2016 IEEE/ACM 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems

Automatically Hardening a Self-Adaptive System against Uncertainty

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

ABSTRACT

A self-adaptive system (SAS) can reconfigure to adapt to potentially adverse conditions that can manifest in the environment at run time. However, the SAS may not have been explicitly developed with such conditions in mind, thereby requiring additional configuration states or updates to the requirements specification for the SAS to provide assurance that it continually satisfies its requirements and delivers acceptable behavior. By discovering both adverse environmental conditions and the SAS configuration states that can mitigate those conditions at design time, an SAS can be hardened against uncertainty prior to deployment, effectively extending its lifetime. This paper introduces two search-based techniques, Ragnarok and Valkyrie, for hardening an SAS against uncertainty. Ragnarok automatically discovers adverse conditions that negatively impact an SAS by searching for environmental conditions that explicitly cause requirements violations. Valkyrie then searches for SAS configurations that improve requirements satisficement throughout execution in response to discovered adverse environmental conditions. Together, these techniques can be used to improve the design and implementation of an SAS. We apply each technique to an industry-provided remote data mirroring application that can self-reconfigure in response to unknown or adverse conditions, such as network message delays, network link failures, and sensor noise.

CCS Concepts

•Social and professional topics → Software selection and adaptation; •Computer systems organization → Reconfigurable computing; •Software and its engineering → Search-based software engineering;

Keywords

search-based software engineering, self-adaptive systems, genetic algorithms, software assurance, requirements engineering

SEAMS'16, May 16-17, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4187-5/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2897053.2897059

1. INTRODUCTION

Self-adaptive systems (SAS) generally comprise an adaptation engine and a set of known configurations that specify application parameter values, where the configurations ensure that the application continually satisfies its requirements at run time. Moreover, an SAS can self-reconfigure at run-time to switch between different configurations in order to respond to changes in the environment. Uncertainty can impact an SAS by presenting environmental conditions unanticipated by the engineer [2, 9, 36], causing the SAS to enter a configuration state previously unknown or untested. This paper discusses two complementary approaches, the first of which enables the automated discovery of adverse SAS environmental conditions, and the second of which automatically discovers SAS configurations that can be used to combat those conditions.

Depending on the complexity of an SAS, it may be impossible for an engineer to fully enumerate all possible configurations and environments that an SAS may both use and experience, respectively [3, 4, 28, 32]. To this end, techniques [13, 14] have been previously introduced to enable the SAS to mitigate uncertainty. Other techniques have been developed to automatically induce failures in an SAS [1, 24, 25] for code coverage testing. Furthermore, self-healing systems have been developed to automatically repair themselves upon identification of faults or error conditions [6, 16, 17]. As such, additional automated techniques are necessary to enable the identification of adverse conditions at design time to assist in minimizing the uncertainty surrounding an SAS. Moreover, automated discovery of mitigation strategies for handling such adversity are also necessary. Together, discovery of adverse conditions and mitigation strategies can be used to harden an SAS in terms of its requirements specification or set of configuration states.

Given the enormous possible solution space of different combinations of system and environmental parameters, and moreover the unexpected interactions that can occur as a result of such combinations, an automated approach is necessary for discovering both adverse conditions and configurations of SAS parameters to handle adverse conditions. This paper introduces two complementary techniques: Ragnarok and Valkyrie. Ragnarok enables the automatic identification of environmental conditions that can otherwise break an SAS. Conversely, Valkyrie automatically identifies SAS configurations that can be used to mitigate those adverse conditions identified by Ragnarok. When used in tandem, Ragnarok and Valkyrie can harden an SAS against uncertainty.

Ragnarok leverages evolutionary search to discover environmental conditions that explicitly violate system requirements. Specifically, Ragnarok searches for combinations of environmental parameters that cause an SAS to violate both its *invariant* and *non-invariant* requirements, where a higher priority is placed upon violating invariant requirements. Monitoring of requirements satisficement requires a set of derived utility functions to quantify run-time performance [7, 35]. For example, consider a robotic vacuum that must clean a room. An adverse environment in this example might include puddles of water that had been spilled, where vacuuming the water could damage the internal mechanisms of the robot. As such, damage to the robot would violate invariant safety requirements. To rectify the violated requirements and ensure that the SAS continues to deliver acceptable behavior while experiencing environmental uncertainty, Valkyrie uses evolutionary search to discover combinations of system parameters that enable the SAS to satisfy its requirements in the adverse environments that Ragnarok previously discovered. Continuing the robotic vacuum example, Valkyrie could generate configurations wherein the robot has an optimally-configured path planning algorithm to avoid the puddles of water. Together, Ragnarok and Valkyrie can be used to identify weaknesses in software requirements with respect to environmental uncertainty and automatically discover SAS configurations that mitigate such uncertainties.

Ragnarok and Valkyrie are each demonstrated through application to a simulated remote data mirroring (RDM) network [21, 22]. The RDM network must replicate and distribute data to all data mirrors (i.e., servers) connected to the network. The RDM network can also experience uncertainty in terms of network link failures, dropped or delayed messages, and non-trustworthy sensor data. Experimental results suggest that Ragnarok can discover environmental conditions that induce significantly more invariant and non-invariant requirements violations when compared to randomly-generated environmental conditions. Furthermore, additional results suggest that Valkyrie can discover SAS configurations that significantly reduce the amount of invariant and non-invariant requirements violations when compared to both previously-known and randomized SAS configurations, as well as increase the overall level of requirements satisficement.

The remainder of this paper is structured as follows. Section 2 discusses background information on the RDM application, goal-oriented requirements engineering, and genetic algorithms. Section 3 then discusses both the Ragnarok and Valkyrie techniques in detail. Section 4 overviews and presents our experimental setups and results, respectively. Section 5 overviews related work, and Section 6 discusses the findings presented in this paper and presents future directions for this research.

2. BACKGROUND

This section overviews background material on the RDM network application, goal-oriented requirements engineering, and genetic algorithms.

2.1 Remote Data Mirroring Application

Remote data mirroring (RDM) [21, 22] is a data protection technique that is used for preventing data loss and ensuring data availability. This technique is enabled by distributing *data replicates* to servers in physically remote locations (i.e., data mirrors). An RDM can be modeled as an SAS [31] and is configurable in terms of its network topology (e.g., minimum spanning tree, redundant topology) and data propagation parameters (e.g, method and timing of data distribution). There are two methods of data distribution that are supported. *Synchronous distribution* automatically distributes data modifications to all other data mirrors, while *asynchronous distribution* batches data modifications to combine edits made to the data. Asynchronous propagation provides better network performance, however data can be lost if a data mirror fails. Conversely, synchronous distribution provides better data protection at the expense of network performance.

The RDM can respond to uncertainty by performing self-reconfigurations, where uncertainty can include unexpected network link failures, randomly dropped or delayed messages, and noise that can affect link and data mirror sensors. Each network link also incurs a cost that can have an impact on the overall budget, and moreover has a measurable latency, throughput, and loss rate. These metrics, combined with the ability to replicate all messages to all data mirrors, determine the overall performance of the RDM. Reconfiguration of the RDM includes updating the network topology or data propagation parameters. Specifically, the RDM can self-reconfigure by updating its network topology by activating and deactivating network links. For example, the RDM can update its topology from a minimum spanning tree to a redundant topology. Furthermore, the RDM can change its data propagation protocols on each data mirror between asynchronous and synchronous propagation. Given its ability to reconfigure, the RDM can be modeled as an SAS.

For example, Figure 1 presents a sample topology of the RDM network following exposure to uncertainty. In this figure, three network partitions exist due to network link failures (i.e., links between Data Mirrors (5) and (9), and Data Mirrors (18) and (23)). Network partitions can cause data distribution to fail as no valid route exists between data mirrors on different partitions. As a result, the RDM network can self-reconfigure its topology by activating other network links. For instance, a link between Data Mirror (9) and Data Mirror (19) could be activated, thereby removing the first partition that exists. Likewise, a network link between Data Mirrors (14) and (15) could be activated, removing the second partition and therefore enabling full data replication among all connected data mirrors.

2.2 Goal-Oriented Requirements Modeling

Goal-oriented requirements modeling (GORE) is an approach for specifying objectives and constraints that a system must provide and satisfy, respectively, to guide the elicitation and analysis of system requirements in a goal-oriented fashion. The GORE process provides for different types of goals, including *functional*, *non-functional*, *safety*, and *failsafe* goals. Functional goals specify a service that must be provided to a system's stakeholders. Non-functional goals impose a quality constraint upon delivery of functional services. Safety goals specify a critical service that cannot be violated. Failsafe goals provide a safe fallback state in case of system failure [20]. Functional goals may additionally be categorized as invariant (i.e., must always be satisfied) or non-invariant (i.e., may be temporarily unsatisfied at run time). Invariant goals are denoted by the keywords *Maintain* or *Avoid*, and non-invariant goals are denoted by the

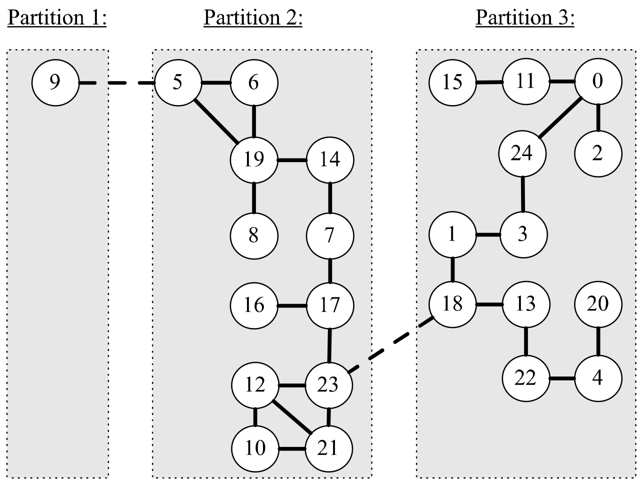


Figure 1: Sample partitioned RDM network.

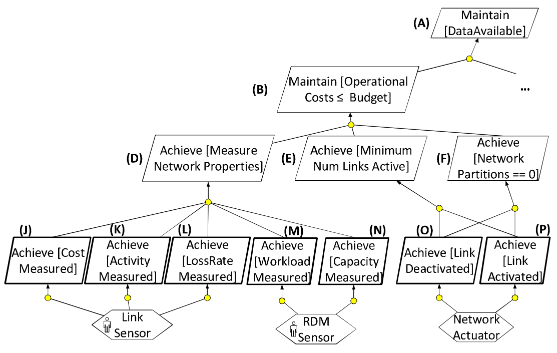
Achieve keyword. Moreover, safety and failsafe goals are always considered to be invariant.

GORE modeling decomposes high-level goals into subgoals using a directed, acyclic graph [33], where each node in the graph represents a goal and each edge represents a goal refinement. KAOS [5, 33] provides an approach for refining goals using AND and OR refinements. An AND-refined goal is satisfied if *all* its subgoals are also satisfied. An OR-refined goal is satisfied if at least *one* of its subgoals have been satisfied. Goal refinement is completed when an agent has been assigned responsibility for satisfying each leaf-level goal, where leaf-level goals are considered to be requirements.

Figure 2 presents a KAOS goal model of the RDM application, where Goals (A) and (B) are specified to be invariant, and all other goals are considered non-invariant. This figure presents the decomposition of goals, starting with a high-level goal of maintaining data availability (i.e., Goal (A)) and refining the model to the leaf-level (i.e., Goals (J) – (W)), where each leaf-level goal must be satisfied by agents (i.e., Link Sensor, Network Controller, etc.). An example AND-refinement is shown at Goal (B), where each of its subgoals (i.e., Goals (D), (E), and (F)) must be satisfied for Goal (B) to also be satisfied. An OR-refinement is shown at Goal (H), where at least one of its subgoals (i.e., Goals (S) and (T)) must be satisfied for it to be satisfied.

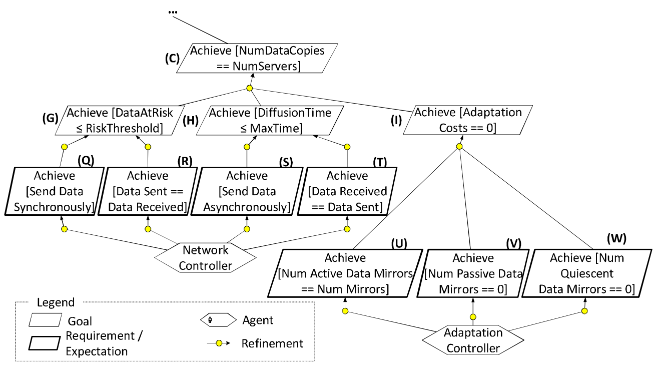
Utility functions.

Utility functions are mathematical formulae that can be used to quantify the level of satisfaction (i.e., satisficement) of software requirements or behaviors at run time in autonomic computing systems [7, 35]. Utility functions can also be derived for KAOS goals to determine run-time satisficement [30]. A utility value of 0.0 generally indicates a violation, and a value of 1.0 indicates satisfaction. Any value in between $(0.0, 1.0)$ indicates the degree of satisficement for that goal or requirement. For example, a utility function has been derived to quantify Figure 2, Goal (B). Goal (B) can be evaluated with a function that returns a value of 1.0 if the cost of operating the RDM network has never exceeded the provided budget, and a value of 0.0 otherwise.



(A)
Maintain [DataAvailable]
(B)
Maintain [Operational Costs ≤ Budget]
...
(D)
Achieve [Measure Network Properties]
(E)
Achieve [Minimum Num Links Active]
(F)
Achieve [Network Partitions == 0]
(J)
Achieve [Cost Measured]
(K)
Achieve [Activity Measured]
(L)
Achieve [LossRate Measured]
(M)
Achieve [Workload Measured]
(N)
Achieve [Capacity Measured]
(O)
Achieve [Link Deactivated]
(P)
Achieve [Link Activated]
Link Sensor
RDM Sensor
Network Actuator

(a) Left half of RDM goal model.



...
(C)
Achieve [NumDataCopies == NumServers]
(G)
Achieve [DataAtRisk ≤ RiskThreshold]
(H)
Achieve [DiffusionTime ≤ MaxTime]
(I)
Achieve [Adaptation Costs == 0]
(Q)
Achieve [Send Data Synchronously]
(R)
Achieve [Data Sent == Data Received]
(S)
Achieve [Send Data Asynchronously]
(T)
Achieve [Data Received == Data Sent]
Network Controller
(U)
Achieve [Num Active Data Mirrors == Num Mirrors]
(V)
Achieve [Num Passive Data Mirrors == 0]
(W)
Achieve [Num Quiescent Data Mirrors == 0]
Adaptation Controller
Legend
Goal
Requirement / Expectation
Agent
Refinement

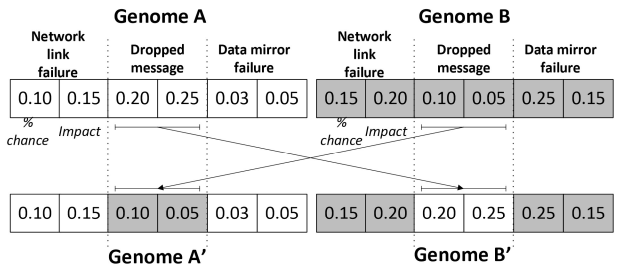
(b) Right half of RDM goal model.

Figure 2: KAOS goal model of RDM application.

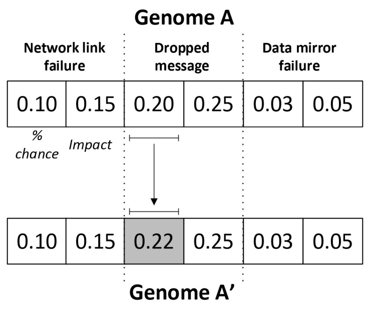
2.3 Genetic Algorithms

A genetic algorithm is a stochastic, search-based heuristic for exploring solutions for complex optimization problems [18]. A genetic algorithm typically comprises a *population*, a *fitness function*, and a set of *evolutionary operators* (e.g., crossover and mutation). A population refers to a set of encoded candidate solutions, where typical solutions can be encoded in a vector-based representation. A fitness function evaluates the quality of each candidate solution using a mathematical formula to guide the search process towards an optimal solution. Evolutionary operators generate new solutions, where the crossover operator exchanges parts of existing solutions to form new solutions, and a mutation operator randomly changes portions of an individual solution to maintain diversity. This procedure continues until a termination criterion (e.g., maximum number of generations to run) is reached and the solution with the best fitness value is provided as output.

For this paper, double-point crossover and single-point mutation have been selected as the evolutionary operators. Figure 3 demonstrates double-point crossover and single-point mutation, respectively, on a Ragnarok genome. In double-point crossover, two indices are randomly selected on the genome to determine which set of genes (i.e., parameters) are swapped between the two randomly selected parent genomes. Crossover then creates two new child genomes based on shared genes from the two parents. In single-point mutation, a gene within the genome is randomly selected for mutation, where the particular value is then mutated. Figure 3(a) presents an example of double-point crossover, where the third and fourth genes are swapped between Genomes A and B to create Genomes A' and B'. Figure 3(b) demonstrates single-point mutation. In this example, the probability that a network link failure will occur has risen from 20% to 22%, while the severity of the failure remains unchanged.



(a) Example of double-point crossover.



(b) Example of single-point mutation.

Figure 3: Illustration of evolutionary operators.

3. APPROACH

This section introduces the Ragnarok and Valkyrie techniques. First, the assumptions, inputs, and outputs are stated for each technique. Then, a description is provided of how each technique is applied to automatically harden an SAS against adverse conditions.

3.1 Assumptions, Inputs, and Outputs

Both Ragnarok and Valkyrie require three key inputs: a goal model representing the functional requirements of the SAS, a set of utility functions that have been derived based on the goal model for run-time requirements monitoring, and an executable specification or prototype of the SAS. While Ragnarok and Valkyrie are intended to be domain-independent, the required inputs to each technique are specific to the application domain. Each of these input elements are next described.

Goal model.
A goal model is required to provide the functional requirements of the SAS. For this paper, KAOS goal models [5, 33] are used, where each goal is designated as invariant (i.e., must be satisfied) or non-invariant (i.e., can temporarily be unsatisfied) by the requirements engineer.

Utility functions.
A set of utility functions must be derived by a requirements engineer for run-time monitoring of SAS requirements satisfaction [7, 30, 35], where each utility function maps to a KAOS goal and comprises a mathematical relationship that maps monitoring data to a value within $[0.0, 1.0]$. The utility value demonstrates how well a given goal is being satisfied at run time. For example, the satisficement of Goal (A) (c.f., Figure 2) can be measured by returning a value of 1.0 if the amount of data replicates matches the number of data mirrors and 0.0 otherwise.

Executable specification.
Both Ragnarok and Valkyrie require an executable specification, such as a simulation or prototype, of the SAS. The simulation applies the set of utility functions to measure run-time requirements satisfaction. Moreover, the executable specification is intended to subject the SAS to as wide of a range of both system and environmental parameters as possible to fully exercise the SAS, from a requirements perspective. The requirements engineer must also specify sources of uncertainty that the SAS may face, including uncertainty in the system and environment. For example, the RDM application may face uncertainty in terms of unexpected message delays, lost messages, randomly severed network links, and sensor noise.

3.2 Combined Process

Ragnarok and Valkyrie are intended to be used sequentially to harden an SAS against uncertainty, where Ragnarok discovers adverse conditions that can negatively impact an SAS, and Valkyrie generates SAS configurations that can be used to mitigate those conditions discovered by Ragnarok. Figure 4 presents a data flow diagram (DFD) that overviews both techniques together, and each step is next described in detail. For reference, Steps (1) and (2) comprise the Ragnarok technique, and Steps (3) and (4) comprise the Valkyrie technique. Given that both Ragnarok and Valkyrie each use a genetic algorithm, the evolutionary process is abstracted into a single DFD and later presented in Figure 7, where the respective genetic algorithms specifically execute in Step (2) (i.e., Ragnarok) and Step (4) (i.e., Valkyrie).

(1) Generate Adverse Environments.
Ragnarok uses a genetic algorithm [18] to generate combinations of adverse parameters that specify the sources of environmental uncertainty, each of which specify a probability of occurrence and impact to the SAS. Each set of parameters is represented as a genome within a population, where each genome is represented as a vector of length n. For each vector, n defines the number of environmental sources of uncertainty. Figure 5 presents an example genome used by Ragnarok. Specifically, the provided genome specifies that, at each timestep of SAS execution, there is a 15% chance of network link failure, and should this failure occur, up to 10% of all network links will fail. Each genome is instantiated and evaluated within the SAS simulation environment.

The genetic algorithm must be configured by a requirements engineer to search for an optimal combination of parameters. As such, evolutionary operators such as population size (i.e., number of genomes instantiated each generation), crossover and mutation rates (i.e., rate of creation of new genomes based on recombination and random modifica-

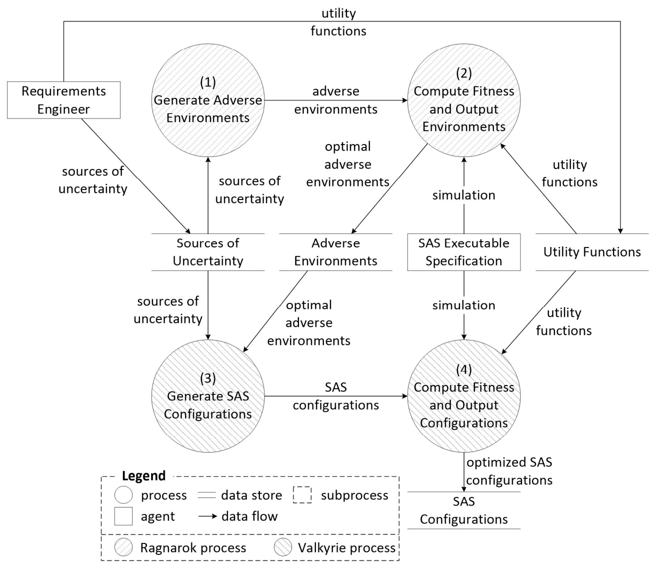


Figure 4: Data flow diagram of Ragnarok and Valkyrie techniques.

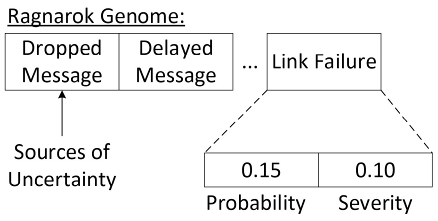


Figure 5: Genome for Ragnarok genetic algorithm.

tion, respectively), and termination criterion (i.e., number of generations to evolve genomes) must be specified. Moreover, definition of these parameters relies on domain knowledge or empirical evidence to ensure that both genetic algorithms arrive at an optimal solution. For this paper, the population size is 50 genomes, the crossover rate is 12.5%, the mutation rate is 25.0%, and the number of generations is 15, where these values were selected based on empirical evidence demonstrating an acceptable rate of convergence to an optimal solution. After the total number of generations is reached, Ragnarok outputs the highest performing genome that specifies the most adverse combinations of environmental uncertainty. Following the initial configuration of the genetic algorithm, Ragnarok generates a randomized population of genomes for the genetic algorithm to operate upon. Step (2) next describes how the impact of each genome is calculated with respect to goal satisficement monitoring.

(2) Compute Fitness and Output Environments.

The genetic algorithm used by Ragnarok optimizes the environmental sources of uncertainty in such a way that the number of *goal violations* are maximized. As such, a *fitness value* for each genome is calculated based on three criteria: maximization of the number of violated *non-invariant* goals, maximization of the number of violated *invariant goals*, and minimization of *average goal satisficement.*

Equations 1 and 2 present equations for calculating the fitness of a Ragnarok genome.[1] $FF_{Ragnarok}$ uses a linear weighted sum to specify the relative importance of each metric, where each weighting coefficient must sum to a value of 1.0. For this paper, $\alpha_{non-inv} = 0.25$, $\alpha_{inv} = 0.50$, and $\alpha_{ave} = 0.25$, as maximizing the number of violated invariant goals is considered more destructive to the SAS than maximizing the number of violated non-invariant goals or minimizing average goal satisficement.[2]

Equation 2 presents the equation for average goal satisficement during execution. Specifically, $|values_{utility}|$ represents the number of utility values calculated per timestep and $timesteps_{sim}$ represents the total number of timesteps specified for the simulation to execute. For the RDM application, $|values_{utility}| = 26$ (i.e., the total number of goals in Figure 2) and $timesteps_{sim} = 300$.

$$\begin{aligned} FF_{Ragnarok} =& 1.0 - ((\alpha_{non-inv} * violations_{non-inv}) + \\ & (\alpha_{inv} * violations_{inv}) + \\ & (\alpha_{ave} * satisficement_{ave})) \end{aligned} \tag{1}$$

where,

$$satisficement_{ave.} = \frac{\sum values_{utility}}{|values_{utility}| * timesteps_{sim}} \tag{2}$$

Upon completion of the genetic algorithm, Ragnarok provides, as output, genome(s) that instantiate sources of uncertainty that induce adverse behavior in the SAS. This output is then used by Valkyrie (i.e., Steps (3) and (4)) to configure the environment for instantiation of adverse conditions.

(3) Generate SAS Configurations.

Valkyrie uses a genetic algorithm to optimize combinations of SAS parameters to mitigate the adverse conditions discovered by Ragnarok. The SAS parameters each specify a particular value that is used to configure the SAS itself. Each set of SAS parameters is represented as a genome, where each genome is a vector of length m. For each vector, m specifies the number of configurable SAS parameters (i.e., genes). Figure 6 presents a sample genome used by Valkyrie. In this example, one configured parameter defines the size of messages replicated within the RDM network to be $2.5mb$. Configuration of Valkyrie's genetic algorithm in terms of evolutionary parameters is reused from Step (1) (i.e., population size, crossover rate, etc.).

(4) Compute Fitness and Output Configurations.

The genetic algorithm used by Valkyrie optimizes the SAS parameters in order to maximize the overall satisficement of goals. A fitness value is calculated based on a piecewise function as shown in Equation 3, where we reuse the definition of $satisficement_{ave}$ from Equation 2. Specifically, if a violation occurs in an invariant goal then the fitness score is 0.0, as the state of the SAS is considered irrecoverable when an invariant is violated. Otherwise, the fitness score

[1]For presentation purposes, Equations 1, 2, and 3 abbreviate invariant (inv), non-invariant (non-inv), simulation (sim), and average (ave).

[2]The weighting coefficients were also selected based on empirical evidence gathered during experimentation on the RDM application.

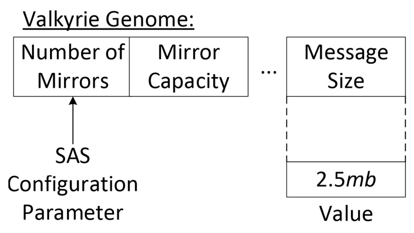


Figure 6: Genome for Valkyrie genetic algorithm.

is the average of all calculated utility values (i.e., goal satisficement) over the course of the simulation. This fitness function is intended to maximize overall satisficement of requirements, and as a result, minimize the number of goal violations.

$$FF_{Valkyrie} = \begin{cases} 0.0 & \text{if } viols_{inv} > 0 \\ satisficement_{ave} & else \end{cases} \tag{3}$$

Upon completion of the Valkyrie genetic algorithm, a set of the highest performing SAS configurations are provided as output. Given that an SAS comprises a collection of configurations (either physically or logically), Valkyrie-generated configurations can be used to harden the initial SAS configuration state (i.e., in production) against those environmental uncertainties discovered by Ragnarok. Hardening the SAS can be performed by including the discovered SAS configurations in the collection, augmenting base SAS configurations using discovered parameters, or by updating SAS requirements specification to consider the Ragnarok-discovered environments.

Evolutionary Loop.

We next present the data flow specific to the evolutionary process. Specifically, Steps (2) and (4) (c.f., Figure 4) can be expanded to comprise a general evolutionary loop as presented in Figure 7. This figure is relevant to both Ragnarok and Valkyrie, however the genomic configurations entering the loop will either be specific to the environment (i.e., Ragnarok, Step (2)) or specific to the SAS (i.e., Valkyrie, Step (4)). Each step is next described in turn.

(A) Evaluate Configurations.

To evaluate the quality of the provided configuration (i.e., adverse environments or optimal SAS configurations), each configuration is instantiated within the provided simulation environment. An initial population of randomized configurations is received (i.e., as output from Step (1) or Step (3)) and then evolved over the course of the genetic algorithm. Specifically, the environmental configuration is mapped to uncertainty parameters specified by the requirements engineer (e.g., probability and impact of network link failure, probability and impact of delayed messages, etc.), and the SAS configuration then instantiates the system itself. For the Ragnarok evolutionary loop, the environmental parameters are being evolved and as such, the SAS configuration remains static. Conversely, the Valkyrie evolutionary loop optimizes the SAS configuration while the environmental configuration, provided by Ragnarok, remains static throughout the loop. For Ragnarok and Valkyrie, each configuration

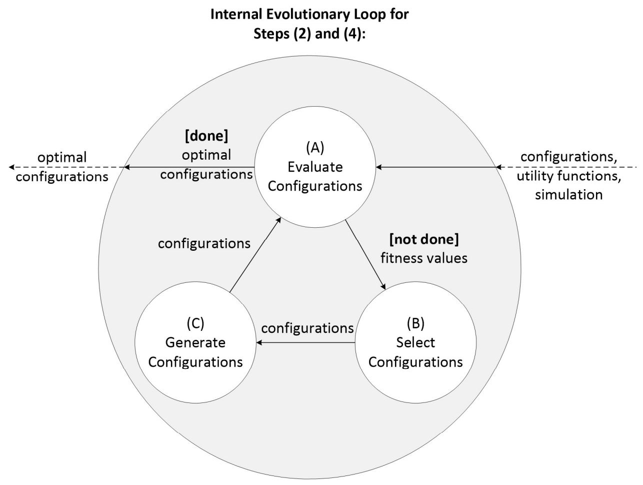

Internal Evolutionary Loop for Steps (2) and (4):
optimal configurations
[done] optimal configurations
(A) Evaluate Configurations
configurations, utility functions, simulation
configurations
[not done] fitness values
(C) Generate Configurations
configurations
(B) Select Configurations

Figure 7: Internal data flow diagram of Ragnarok and Valkyrie genetic algorithms.

is evaluated based on the fitness criteria specified in Equations 1 and 3, respectively.

(B) Select Configurations.

Using the fitness values calculated for each genome, the genetic algorithm selects the highest-performing individuals from the population. Selection drives the search process towards optimal areas of the overall solution space. In this regard, Ragnarok and Valkyrie each implement tournament selection [18], a technique that randomly selects k individuals from the population. These k individuals then compete to determine which survives to the next generation. The individual with the highest fitness value survives, and the remaining individuals are discarded.

(C) Generate Configurations.

The evolutionary process then generates new configurations, internal to the evolutionary loop, to ensure that the population size remains constant. As such, new genomes are generated based on two-point crossover and single-point mutation, respectively (c.f., Section 2.3). The intent of the crossover operator is to ideally construct better genomes based on parent genomes that already are performing well. Mutation, however, attempts to introduce diversity into the population by through random change. Diversity is necessary to ensure that the search procedure does not get stuck in a local optima, but rather is guided towards a global optima in the overall solution space. This process continues until the total number of generations is exhausted, at which point the optimized configurations are provided as output. For Ragnarok, optimized adverse environments are provided as output for Valkyrie. For Valkyrie, optimized SAS configurations are provided as output to the requirements engineer.

4. EXPERIMENTAL RESULTS

This section describes the experimental setup and presents experimental results from applying Ragnarok and Valkyrie to the RDM application.

4.1 Experimental Setup

For this paper, the RDM network was modeled as a completely connected graph, where each node represents an RDM and each edge represents a network link. There were 50 experimental trials performed for statistical significance. For each trial, a random number of data mirrors (i.e., within $[15, 30]$) formed the network. A random number of messages (i.e., within $[100, 200]$) were randomly inserted into different mirrors in the network and were required to be replicated to all other data mirrors. The simulation was performed over 300 timesteps.

Both system and environmental uncertainty were modeled in the RDM application. The RDM can experience delays in message distribution, randomly dropped messages, unexpected network link failures, and random noise applied to both data mirrors and network link sensors. To mitigate these uncertainties, the RDM network can reconfigure to ensure that its requirements are continually satisfied. Possible reconfigurations include changes to the network topology and updates to data propagation parameters.

This paper presents two experiments. The first experiment explores environmental conditions that adversely affect the RDM application. Using the fitness function provided in Equation 1, Ragnarok searches for combinations of environmental parameters that negatively impact the utility functions derived for the RDM. The resulting sets of environmental configurations can then be used to demonstrate environments in which the RDM application cannot effectively function. For this experiment, Ragnarok-generated environmental configurations are compared to those generated by random search, as we do not know specifically which combinations of parameters will induce requirements violations in the RDM application.

The second experiment subjects the RDM application to the Ragnarok-generated environments in order to search for combinations of system parameters that can mitigate those adverse conditions. Specifically, Valkyrie uses the fitness function provided in Equation 3 to optimize the parameters (e.g., number of data mirrors, size of data message, network link capacity, etc.) that configure the RDM application. Valkyrie-generated RDM configurations are then compared to configurations generated by random search, as we do not explicitly know which combination of parameters can handle such adverse conditions.

For each experiment, we use the Wilcoxon-Mann-Whitney U-test to determine if statistical significance exists between our data samples, given no normality of data is assumed.

4.2 Experimental Results

We next present the experimental results from applying Ragnarok and Valkyrie to the RDM application.

Ragnarok Experimental Results.

For this experiment, we define the null hypothesis $H1_0$ to state that "there is no difference between an environmental configuration generated by Ragnarok and a randomly-generated environmental configuration". Figures 8 and 9 present boxplots of the average number of invariant and non-invariant goal violations encountered throughout execution of the RDM simulation, respectively. These figures demonstrate that Ragnarok can induce significantly more violations of both invariant and non-invariant goals in an SAS when compared to random search ($p < 0.05$). Moreover, viola-

Parameter	Random	Ragnarok
Prob. data mirror failure	1.4%	3.8%
Prob. network link failure	0.3%	2.9%
Prob. data message dropped	10.3%	16.0%
...	...	...
Sensor fuzz rate	11.4%	8.4%

Table 1: Sample environmental configurations.

tion of any invariant goal (i.e., Figure 2, Goals (A) and (B)) causes total system failure. Therefore, the conditions found by Ragnarok provide a significantly destructive set of adverse conditions that the SAS cannot tolerate that may not have been considered during requirements elicitation. The environmental configuration with the highest fitness value (c.f., Equation 1) is provided to Valkyrie as output.

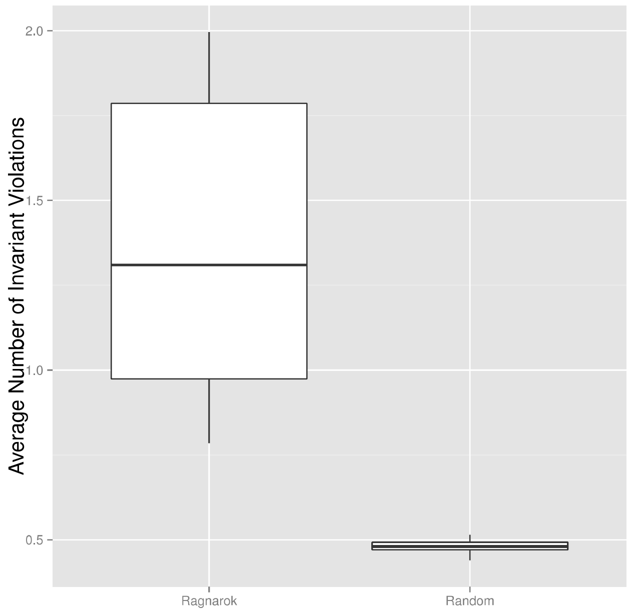


Figure 8: Comparison of RDM invariant goal violations between Ragnarok and random search.

These figures demonstrate the destructive nature of Ragnarok. Specifically, Ragnarok generates combinations of environmental parameters that induce requirements violations. For example, Table 1 presents subsets of two environmental configurations that were generated by random search and Ragnarok, respectively. As such, a Ragnarok environmental configuration can specify a higher probability that data mirrors and network links will fail, and moreover specify that there is a higher chance that messages will be dropped during replication. Moreover, there is also a chance that Ragnarok-generated values will be less adverse than expected (e.g., the rate at which noise is applied to RDM sensors). This type of configuration is provided for each source of environmental uncertainty specified by the requirements engineer.

The parameters discovered by Ragnarok are not strictly value increases to simply induce violations. Each uncertainty parameter is specified, by the requirements engineer,

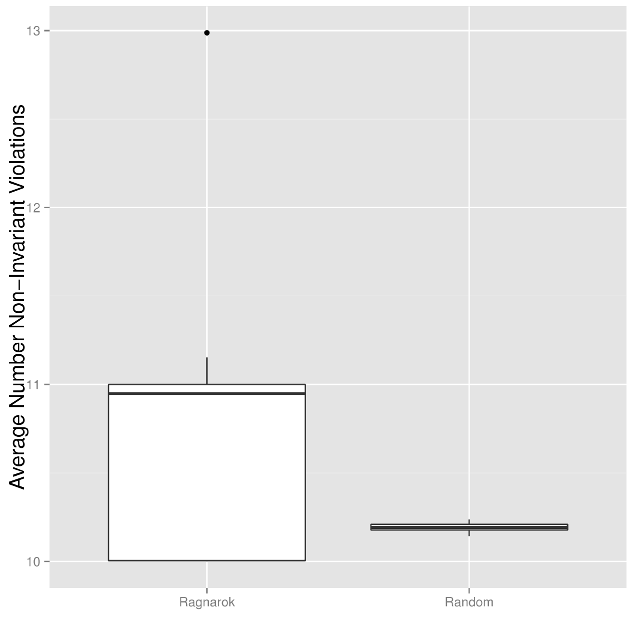


Figure 9: Comparison of RDM non-invariant goal violations between Ragnarok and random search.

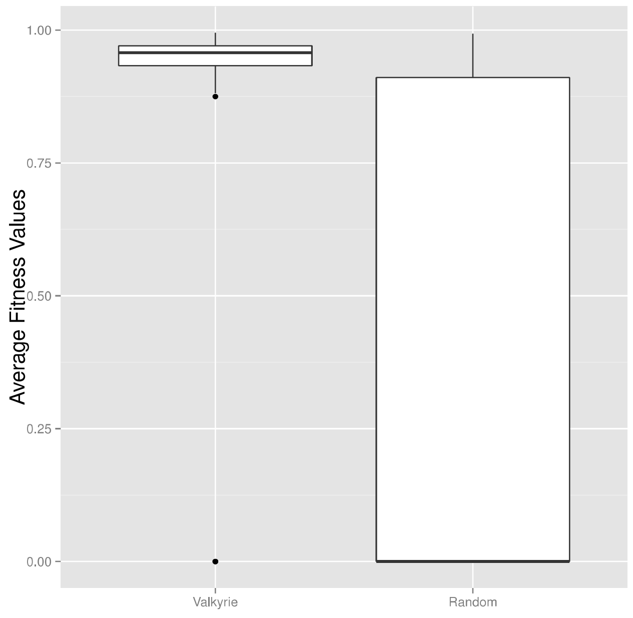


Figure 10: Comparison of average fitness values between Valkyrie and random search.

to take on an acceptable range of values prior to evolution. For example, each Ragnarok-generated value presented in Table 1 falls within an acceptable range of values. However, the *combination* of such parameters induces requirements violations. Given the results presented in Figures 8 and 9, $H1_0$ can be rejected, suggesting that Ragnarok-generated environments are more adverse than randomly-generated environments. The following section next discusses how Valkyrie can automatically mitigate such conditions.

Valkyrie Experimental Results.

For this experiment, we define the null hypothesis $H2_0$ to state that "there is no difference between an SAS configuration generated by Valkyrie and a randomly-generated SAS configuration". Moreover, the environment configurations used for this experiment were generated by Ragnarok. Figure 10 presents the average fitness values (c.f., Equation 3) calculated for Valkyrie-generated RDM configurations and those generated by random search. The boxplots in this figure demonstrate that Valkyrie can significantly increase the overall fitness (i.e. goal satisficement) of the RDM application when subjected to adverse conditions ($p < 0.05$).

Next, Figure 11 presents boxplots of the average number of non-invariant goal violations that occurred throughout each simulation. While both Valkyrie and random search each had a minimum number of 10 non-invariant violations, there are significantly fewer violations that occurred in Valkyrie-generated configurations ($p < 0.05$), indicating that Valkyrie can help to minimize non-invariant goal violations.

Lastly, Figure 12 presents boxplots that show the average amount of invariant goal violations that manifested throughout each simulation. As such, Valkyrie-generated configurations significantly minimized the number of invariant goal violations when compared to random search ($p < 0.05$). This result suggests that Valkyrie-generated configurations can also assist in minimizing invariant goal violations when experiencing adverse conditions.

Table 2 presents a subset of RDM configurations generated by random search and Valkyrie, respectively. For example, parameters can include the size of messages to be replicated across the network, the base capacity of all data mirrors, the overall operating budget to run the RDM network, the number of available data mirrors, and the initial network topology. In this example, Valkyrie was able to increase overall goal satisficement by lowering the configured size of messages, increasing the capacity of each data mirror, and increasing the overall operating budget. Interestingly, the number of available data mirrors was reduced from 29 to 22, indicating that the increased capacity must be balanced by reducing the number of mirrors. Moreover, the network topology remained the same, suggesting that the topology was not a major factor in goal satisficement for this particular configuration of *environmental parameters*. Given the results presented in Figures 10 – 12, $H2_0$ can be rejected, suggesting that Valkyrie-generated configurations can increase the overall effectiveness of the RDM when faced with highly-adverse environmental conditions.

Parameter	**Random**	**Valkyrie**
Message size	$2.44mb$	$1.68mb$
Data mirror base capacity	$4.85gb$	$7.34gb$
Budget	$\$401,944.87$	$\$532,850.52$
...	...	...
Number of data mirrors	29	22
Base network topology	Grid	Grid

Table 2: Sample RDM configurations.

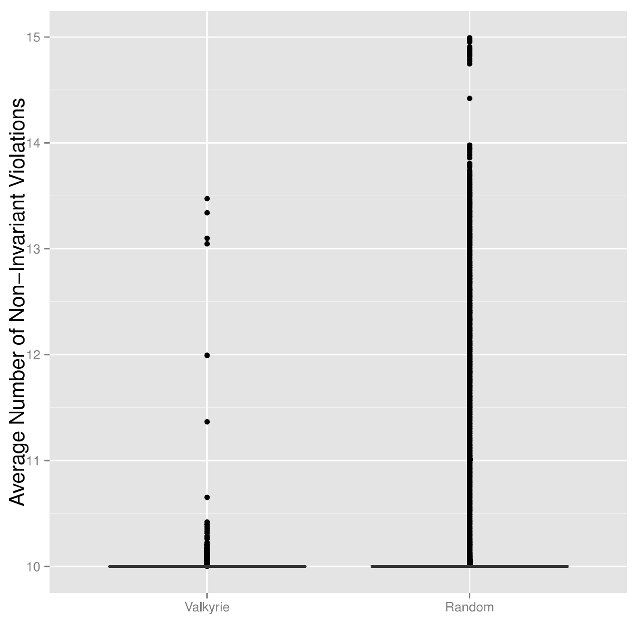


Figure 11: Comparison of RDM non-invariant goal violations between Valkyrie and random search.

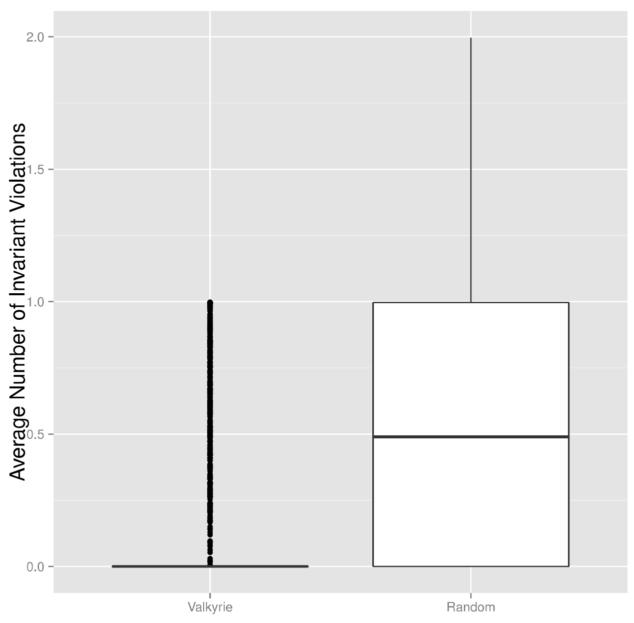


Figure 12: Comparison of RDM invariant goal violations between Valkyrie and random search.

4.3 Mitigation Strategies for Destructive Uncertainty

The experiments performed for Ragnarok and Valkyrie demonstrated the effectiveness of using search-based techniques for both discovering adverse combinations of environmental parameters and mitigating those conditions using optimized combinations of SAS parameters. For instance, one Ragnarok-generated configuration had elevated probabilities of data mirror failure, dropped messages, and a lower rate at which sensor fuzz is applied (c.f., Table 1). In this case, Valkyrie discovered that reducing the size of each message, increasing the overall capacity of each data mirror, and increasing the overall budget enables the RDM to perform better than with its original configuration (c.f., Table 2). Furthermore, Valkyrie discovered that the total number of data mirrors could be reduced as well, given the increased capacity of each mirror.

A requirements engineer could then use the information found by both Ragnarok and Valkyrie to harden the RDM application prior to deployment. Specifically, the engineer could augment the requirements specification to consider the environment(s) found by Ragnarok. Such an approach would entail adding requirements to specifically mitigate the discovered conditions (e.g., network must be pre-configured to a specific topology with a set of particular links activated), or to request a budget increase that can support an increased operating cost of reconfigured network links when faced with adversity. Another approach would be to store the Valkyrie-generated RDM configuration for use if and when the environmental conditions are detected. By monitoring its environment (i.e., as part of the SAS MAPE-K loop [23]), the RDM can detect when the conditions discovered by Ragnarok manifest within its environment. Upon detection, the RDM could self-reconfigure to use stored, Valkyrie-generated RDM configuration for use in mitigating an environment that had already been resolved.

Threats to Validity.

This research was a proof of concept to determine the feasibility of both generating combinations of environmental parameters that specifically break a system and generating SAS configurations that can mitigate those specific environments. One threat to validity is if Ragnarok and Valkyrie will achieve similar results in other application domains, specifically those that involve scalability concerns. Another threat to validity involves the manual derivation of the utility and fitness function coefficients. A third threat to validity lies in the viability of using the automatically-generated SAS configurations to environments that differ from those generated by Ragnarok. Lastly, a fourth threat to validity is if Ragnarok and Valkyrie will achieve similar results when compared to other search-based techniques (e.g., simulated annealing, pattern search, etc.).

5. RELATED WORK

This section highlights related work on uncertainty generation, self-healing systems, and obstacle mitigation.

5.1 Automated Generation of Uncertainty

Ramirez *et al.* previously introduced Loki [29], a technique for automatically discovering *novel* combinations of parameters for configuring both an SAS and its environment. Using these parameter combinations, Loki aims to uncover unexpected or latent errors within the SAS requirements specification. Furthermore, Fenrir [15] is a complementary technique that explores how uncertainty can impact an SAS at the code level by generating combinations of system and environmental parameters that yield novel execution traces. Both Loki and Fenrir use *novelty search* [26] to explore unique combinations of parameter values. As such, Ragnarok differs from both techniques in that the solution space comprises *destructive* parameters as opposed to *novel* parameters. Specifically, Ragnarok is concerned with requirements violations rather than requirements interactions, as Loki and Fenrir do not necessarily discover conditions that otherwise break the system. Moreover, this paper introduces a complementary technique, Valkyrie, to automatically mitigate the discovered conditions, whereas both Loki and Fenrir specified that the requirements engineer manually resolve discovered issues.

Research into quantifying and mitigating uncertainty has recently been the focus of the software engineering community [2, 4, 8, 9]. Moreover, a taxonomy of different sources of uncertainty has recently been published by Esfahani and Malek [10]. As such, our work attempts to further examine the issues related to uncertainty by examining both how it can negatively impact a system as well as provide automatically-generated mitigation strategies. Etxeberria *et al.* have recently performed a sensitivity analysis on how uncertainty parameters can affect system performance [11]. While this paper does not strictly perform a sensitivity analysis on the derived utility functions and fitness functions, the search for parameters that can affect an SAS at its requirements level does provide a measure of sensitivity analysis with respect to configurable system and environmental parameters.

5.2 Self-Healing Systems

Self-healing systems can automatically repair themselves based on identification of errors at run time using a robust adaptation engine. Such systems are very similar to SASs, however an adaptation in a self-healing system is typically in response to a perceived fault condition as opposed to monitored requirements violations. As such, Ghosh *et al.* have provided an excellent survey on self-healing systems [17], including an overview of healing strategies and a discussion of existing techniques. Dashofy *et al.* have introduced a research path for an architecture-based self-healing system where healing occurs by repairing deficiencies in the system's architecture [6]. Garlan and Schmerl have also discussed how adaptation in self-healing systems can be accomplished using a model-based approach, including facilities for monitoring, translating monitored values to the architectural level, updating the architecture, validating updates, and then performing the repair [16]. Ragnarok and Valkyrie provide a level of healing within the SAS domain, wherein Ragnarok serves as a monitoring and fault discovery mechanism, and Valkyrie serves as an analog for healing the system. However, both Ragnarok and Valkyrie are concerned with the requirements level as opposed to the architectural level.

5.3 Obstacle Mitigation

A set of strategies for identifying, analyzing, and resolving obstacles to objective satisfaction has been previously proposed by van Lamsweerde *et al.* [33, 34]. If an obstacle is unavoidable, then a mitigation strategy can be used to temporarily tolerate goal violation. As such, Ragnarok can be used to complement this approach by automatically identifying environmental conditions that induce goal violations. Moreover, Valkyrie can be used to complement a mitigation strategy by optimizing the SAS configuration for tolerating adversity. Letier and van Lamsweerde have introduced a probabilistic approach for specifying a probability that a goal will be satisfied using domain knowledge [27]. However, this approach only considers goals to be satisfied or unsatisfied, whereas Ragnarok and Valkyrie use a utility-based approach to quantifying goal satisfaction. Another approach uses search-based techniques to discover faults in real-time embedded systems [19]. This approach is similar to Ragnarok in that environmental conditions are generated to induce faults, however this approach focuses on black-box testing of embedded systems, whereas Ragnarok focuses on validating requirements via utility value monitoring, and moreover focuses specifically on SASs. Lastly, formal guarantees for SAS controllers have been recently proposed by Filieri *et al.* [12]. This approach leverages control theory and verification techniques to provide an SAS that is guaranteed to deliver acceptable behavior. Conversely, Ragnarok and Valkyrie do not specifically provide this guarantee, however they do provide a measure of requirements validation in response to uncertainty, specifically in regard to system and environmental conditions that may not have been accounted for by the requirements engineer.

6. CONCLUSION

This paper has presented Ragnarok and Valkyrie, two design-time techniques that can be used together to harden an SAS against uncertainty. Ragnarok first automatically identifies environmental conditions that induce requirements violations in an SAS. Valkyrie can then be used to automatically identify SAS configurations that can mitigate the discovered conditions. The requirements specification can then be improved based on adversity identified by Ragnarok, and the set of available SAS configurations or base SAS configuration can be augmented with Valkyrie-generated configurations. Ragnarok and Valkyrie were each demonstrated on an RDM network application that was required to replicate data messages across a network of data mirrors. The RDM could experience uncertainty in terms of dropped messages, delayed messages, network link failures, and sensor noise. The RDM could then self-reconfigure to mitigate run-time uncertainty. Experimental results suggest that Ragnarok can generate combinations of environmental parameters that can induce significantly more invariant and non-invariant goal violations than can be found with random search, and Valkyrie-generated SAS configurations can significantly increase goal satisficement and reduce goal violations than configurations generated by random search. Future work includes application of Ragnarok and Valkyrie to other SAS application domains (e.g., embedded systems and cloud-based applications), exploration of other search-based techniques (e.g., novelty search, particle swarm optimization), and extension of Valkyrie for run-time generation of optimal SAS configurations.

7. ACKNOWLEDGMENTS

This work has been supported by Oakland University. 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 Oakland University or other research sponsors.

8. REFERENCES

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

[2] L. Baresi, L. Pasquale, and P. Spoletini. Fuzzy goals for requirements-driven adaptation. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 125 –134, 27 2010-oct. 1 2010.

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

[4] 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 *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 468–483, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] A. Dardenne, A. Van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of computer programming*, 20(1):3–50, 1993.

[6] E. M. Dashofy, A. van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, WOSS '02, pages 21–26. ACM, 2002.

[7] P. deGrandis and G. Valetto. Elicitation and utilization of application-level utility functions. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 107–116. ACM, 2009.

[8] S. Elbaum and D. S. Rosenblum. Known unknowns: Testing in the presence of uncertainty. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 833–836, 2014.

[9] N. Esfahani, E. Kouroshfar, and S. Malek. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 234–244. ACM, 2011.

[10] N. Esfahani and S. Malek. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, pages 214–238. Springer, 2013.

[11] L. Etxeberria, C. Trubiani, V. Cortellessa, and G. Sagardui. Performance-based selection of software and hardware features under parameter uncertainty. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 23–32, 2014.

[12] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, pages 71–82, 2015.

[13] E. M. Fredericks, B. DeVries, and B. H. C. Cheng. Autorelax: Automatically relaxing a goal model to address uncertainty. *Empirical Software Engineering*, 19(5):1466–1501, 2014.

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

[15] E. M. Fredericks, A. J. Ramirez, and B. H. C. Cheng. Validating code-level behavior of dynamic adaptive systems in the face of uncertainty. In *Search Based Software Engineering*, volume 8084 of *Lecture Notes in Computer Science*, pages 81–95. Springer Berlin Heidelberg, 2013.

[16] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the First Workshop on Self-healing Systems*, pages 27–32, Charleston, SC, 2002. ACM.

[17] D. Ghosh, R. Sharman, H. Raghav Rao, and S. Upadhyaya. Self-healing systems – survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2006.

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

[19] M. Z. Iqbal, A. Arcuri, and L. Briand. Empirical investigation of search algorithms for environment model-based testing of real-time embedded software. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 199–209, 2012.

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

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

[22] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.

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

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

[25] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE'09, pages 510–514, Auckland, New Zealand, November 2009. IEEE Computer Society.

[26] J. Lehman and K. O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *Proceedings of the Eleventh*

International Conference on Artificial Life, ALIFE XI. MIT Press, 2004.

[27] E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 53–62, 2004.

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

[29] 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. (Preliminary work described in short paper).

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

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

[32] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein. Requirements-aware systems: A research agenda for re for self-adaptive systems. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 95 –103, 2010.

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

[34] A. Van Lamsweerde and E. Letier. Handling obstacles in goal-oriented requirements engineering. *Software Engineering, IEEE Transactions on*, 26(10):978–1005, 2000.

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

[36] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. Bruel. Relax: Incorporating uncertainty into the specification of self-adaptive systems. In *17th IEEE International Requirements Engineering Conference (RE'09)*, pages 79–88, 2009.