such techniques still may not fully examine the entire solution space of all possible configurations. Validating a system's behavior at run time, then, relies on a well-developed set of requirements and test cases that specify a system's behavior to the extent possible at design time. Previously, the MAPE-T framework [20] was proposed to align testing with an SAS's ability to self-reconfigure by applying adaptive principles to a run-time testing framework. Given that not all test cases may be optimized for all environments at design time, the (1+1)-ONLINE EA (a lightweight EA intended for run time) [4] can be used to provide search capabilities for new combinations of test case parameters. The (1+1)-ONLINE EA sacrifices search power for speed as it only considers two individuals per generation, in comparison to a more powerful genetic algorithm that may consider several hundreds of individuals per generation [27].

Given that a single parameter, σ, is responsible for the search procedure within the (1+1)-ONLINE EA, we examine how varying this parameter impacts the run-time testing process. Specifically, we present a sensitivity analysis on the search procedure in (1+1)-ONLINE EA, where σ dictates the relative difference between a parent and a child genome via mutation. To this end, we vary the

value of σ for two case studies in distinct application domains, selecting multiple values of σ both manually and automatically, using domain knowledge for manual selection and the stepwise adaptation of weights (SAW) heuristic [45] to more intelligently vary the mutation operator automatically. Each aspect of the sensitivity analysis will be used to demonstrate the effects of the σ parameter on run-time testing in terms of test case fitness.

To this end, we apply our sensitivity analysis to two case studies. The first case study is a remote data mirroring (RDM) application that has been provided by industrial collaborators [29, 30]. RDM is a technique for ensuring that data is always protected and available within a distributed network of servers and has been modeled as an SAS. The second case study models a smart vacuum system (SVS) that is tasked with cleaning a simulated room while maintaining safety and failsafe concerns. Each case study is subjected to multiple forms of uncertainty that can induce requirements violations and run-time test case failures. Moreover, each case study is instrumented with an adaptive testing framework that leverages the SAS's adaptation engine.

Experimental results suggest that, for both case studies, the value of σ does not significantly impact the results of run-time testing, even as the SAS reconfigures as a result of uncertainty. The remainder of this paper is structured as follows. Section 2 presents background information on each case study, run-time software testing, and run-time evolutionary algorithms. Section 3 details our approach for performing the analysis on σ. Section 4 presents our experimental setup and results. Finally, Section 5 summarizes this paper and outlines future work.

2 BACKGROUND

This section presents relevant background information on the RDM and SVS applications, run-time software testing, and run-time evolutionary algorithms. Note that, for presentation purposes, we do not provide a dedicated related work section and opt to include relevant related works in this section and throughout the paper.

2.1 Remote Data Mirroring

RDM is a network-based application that ensures data is made available and prevents data loss by disseminating copies (i.e., replicates) of messages to all servers (i.e., data mirrors) connected to the network [29, 30]. In this fashion, a user could access the nearest server when requesting data to provide faster response times, or when requesting data from an unavailable server, the RDM network can failover to a different server that is relatively close to the user. Moreover, data is protected from loss or damage in the RDM application as data recovery techniques can be triggered upon determination of data loss by reconstructing data from a different server.

The RDM application has been modeled as an SAS and can reconfigure in response to uncertainty [40]. Uncertainty can manifest in terms of dropped, delayed, or corrupted messages, network link failures, server failures, and sensor fuzz applied to server and link sensors. The RDM self-reconfigures in terms of changes to its network overlay to facilitate link recovery (e.g., changing from a grid topology to a completely-connected topology), changes to its data propagation protocols (e.g., changing from asynchronous to synchronous message transmission), and updates to server state (i.e., updating from actively to passively servicing message transactions).

2.2 Smart Vacuum System

An SVS is an autonomous robotic vacuum (similar to an iRobot Roomba[1]) that is tasked with cleaning a desired space, facilitated by monitoring sensors to select an appropriate path planning algorithm, manage power consumption, and mitigate safety concerns. Available sensors can include distance sensors to measure the distance between the SVS and other objects in the room, bumper sensors to detect collisions, cliff sensors to detect steps, and sensors embedded within the wheel and suction motors to accurately monitor SVS velocity and suction power, respectively. A central controller unit aggregates all incoming sensor data and performs an analysis to ensure that the SVS is operating safely and efficiently by selecting appropriate path and power moding strategies towards its goal of maximally cleaning the room.

The SVS is modeled as an adaptive system and can self-reconfigure via mode changes [2, 35], where a mode change is a common strategy for performing adaptations within embedded systems. A sample mode change can be a *reduced power* mode where the SVS limits power consumption from the wheel motors, effectively slowing the SVS. Such a strategy may enable the robot to "limp home" to a charging station in the event that battery levels are critically low. The SVS can experience uncertainty in terms of randomly instantiated obstacles (e.g., pets, liquid spills, downward steps, etc.), occluded or failing sensors, unexpected power drains, and the amount, location, and distribution of dirt within the room. To mitigate such uncertainties, the SVS can self-reconfigure, via mode changes, in terms of reduced power consumption modes, different types of pathfinding algorithms, and various measures for quickly and safely avoiding unexpected obstacles.

2.3 Run-Time Software Testing

While software testing is a relatively well-understood field [3, 25, 34] and is generally performed at design time, performing testing at *run time* introduces significant problems in terms of overhead to the running system and concerns that testing a live system may impact or influence its behavior. However, testing at run time can also ensure that the system is satisfying its goals during execution [3, 11]. To combat performance concerns, techniques such as record-and-replay [44] and multi-agent testing [36] have been introduced to offload testing activities to either a sandboxed environment or an additional agent with spare computing power.

As with software requirements [46], test cases can be defined as *invariant* or *non-invariant*. An invariant test case describes a critical concern, such as safety, that cannot fail at run time. If a failure occurs in an invariant test case, the system has experienced a severe fault that prohibits recovery. Conversely, a non-invariant test case can temporarily be considered as failing, however such a failure can be transient in nature and be mitigated by an SAS reconfiguration. For this paper, only non-invariant test cases can be adapted, as invariant test cases generally focus on some safety-critical task. This paper also uses the IEEE definition of test cases, where a test

[1] iRobot Roomba: http://www.irobot.com

case comprises an expected value and the conditions for which a pass/fail determination can be applied [28]. We focus mainly on functional testing (i.e., validation against a test specification) and regression testing (i.e. validation against a test specification following a system update/change) [3, 26].

Veritas. Veritas is a technique for adapting test cases at run time using the (1+1)-ONLINE EA in SASs [19]. Specifically, Veritas leverages the SAS's MAPE-K loop [31] to determine if a reconfiguration is necessary in response to changing operating contexts and then adapts test cases to better fit the new context while ensuring that safety/failsafe concerns are not violated. We describe the Veritas technique in greater detail in Section 3.1.

2.4 Run-Time Evolutionary Algorithms

Evolutionary algorithms (EA) are commonly used to efficiently navigate a prohibitively-large search space for solving optimization problems, with a common example being the genetic algorithm [27]. However, such algorithms often suffer from enormous overhead in terms of processing time and memory required to evaluate each candidate solution, as each solution must not only be encoded but also simulated/executed to determine the overall fitness for each candidate. To combat these difficulties, *run-time* EAs have been developed to provide lightweight search capabilities as the system executes. One such example is the (1+1)-ONLINE EA [4], an algorithm based on the (1+λ)-EA [12, 42]. The (1+1)-ONLINE EA sacrifies searching power for speed by only maintaining two genomes at any given time, one parent and one child. Search is facilitated by a mutation value σ that can be adapted to search locally or globally, resulting from analysis of each candidate's fitness value. If fitness has been determined to be in a state of *stagnation* (i.e., little to no change in fitness over a specified interval), then σ is updated to search more broadly. In this case, stagnation can indicate the presence of a local optima.

Each candidate solution is instantiated to measure its fitness value, with the better-performing candidate surviving and the worse-performing candidate being discarded. In this technique, a new candidate is created by mutating the genome of the winner according to σ. In terms of run-time testing, Table 2 presents an example of (1+1)-ONLINE EA's mutation as applied to an RDM test case that measures the expected *diffusion time* of a message across the network. In this table, the upper bound and lower bound are mutated by σ, where $\sigma = 2.0$ in this case. Note that σ mutates the boundaries within a pre-defined tolerance to ensure that the failure of this test case does not impact safety concerns. We discuss this table in greater detail in Section 3.1.

The (1+1)-ONLINE EA cannot exhaustively search the entire solution space as could a normal EA, however the ability to search at run time, in parallel to a system's normal execution tasks, facilitates online solving of optimization problems.

Stepwise Adaptation of Weights. SAW is a hyper-heuristic [5] for updating a weighting scheme in a linear-weighted sum, generally of a fitness function, to determine if different weighting schemes can better reflect operating conditions to yield an optimal fitness value [13, 45]. For instance, definition of a fitness function often relies on the domain knowledge of an engineer or is based on observed/calculated metrics, and as such, may not accurately reflect the ideal composition of the function.

SAW can be implemented either *offline* or *online*, where an offline SAW implementation adjusts fitness function weights following execution of an EA. An online SAW implementation, as used in this paper, dynamically adjusts fitness function weights at run time. For the purposes of this paper, we update the value of σ at run time using SAW in an online fashion [45]. In this case, the value of σ is selected to be as diverse as possible as we are optimizing a single value.

For example, a fitness function may comprise three objectives that are each measured using separate functions, each of which is weighted to reflect its individual importance to the aggregate fitness function. As such, the weights initially selected by the engineer may not be optimal in all situations, and as a result, an automated technique such as SAW can examine fitness results over time and then automatically update weights at run time. SAW can select one objective to be considered as "more important" based on monitored conditions, thereby increasing its weight. SAW would then normalize the remaining weights such that the sum of all weights equals 1.0, thereby lessening the contribution of other objectives to the fitness function while still maintaining their input. SAW follows this heuristic until the fitness stagnates or program execution ends.

Sensitivity Analysis. A sensitivity analysis is a technique for determining the relative impact of a parameter or set of parameters on a system under test [32]. While many techniques exist for performing sensitivity analyses [24], we opt to vary the values of the parameter in question based on a range of values that a test engineer would select resulting from domain knowledge. Additionally, we include random value selection as well as a hyperheuristic (i.e., SAW) for selecting values to test.

3 APPROACH

This section describes our approach for examining the impact of the mutation operator σ on the (1+1)-ONLINE EA as implemented within the Veritas run-time adaptive testing framework.

3.1 Run-Time Adaptive Testing

Veritas [19] is a run-time technique for providing assurance in an SAS via online, evolutionary testing. Specifically, Veritas leverages the (1+1)-ONLINE EA [4] to explore how different configurations of test case parameters (i.e., lower bound, upper bound, and expected value) can represent the continually-changing space in which an SAS resides. Figure 1 presents an overview of the Veritas technique. Veritas is executed each timestep, or as often as the SAS engineer desires. Veritas takes as input a set of utility functions that capture the run-time performance of software requirements [10, 21] and are used to validate test results. During each testing cycle, (1) Veritas uses the SAS's monitoring framework to determine in which operating context the system is executing. Next, (2) Veritas selects a set of test cases that are impacted by the operating context to be adapted at run time. Following, (3) Veritas executes a testing cycle and (4) monitors test results (i.e., which tests pass or fail). Veritas will then (5) adapt the appropriate attributes of each failing test case. Steps (1) – (5) are supported by the (1+1)-ONLINE EA, as denoted by the evolutionary loop. Upon completion, (6) Veritas

updates the test specification with the test case parameter values that resulted from online optimization.

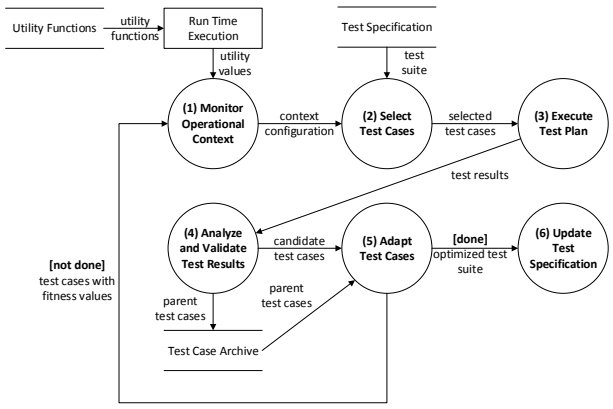

Figure 1: Data flow diagram of Veritas [19].

For example, consider a test case (TC_{62}) that was derived to ensure that the SVS executes a specified path plan for 30 seconds. In this case, the *expected value* is 30 *sec*, however an upper and lower bound of ± 1.0*sec* have been defined to provide flexibility in the case of encountered uncertainties. As such, a measured execution time for this particular path of 30.5 *sec* would be accepted as a *passing* test case, whereas a value of 31.0 *sec* would be considered a failure. Moreover, a *safety threshold* of ± 5.0*sec* has also been specified to restrict this test case from adapting beyond understood critical boundaries. Veritas would then be given the flexibility to explore a lower threshold, upper threshold, and expected value within the specified safety tolerance (i.e., 30 *sec* ± 5.0 *sec*).

During the course of SAS execution, Veritas examines the results of each test case to determine if an adaptation is necessary as a result of SAS reconfiguration. By correlating online testing with requirements monitoring (i.e., ensuring that requirement utility functions reflect the results of test cases), Veritas can determine if a test case adaptation is warranted. For example, Veritas may determine that a test case has failed, while requirements that were linked to this particular test case are satisfied. In this case, Veritas would determine that a *false positive* resulted from testing and prevent online adaptation. However, in the case of a *true positive* (i.e., both test case and correlated requirement(s) report failure/violation), Veritas would determine that the test case requires adaptation. Upon this determination, Veritas applies a mutation operator σ to search either the global search space (i.e., broad) or local search space (i.e., fine-grained) by mutating the boundary elements of each affected test case, within pre-defined safety tolerances. Given that Veritas leverages the (1+1)-ONLINE EA, only two concurrent instances of each test case exist (in comparison to a more standard type of EA in which many different solutions exist in memory).

To this end, we define a test case to be either an exact value (i.e., a passing test case must match the expected value) or a range of values (i.e., a passing test case must fall within a range, with the expected value denoting a perfect match) as shown in Table 1. Each type of test case comprises an *expected value* (*exp*), a *lower* and *upper* safety threshold (*lsb*, *usb*), and an additional *lower* and *upper* bound (*lb*, *ub*) to denote the acceptable range of values for a ranged test case. For an exact test case, the expected value can be mutated within the lower and upper bounds. However, the expected value, lower, and upper bounds cannot be mutated outside of the safety thresholds specified for each test case. For example, a ranged test case may be defined as $TC_i = \langle\ lsb = 1.0, lb = 3.0, exp = 4.0, ub = 5.0, usb = 7.0\ \rangle$

Parameter	Adaptive	Symbol
Lower bound	**Yes**	*lb*
Upper bound	**Yes**	*ub*
Expected value	**Yes**	*exp*
Lower safety threshold	No	*lsb*
Upper safety threshold	No	*usb*

Table 1: Test case definition.

The σ mutation operator can be applied to both types of test cases. For an exact test case, σ is applied to the *expected value* parameter by selecting a new random value between the *lower* and *upper bounds*, with each bound divided by σ. For instance, a test case that measures the diffusion time for messages in the RDM application may be expecting that a message is diffused within 8.0*sec*. As such, the test engineer has defined TC_7 according to Equation 1:

$$TC_7 = \langle lsb = 3.0sec, lb = 6.0sec, exp = 8.0sec, ub = 10.0sec, usb = 13.0sec \rangle \quad (1)$$

To demonstrate mutation, Equation 1 is expanded in Table 2 with a configured $\sigma = 2.0$. Mutation of the *expected value* selects a random value between $[lsb = 6.0/\sigma, usb = 10.0/\sigma]$ and then reassigns the *expected value* of TC_7 to be 4.4*sec*. The new test case TC_7' would then temporarily replace TC_7 during the next iteration of the testing procedure to determine if TC_7' better reflects the environment. Note that Veritas does not adapt test cases to simply pass, but rather adapts test cases to adequately reflect the operating context.

Field	Value	Mutated Value ($\sigma = 2.0$)
Lower bound	6.0 *sec*	6.0 *sec* $/\sigma$ = 3.0 *sec*
Upper bound	10.0 *sec*	10.0 *sec* $/\sigma$ = 5.0*sec*
Lower safety threshold	3.0 *sec*	*NA*
Upper safety threshold	13.0 *sec*	*NA*
Expected value	8.0 *sec*	randFloat([$bound_{low}$, $bound_{high}$]) = 4.4*sec*

Table 2: Mutation of test case with (1+1)-ONLINE EA.

To quantify the performance of an adaptive test case, *test case fitness* has been defined to capture its relevance to its operating context. Specifically, multiple fitness subfunctions were defined to capture the performance of each type of test case. Table 3 presents the fitness subfunctions previously defined by Fredericks *et al.* [19].[2]

[2]For presentation purposes, we abbreviate *measured* as *M*, *expected* as *E*, *optimal* as *O*, *value* as val, and fitness function as ff.

Type	Fitness subfunction
Invariant - Exact	$if(val_M == TC_E) : ff_M = 1.0$ $else : ff_M = 0.0$
Invariant - Range	$if(val_M \in [val_{lb}, val_{ub}]) : ff_M = 1.0$ $else : ff_M = 0.0$
Non-invariant - Exact	$ff_M = 1.0 - \frac{\lvert val_M - val_E \rvert}{\lvert val_E \rvert}$
Non-invariant - Range	$if(val_M \in [val_{lb}, val_{ub}]) : ff_M = 1.0$ $else : ff_M = 1.0 - \frac{\lvert val_M - val_O \rvert}{\lvert val_O \rvert}$

Table 3: Test case fitness subfunctions.

In the case of non-invariant ranged test cases, an optimal test case value is specified to be the nearest acceptable range boundary to the measured value. Consider TC_7 as defined in Equation 1. If the measured value (i.e., val_M) is $6.2sec$, then the optimal test case value (i.e., val_O) will be its nearest boundary lb, or $6.0sec$.[3]

Lastly, the overall fitness value is calculated for each test case, comprising a weighted linear sum as defined in Equation 2:

$$ff_{test\_case} = \alpha_M * ff_M + \alpha_V * ValidResult \tag{2}$$

where:

$$ValidResult = \begin{cases} 1.0 & if\ val_M \in [val_{lsb}, val_{usb}], \\ 0.0 & else. \end{cases} \tag{3}$$

3.2 Stepwise Adaptation of Weights

In addition to static values of α that will be evaluated, we introduce the hyper-heuristic SAW into Veritas to intelligently adapt σ at run time. Specifically, our implementation of the online variation of SAW [45] is as follows in Algorithm 1:

Algorithm 1 Online SAW implementation for RDM application.

```
1:  timesteps ← 300
2:  test_cases ← instantiateTestCases()
3:  for (i = 0; i < timesteps; ++i) do
4:      Execute RDM at i^th timestep
5:      Execute SAS adaptation engine
6:      Execute Veritas
7:      ts ← i mod 10
8:      if ts = 0 then % Execute SAW
9:          Retrieve all prior values of σ
10:         Calculate mean of σ prior values
11:         Generate random set of candidate σ values
12:         Select σ with maximum distance from mean
13:         Apply new σ to Veritas-selected test cases
14:     end if
15: end for
```

Algorithm 1 demonstrates that SAW is triggered every 10^{th} timestep to update the value of σ used by Veritas in the following execution cycles. Here, SAW retrieves *all* prior values of σ up to the current timestep and calculates the mean of these values. Next, SAW generates a random pool of σ candidates, bounded within a safety range specified by the test engineer, and then selects the candidate value that maximizes the distance from the mean of all prior σ values. In this regard, SAW is maximizing the explored *search space* governed by σ. The SAW implementation for the SVS application follows the same algorithm, however the number of timesteps is configured to be 120.

[3]Note that this is considering the non-mutated form of TC_7.

4 EXPERIMENTAL RESULTS

This section describes our experimental setup and results from investing how σ impacts the run-time evolutionary search process on two separate application domains: the RDM and SVS applications.

4.1 Experimental Setup

For this paper, we focus on the impact of the mutation parameter σ within the (1+1)-ONLINE EA as applied to run-time adaptive testing for SASs. We have implemented Veritas [19] on top of the MAPE-K loop [31] that guides adaptation of the RDM and SVS applications, respectively, where Veritas leverages the (1+1)-ONLINE EA as previously introduced by Bredeche *et al.* [4].

The RDM application was simulated as a completely-connected graph, where each node in the graph represents a server (i.e., data mirror) and each edge represents a network link between servers. Uncertainty was simulated to manifest via randomly inserted messages for dissemination at any point during execution, unexpected network link failures, random noise applied to both server sensors and network traffic, and randomly dropped and/or delayed messages. To mitigate uncertainty, the RDM application can self-reconfigure in terms of its network topology, methods of propagating messages, and server state (e.g., from actively servicing transactions to refusing to service transactions). The RDM application was executed for 300 timesteps during which all servers must receive a copy of all messages inserted into the network.

The SVS application was simulated as an autonomous robotic vacuum that comprises a set of sensors, each of which interact with the environment or SVS itself. Available sensors include bumper sensors that detect collisions, an object sensor to measure the distance between the SVS and nearby objects (either stationary or in motion), cliff sensors to detect stairs, and sensors instrumented within the wheel and suction motors. Each sensor has a probability of fuzz and failure, where fuzz occludes the sensor's readings, and failure causes the sensor to cease function for the remainder of execution. The SVS also has a central controller for handling sensor input and making decisions based on its understanding of the environment. Moreover, the controller is also responsible for providing MAPE-K capabilities in terms of self-reconfiguration, where a reconfiguration may be triggered by unsafe conditions (e.g., a detected step or pool of water) or objects (e.g., a pet or child) that must be quickly avoided by the SVS. The SVS was executed for 120 simulated timesteps and was required to vacuum at minimum 50% of the dirt particles within the room. The SVS experiences environmental uncertainty in terms of the amount, location, and size of dirt particles; the width and height of the room; the appearance and location of a downward step; and instantiated objects (e.g., circular liquid spill, columns, pets) that must be navigated safely around by the SVS. System uncertainty was represented by occluded and/or

failing sensors that could be triggered at each timestep, based on the defined probability of occlusion/failure for each sensor. For each experimental treatment, the SVS was placed in 15 unique configurations of system and environmental parameters as generated by Loki [39], a technique for generating diverse combinations of system and environmental parameters.

The RDM test specification comprises 36 test cases, where 7 are invariant and 29 are non-invariant [18]. The SVS test specification comprises 72 test cases, 17 of which are invariant and 55 of which are non-invariant. As defined by Veritas, only non-invariant test cases can be adapted at run time to ensure that safety/failsafe concerns are continually satisfied. The fitness function weights (α_M and α_{valid}) for both the RDM and SVS were set to 0.4 and 0.6, respectively [19].

For both the RDM and SVS applications, the online version of SAW has been implemented for varying σ, as shown in Algorithm 1. SAW was triggered every 10 timesteps, where SAW examines the previous states of σ up to the current time and selects a new value of σ to maximize its distance from all prior instantiations of σ.[4]

Table 4 presents the different values of σ that were selected for study. Specifically, the value of σ dictates the upper and lower bounds that can be randomly selected for mutating a test case's acceptable range of values, where the expected value of the test case is then modified to be randomly generated between the new bounds. Each mutation introduced is constrained to not violate any defined safety thresholds as previously specified by a test engineer. For each case study, we examine seven σ values, where σ_{1-5} use a varying static value, σ_6 introduces randomness to achieve diversity at each timestep of execution, and σ_7 uses the SAW heuristic to intelligently update this value. A control in which no adaptation (i.e., Veritas was disabled) was also performed to provide a basis for comparison. For both the RDM and SVS applications, 50 experimental replicates were performed for each value of σ to achieve statistical significance. In addition to randomly varying the configuration of system and environmental parameters, a random distribution was also selected for each replicate to seed the random number generator. Possible distributions include [*Beta*, *Binomial*, *ChiSquare*, *Exponential*, *Gamma*, *Geometric*, *Gaussian*, *Poisson*, *Triangular*, *Uniform*].

σ_{ID}	**Value**
σ_1	1.0
σ_2	2.0
σ_3	4.0
σ_4	8.0
σ_5	12.0
σ_6	$randFloat([1.0, 12.0])$ *per timestep*
σ_7	SAW

Table 4: Tested values of σ.

[4]As there is only a single instance of σ at any given point, we opted to examine diversity over performance in terms of the SAW algorithm.

4.2 Experimental Results

We now present the results of each case study. For presentation purposes, results for both case studies will be demonstrated together. To determine statistical significance, we performed a one-way ANOVA test ($p < 0.05$) to determine if a significant difference exists between data sets, as well as Wilcoxon-Mann-Whitney u-tests in a pairwise fashion ($p < 0.05$) to analyze the data more closely. While Veritas has been previously presented to significantly increase adaptive test case fitness over non-adaptive test case fitness [19], we also include non-adaptive test case fitness values as a control. To this end, we define the following null hypothesis $H0$ to state that "there exists no significant difference between different values of σ for adaptive testing with the (1+1)-ONLINE EA" and the alternate hypothesis $H1$ to state that "there is a significant difference between different values of σ for adaptive testing with the (1+1)-ONLINE EA."

Figure 2 presents the average test case fitness values from a sensitivity analysis of σ on Veritas in the RDM and SVS applications, with RDM values on the left of the separator and SVS values on the right. As this figure indicates, there exists a significant difference between the Control and Veritas experiments for both the RDM and SVS applications ($p < 0.05$, ANOVA and pairwise Wilcoxon-Mann-Whitney u-tests), confirming prior results that performing adaptation on run-time testing provides a positive impact on test case fitness [19]. However, no significant difference exists between each experiment where σ is varied ($p < 0.05$, ANOVA and pairwise Wilcoxon-Mann-Whitney u-tests). This result is surprising, as σ provides the only means of search within the (1+1)-ONLINE EA and was expected to influence fitness results. Furthermore, the SVS fitness values are significantly lower than those of the RDM overall, however that is related to the performance of the SVS, with test case values indicating that the SVS is performing relatively poorly. At minimum, Veritas still significantly enhances test case performance.

While these results suggest that we can accept the null hypothesis $H0$ in that varying σ has no impact on run-time adaptive testing, the implication that varying a mutation parameter has minimal impact is concerning. As previously shown by Bredeche *et al.*, varying σ can have a significant impact on a robot controller, where the EA searches for configurations of a neural network [4]. Therefore, the remaining conclusion to draw is that the impact of σ in Veritas specifically is limited. While performing adaptive testing significantly improves test case fitness [19], there must be little variation in the test case parameter values discovered for each operating context. Therefore, the important aspect of Veritas must be *adapting to the new context*, rather than minutely examining the search space of test case parameters. Moreover, test cases are extremely fine-grained by nature, and as a result the search space for a valid test case will also be fine-grained.

In terms of contributions and novelty, this paper demonstrates the effect that a relatively small search space imparts onto an SAS. Specifically for this context, each test case has a limited range of possible values that can be selected at run time as *valid*, resulting from safety constraints that are necessary to ensure that no invariant goals are violated. While there is little variation in discovery of "optimal" test case parameters, the fact that Veritas significantly performs better than the Control indicates that an online search

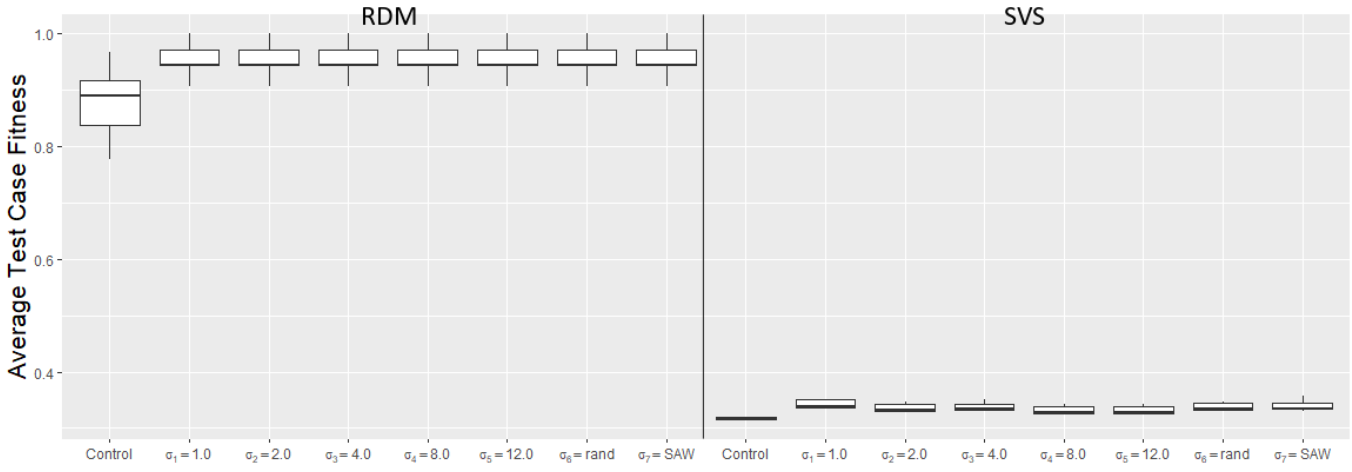

Figure 2: Comparison of average test case fitness values for RDM and SVS applications.

process is useful. This result suggests that, for an SAS, performing a lightweight search process at run time can enhance overall assurance that the system is behaving as intended. In terms of the non-significance between Veritas results, future work can explore more heavy-processing techniques such as multi-objective optimization or model-based testing, where processing tasks are offloaded to external agents.

Threats to validity. The research presented in this paper examined the importance of the mutation operator σ on the (1+1)-ONLINE EA, as implemented within Veritas. As such, we've identified the following threats to validity. One threat to validity lies in the derivation and validity of test cases for each case study. Another threat lies in the configuration and implementation of both the RDM and SVS applications. The test cases may also be too fine-grained/constrained for evolution to successfully discover global optima (i.e, a more diverse set of test cases may provide different results), resulting in a limited search space where test case parameters converge to similar values. Finally, the online version of SAW technically updates the fitness function at run time, leading to the possibility that the fitness function becomes too flexible, leading to solutions that stray from the original intent of each respective test case.

5 DISCUSSION

This paper has presented an empirical study on the impact that the mutation operator σ imparts onto a run-time EA that has been implemented within a run-time testing framework, where the run-time EA guides test case adaptation as uncertainty manifests within an SAS and its environment. Specifically, we examined σ in the context of the Veritas technique as applied to two case studies in different application domains: the RDM and SVS applications. The RDM application ensures that data is replicated across a network of physically-remote servers to ensure data availability and reliability, and the SVS application simulates an autonomous robotic vacuum that must clean a room while mitigating uncertainty and safety concerns. Veritas implements the (1+1)-ONLINE EA to search for combinations of test case parameters that more accurately reflect changing operating contexts. Experimental results suggest that, for each application domain, the role of σ has no significant impact on both the search process and test results. Future work includes further examination of σ in other application domains, including those that are not specifically test-oriented. Moreover, we intend to apply other types of run-time evolutionary techniques to the Veritas testing framework to determine the feasibility and result of other techniques.

ACKNOWLEDGEMENTS

This research has been supported in part by NSF grant CNS-1657061, the Michigan Space Grant Consortium, the Comcast Innovation Fund, and Oakland University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Oakland University or other research sponsors.

REFERENCES

[1] Jason Matthew Aughenbaugh. 2006. *Managing uncertainty in engineering design using imprecise probabilities and principles of information economics.* Ph.D. Dissertation.

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

[3] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering, 2007. FOSE '07*. 85–103.

[4] N. Bredeche, E. Haasdijk, and A.E. Eiben. 2010. On-Line, On-Board Evolution of Robot Controllers. In *Artificial Evolution*, Pierre Collet, Nicolas Monmarché, Pierrick Legrand, Marc Schoenauer, and Evelyne Lutton (Eds.). Lecture Notes in Computer Science, Vol. 5975. Springer Berlin Heidelberg, 110–121.

[5] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. 2003. Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In *Handbook of Metaheuristics*, Fred Glover and Gary A. Kochenberger (Eds.). Vol. 57. Springer US, 457–474.

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

[7] Betty H. C. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, and et al. 2009. Software engineering for self-adaptive systems: A research roadmap. In *Software engineering for self-adaptive systems*. Springer-Verlag, Berlin, Heidelberg, Chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, 1–26.

[8] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. 2009. 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*. Springer-Verlag, Berlin, Heidelberg, 468–483.

[9] Zack Coker, David Garlan, and Claire Le Goues. 2015. SASS: Self-adaptation using stochastic search. In *Proceedings 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*. IEEE, 168–174.

[10] Paul deGrandis and Giuseppe Valetto. 2009. Elicitation and Utilization of Application-level Utility Functions. In *Proc. of the 6th International Conference on Autonomic Computing (ICAC '09)*. ACM, 107–116.

[11] N. Delgado, A.Q. Gates, and S. Roach. 2004. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering* 30, 12 (2004), 859–872.

[12] Stefan Droste, Thomas Jansen, and Ingo Wegener. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science* 276, 1 (2002), 51–81.

[13] A.E. Eiben and J. K. van der Hauw. 1998. Adaptive penalties for evolutionary graph coloring. In *Artifical Evolution*. Springer.

[14] N. Esfahani. 2011. A framework for managing uncertainty in self-adaptive software systems. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 646–650.

[15] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. 2011. Taming uncertainty in self-adaptive software. In *Proceedings. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 234–244. https://doi.org/10.1145/2025113.2025147

[16] Yliès Falcone, Mohamad Jaber, Thanh-Hung Nguyen, Marius Bozga, and Saddek Bensalem. 2011. Runtime verification of component-based systems. In *Proc. of the 9th international conference on Software engineering and formal methods*. Springer-Verlag, Berlin, Heidelberg, 204–220.

[17] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. 2012. A formal approach to adaptive software: continuous assurance of non-functional requirements. *Formal Aspects of Computing* 24 (2012), 163–186. Issue 2.

[18] Erik M. Fredericks and Betty H. C. Cheng. 2015. Automated Generation of Adaptive Test Plans for Self-Adaptive Systems. In *Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '15)*.

[19] Erik M. Fredericks, Byron DeVries, and Betty H. C. Cheng. 2014. Towards Runtime 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 (SEAMS '14)*.

[20] Erik M. Fredericks, Andres J. Ramirez, and Betty H. C. Cheng. 2013. 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)*. IEEE Press, 169–174.

[21] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. 2004. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (2004), 46–54.

[22] C. Ghezzi. 2010. Adaptive Software Needs Continuous Verification. In *Software Engineering and Formal Methods (SEFM), 2010 8th IEEE International Conference on*. 3 –4.

[23] Heather J. Goldsby, Betty H. C. Cheng, and Ji Zhang. 2008. Models in Software Engineering. In *Models in Software Engineering*, Holger Giese (Ed.). Springer-Verlag, Berlin, Heidelberg, Chapter AMOEBA-RT: Run-Time Verification of Adaptive Software, 212–224.

[24] Amir Hakami, M. Talat Odman, and Armistead G. Russell. 2003. High-Order, Direct Sensitivity Analysis of Multidimensional Air Quality Models. *Environmental Science & Technology* 37, 11 (2003), 2442–2452.

[25] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2009. 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).

[26] Mark Harman, Phil McMinn, Jerffeson Teixeira Souza, and Shin Yoo. 2012. Search Based Software Engineering: Techniques, Taxonomy, Tutorial. In *Empirical Software Engineering and Verification*. Lecture Notes in Computer Science, Vol. 7007. Springer Berlin Heidelberg, 1–59.

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

[28] IEEE. 2010. Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)* (Dec 2010), 1–418.

[29] Minwen Ji, Alistair Veitch, and John Wilkes. 2003. Seneca: Remote mirroring done write. In *USENIX 2003 Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 253–268.

[30] Kimberly Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. 2004. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*. USENIX Association, Berkeley, CA, USA, 59–62.

[31] J.O. Kephart and D.M. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (January 2003), 41 – 50.

[32] Barbara A Kitchenham, Shari Lawrence Pfleeger, Lesley M Pickard, Peter W Jones, David C. Hoaglin, Khaled El Emam, and Jarrett Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on software engineering* 28, 8 (2002), 721–734.

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

[34] Glenford J Myers, Corey Sandler, and Tom Badgett. 2011. *The art of software testing*. John Wiley & Sons.

[35] Sandeep Neema, Ted Bapty, and Jason Scott. 1999. Development environment for dynamically reconfigurable embedded systems. In *Proceedings of the International Conference on Signal Processing Applications and Technology. Orlando, FL*.

[36] Cu D. Nguyen, Anna Perini, Paolo Tonella, and Fondazione Bruno Kessler. 2007. Automated Continuous Testing of MultiAgent Systems. In *The Fifth European Workshop on Multi-Agent Systems (EUMAS)*.

[37] Duy Cu Nguyen, Anna Perini, and Paolo Tonella. 2008. A goal-oriented software testing methodology. In *Proc. of the 8th international conference on Agent-oriented software engineering VIII*. Springer-Verlag, Berlin, Heidelberg, 58–72.

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

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

[40] Andres J. Ramirez, David B. Knoester, Betty H. C. Cheng, and Philip K. McKinley. 2009. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th International Conference on Autonomic Computing*. 97–106. (Best paper award).

[41] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. 2010. Requirements-Aware Systems: A Research Agenda for RE for Self-adaptive Systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*. 95 –103.

[42] Hans-Paul Schwefel. 1981. *Numerical optimization of computer models*. John Wiley & Sons, Inc.

[43] Gabriel Tamura, NorhaM. Villegas, HausiA. Müller, JoãoPedro Sousa, Basil Becker, Gabor Karsai, Serge Mankovskii, Mauro Pezzè, Wilhelm Schäfer, Ladan Tahvildari, and Kenny Wong. 2013. Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems. In *Software Engineering for Self-Adaptive Systems II*. Lecture Notes in Computer Science, Vol. 7475. Springer Berlin Heidelberg, 108–132.

[44] J.J.-P. Tsai, K.-Y. Fang, Horng-Yuan Chen, and Yao-Dong Bi. 1990. A noninterference monitoring and replay mechanism for real-time software testing and debugging. *Software Engineering, IEEE Transactions on* 16, 8 (1990), 897–916.

[45] van der Hauw K. 1996. *Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems*. Master's thesis. Leiden University.

[46] Axel van Lamsweerde. 2009. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley.

[47] O. Wei, A. Gurfinkel, and M. Chechik. 2011. On the consistency, expressiveness, and precision of partial modeling formalisms. *Information and Computation* 209, 1 (2011), 20–47.