Automatically Exploring How Uncertainty Impacts Goal Satisfaction

Andres J. Ramirez, Adam C. Jensen, Betty H.C. Cheng, and David B. Knoester
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI, USA
{*ramir105, acj, chengb, dk*}*@cse.msu.edu*

***Abstract*—A dynamically adaptive system (DAS) monitors itself and its execution environment to assess requirements satisfaction at run time. Unanticipated environmental conditions may cause sensory inputs that alter the self-assessment capabilities of a DAS in unpredictable and undesirable ways. Moreover, it is impossible for a human to know and/or enumerate all possible combinations of system and environmental conditions that a DAS may encounter. This paper introduces Loki, an evolutionary computation-based approach that automatically discovers environmental conditions that lead to requirements violations and latent behaviors in a DAS. As such, Loki facilitates the identification of goals with inadequate obstacle mitigations or insufficient constraints to prevent unwanted behaviors. We apply Loki to an autonomous vehicle system that performs adaptive cruise control, lane keeping, and collision avoidance. Experimental results show that Loki discovers environmental conditions that obstruct goals and detects a significantly larger number of different and unexpected behaviors than randomized search.**

***Keywords*-requirements engineering, evolutionary algorithm, novelty search, environmental uncertainty, self-adaptation**

I. Introduction

A dynamically adaptive system (DAS) must monitor itself and its execution environment to detect conditions warranting a reconfiguration. At run time, a DAS analyzes this monitoring information to determine when and how to *safely* reconfigure itself such that it satisfies its requirements. Unfortunately, unanticipated environmental conditions may negatively impact the accuracy and reliability of monitoring information, thereby compromising the decision-making abilities of a DAS. Moreover, it is impossible for a human to know and/or enumerate all possible combinations of environmental conditions that a DAS may encounter throughout its lifetime [1]. As a result, it is important to explore how environmental conditions impact the behavior of a DAS before the implementation phase, while there is greater flexibility for resolving obstacles that prevent the satisfaction of goals [2]. This paper introduces Loki [1] an evolutionary computation-based approach for automatically identifying combinations of environmental conditions that obstruct goals. Loki can be leveraged to generate suites of test cases as well as suggest refinements to a goal model.

[1]In Norse mythology, Loki is the god of mischief and trickery.

Early system requirements and domain assumptions are often ambiguous and idealized, thus leading to inconsistencies between a system specification and its behavior at run time [2], [3]. To augment goal models with more comprehensive and realistic requirements, Van Lamsweerde and Letier proposed a set of heuristics, refinement patterns, and formal techniques for reasoning about obstacles [2]–[4] and partial goal satisfaction [5]. However, as DASs become more intertwined with the physical elements, including the environment, it becomes increasingly impractical for a human to exhaustively explore environmental conditions that adversely impact a DAS [1], [6]. Lutz and Mikulski demonstrated how testing could discover incomplete requirements and unexpected requirements interactions [7]. Recently, evolutionary algorithms have been applied to generate suites of test cases that cause a failure in the system under test [8]–[10]. While promising, these approaches often require extending an evolutionary algorithm with domain-specific objective functions that evaluate the quality of generated test cases. As such, these approaches may also converge upon specific types of failures that satisfy the objective function's criteria.

This paper introduces Loki, a domain-independent evolutionary computation-based approach for automatically exploring how uncertainty obstructs goals in a DAS, where uncertainty refers to the unknown effects of environmental conditions upon goal satisfaction. Instead of searching for specific instances of goal obstructions, however, Loki generates a diverse set of behaviors in a DAS that may lead to goal obstructions. To achieve this objective, Loki applies the concept of novelty search [11] to generalize, or collapse, vast collections of possible behaviors into a small number of representative behaviors. Searching for novelty enables Loki to discover both requirements violations and latent behaviors. While a requirements violation clearly obstructs a specific set of system goals, latent behaviors manage to satisfy requirements through unexpected and potentially undesirable behaviors. The set of environmental conditions that caused requirements violations and latent behaviors should be reused to guide the testing process of implemented systems.

Traditional evolutionary algorithms, such as genetic algorithms [12], are stochastic search-based techniques that use the process of evolution by natural selection to generate

solutions to complex problems. Typically, this search process is guided by a domain-specific *fitness function* that evaluates the quality of generated solutions. Novelty search [11] is a type of evolutionary algorithm where the fitness function is replaced by a domain-independent *novelty function* that measures the behavioral difference between solutions. Specifically, novelty functions reward solutions that are different from those previously discovered. To this end, Loki first applies a genetic algorithm to generate configurations that specify the type, duration, and severity of noise that is applied to sensors in a DAS. Loki then simulates system and environmental conditions and applies a set of utility functions to assess how well the DAS satisfied requirements at run time [13], [14]. Comparing the differences between values produced by these utility functions enables Loki to automatically evaluate the behavior of a DAS in response to perceived environmental conditions, thereby facilitating the identification of requirements violations and latent behaviors.

We illustrate Loki by applying it to an autonomous intelligent vehicle system (IVS) that performs adaptive cruise control, lane keeping, and collision avoidance. To this end, we leverage a set of IVS goal models to implement an IVS prototype in the Webots simulation platform [15]. Experimental results show Loki is able to discover combinations of environmental conditions that lead to requirements violations and latent behaviors. The remainder of this paper is organized as follows. Section II provides background information on goal-oriented requirements engineering and novelty search. In Section III, we describe and apply Loki to the IVS application. Next, we present experimental results in Section IV. Section V provides an overview of related work. Lastly, we summarize our approach, discuss findings, and present future directions for this work in Section VI.

II. Background

In this section, we provide background information on goal-oriented requirements engineering and evolutionary computation, including novelty search.

A. Goal-oriented Requirements Engineering

A goal captures the intentions of a stakeholder on the system-to-be and its execution environment [16]. Essentially, a goal restricts the states that the system-to-be may reach during execution. Goals refer either to functional or non-functional properties, the key distinction being that a functional goal declares a service that the system-to-be must provide whereas a non-functional goal imposes a quality criterion upon the delivery of those services. In addition, goals may also be classified as hard or soft goals. While the satisfaction of a hard goal can be assessed in a clear-cut (yes or no) sense, the satisfaction of a soft goal cannot be determined precisely because it involves subjective preferences [16], [17]. As such, hard goals can be achieved whereas soft goals can be *satisficed*, or satisfied to a sufficient degree according to preferences [17].

Functional and non-functional goals may be decomposed into subgoals through AND/OR refinements [16]. A goal that has been AND-decomposed is satisfied if all of its subgoals are satisfied. In contrast, a goal that has been OR-decomposed is satisfied if at least one of its subgoals is satisfied. The objective of goal decomposition is to gradually refine high-level goals into finer-grained goals that can be assigned to a single system or environmental *agent* that is responsible for accomplishing the goal. Goals under the responsibility of a single agent in the system-to-be are *requirements*, and those under the responsibility of a single environmental agent are *expectations*. Goals can also be associated with specific obstacles that are conditions that prevent or obstruct the satisfaction of the goal itself. An obstacle mitigation is a necessary precondition for the respective goal to be achieved [2].

B. Evolutionary Computation and Novelty Search

Evolutionary computation is a stochastic search-based technique that applies the concept of natural selection as a heuristic to optimize solutions to complex problems. In general, evolutionary computation approaches, such as genetic algorithms [12], comprise a *population*, or collection, of genomes, where each genome *encodes*, or represents, a solution. For example, it is common for a genome in a genetic algorithm to comprise a vector where each cell represents some dimension of a solution. Evolutionary computation approaches typically generate new solutions by applying evaluation, selection, crossover, and mutation operators. Specifically, the *fitness* of a genome is evaluated by applying a fitness function that maps each genome to a scalar value proportional to its quality. This fitness value is then used to select the best genomes in the population for further exploration. While crossover exchanges parts of existing genomes in the population to form new solutions, mutation randomly modifies a genome. These processes are repeated until either a "good-enough" solution is found, or the maximum number of *generations* or iterations are exhausted. Typically, the genome with the highest fitness value is returned as the solution.

Novelty search is a search heuristic developed by Lehman and Stanley [11] to prevent evolutionary computation approaches from becoming "trapped" in deceptive and suboptimal areas of the solution space. To achieve this objective, novelty search replaces the fitness function of an evolutionary computation approach with a novelty function that comprises a distance metric, such as Euclidean distance. Novelty search applies this novelty function to calculate the distance between solutions in the population and a *novelty archive* of previously stored solutions. The novelty of a solution is computed by calculating the mean distance between a solution and its k-nearest solutions, where k is usually

determined empirically. Moreover, if the novelty value of a solution is above some threshold, then the solution is added to the novelty archive to track which areas of the solution space have already been explored. Rewarding solutions in sparse areas of the solution space enables novelty search to discover new and distinguishable solutions.

III. Approach

This section describes how Loki generates environmental conditions to produce different behaviors in a DAS. First, we present an overview of Loki and state its assumptions, inputs, and outputs. We then introduce the intelligent vehicle system (IVS) application that is used throughout the remainder of this paper to illustrate Loki, list its requirements, and describe its corresponding goal model. Next, we describe how to manually derive utility functions for assessing the satisfaction of requirements at run time. Lastly, we describe how to configure Loki to search for different behaviors of a DAS.

A. Overview of Loki

Loki explores the behaviors of a DAS in response to different combinations of environmental conditions in order to facilitate the task of identifying missing or inadequate obstacle mitigations and goals that could be further constrained. The data flow diagram in Figure 1 provides an overview of Loki. For this work, we assume a requirements engineer has already constructed a KAOS goal model [16], [18] and a simulation-based prototype that provides an executable specification of the system-to-be. Given these inputs, a requirements engineer (1) manually **derives utility functions** to assess requirements satisfaction at run time. These utility functions, along with an object model that specifies system and environmental agents, are used to (2) **configure the simulation and novelty search algorithm**. Next, (3) the **novelty search algorithm is applied** to execute the simulations, assess requirements satisfaction, and measure the distances between discovered solutions. Lastly, the observed behaviors, and the environmental conditions that caused such behaviors, are (4) analyzed to **revise the current goal model**.

B. Application Description

An intelligent vehicle system (IVS) provides autonomous vehicle control not only as a convenience to drivers, but also to ensure the safe and efficient transportation of passengers across roadways. To this end, an IVS must perform adaptive cruise control (ACC) and lane keeping while avoiding collisions. ACC commands the vehicle's engine to maintain a desired speed that depends upon the presence of other vehicles in front of the IVS. Lane keeping, on the other hand, detects roadway markings and commands the vehicle's steering mechanisms to maintain the vehicle within the center of the driving lane.

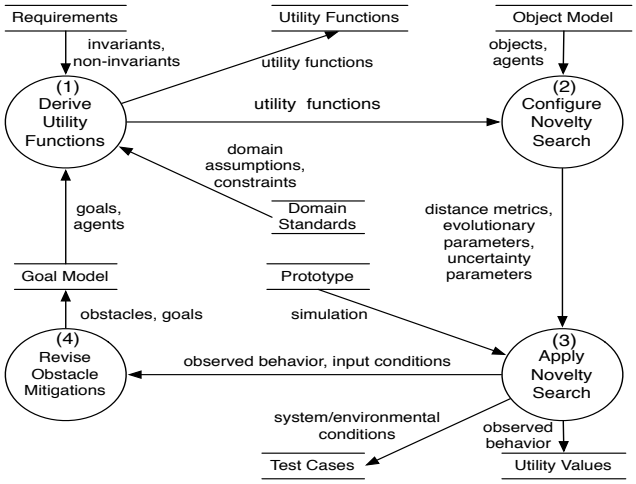

Figure 1. Data flow diagram describing the Loki approach.

For this work, we extended the implementation of a generic IVS vehicle provided with the Webots simulation platform [15]. As Figure 2 illustrates, the IVS is equipped with a monitoring infrastructure that supports ACC, lane keeping, and collision avoidance. The default IVS model implemented in Webots comprises a GPS unit to compute the position and speed of the IVS, an accelerometer to determine the acceleration and deceleration rates of the vehicle, and a camera for detecting lane markings and obstacles on the road. In addition to these devices, we also added a compass to compute the orientation of the IVS, a gyroscope to detect abrupt changes in vehicle speed or heading, three additional cameras to monitor lane boundaries and obstacles, and ten laser and sonar-based distance sensors to measure distance to obstacles around the IVS. For the remainder of this paper, we use the term IVS to refer to the extended IVS comprising the extra sensors.

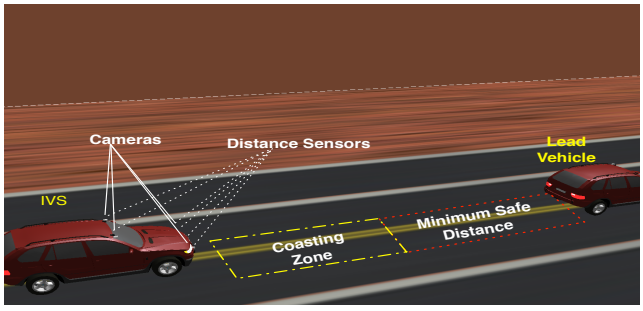

Figure 2. IVS configuration and operational environment.

For the IVS application, we identified the following invariants:

R1: The system shall maintain a safe minimum distance between the IVS and other vehicles on the road.
R2: The system shall maintain the vehicle within the lane boundaries.

In addition to these invariants, we also identified the following non-invariant requirements:

R3: The system shall provide automated speed adjustments to achieve vehicle speed equal to desired speed.
R4: The system shall minimize abrupt changes in vehicle heading and vehicle speed.

The KAOS goal model in Figure 3 further elaborates the ACC goals and their corresponding refinements into fine-grained requirements and expectations. In particular, ACC must always maintain a *minimum safe distance* (as illustrated in Figure 2) between the IVS and other vehicles on the road. To achieve this high-level goal, ACC must achieve a desired speed that depends upon both the vehicle's current speed and the measured distance to vehicles in front of the IVS. As such, the ACC must first compute the vehicle's current speed and detect the presence of nearby vehicles. Two OR-refinements exist for computing the vehicle's speed as well as for measuring the distance between the IVS and nearby vehicles, where each branch differs in the set of agents assigned to satisfy the corresponding requirements and expectations. After computing the vehicle's speed and distance to nearby vehicles, the IVS issues commands to a speed controller to adjust its current speed as necessary while the IVS is in the *coasting zone*. We elide the lane keeping goal model due to space constraints.

C. Deriving Utility Functions

Recently, utility functions have been applied for self-assessment purposes in DASs. For example, Valeto *et al.* applied utility functions to determine if a DAS should self-reconfigure to improve service delivery at run time [14]. Similarly, Garlan *et al.* [13] applied utility functions to select architectural reconfigurations that best satisfied non-functional architectural constraints. In the same spirit, Loki uses a set of utility functions to assess how well a DAS satisfies its requirements at run time. Specifically, Loki uses this information as part of the novelty search distance metric. For this work, we manually derive a set of utility functions by applying different heuristics, such as those proposed by Letier and van Lamsweerde for deriving objective functions and quality variables when reasoning about partial goal satisfaction [5].

Example. Consider the following utility function for the two goals in Figure 3 associated with achieving a safe speed:

$$U(ss, vs) = 1 - \{\min(\frac{|ss - vs|}{ss}, 1)\}$$

where *ss* and *vs* refer to the vehicle's safe speed and current speed, respectively. This utility function returns values close to 1 as the vehicle's speed approaches the safe speed. If the vehicle's speed diverges from the safe speed, then the utility function returns values that tend toward 0 in proportion to the size of the divergence. To derive this utility function, we identified the environmental property and its constraint as defined in the goal "Achieve[*VehicleSpeed* equal to *SafeSpeed*" in Figure 3. In this particular goal, *VehicleSpeed* is the environmental property that can be measured through the IVS's sensors, and *SafeSpeed* is the constraint that imposes a target value or threshold that the IVS must achieve. Safe speed is computed as follows:

$$\text{SafeSpeed} = \frac{\text{Dist}_{\text{LeadVehicle}}}{2.5\text{secs}}$$

where $\text{Dist}_{\text{LeadVehicle}}$ is the distance between the IVS and the Lead Vehicle. As such, this safe speed provides the IVS with 2.5 seconds of reaction time, as required by domain standards (i.e., domain knowledge, assumptions, and safety regulations).

D. Generating Environmental Conditions

Loki combines a genetic algorithm [12] with novelty search [11] to explore the impact of uncertainty upon the behavior of a DAS. As Figure 4 illustrates, each genome in the genetic algorithm comprises a vector whose length is equal to the number of sensors in the DAS's monitoring infrastructure. Furthermore, each sensor entry specifies the type, duration, and severity of environmental uncertainty that will be applied throughout the simulation. For this work, we support four types of environmental uncertainty that include static, periodic, and sporadic noise, as well as sensor failure. In addition, periodic and sporadic noise are applied to sensors for the duration specified in the time period parameter. The severity or degree of noise applied to each sensor, expressed as a floating-point value, is generated within a range of possible values for the given sensor, which is extracted from the object model (omitted due to space constraints). For example, the encoding shown in Figure 4 indicates that noise will be periodically applied to the IVS's top-right camera (cTR) for a duration of 20 simulation time steps and will affect approximately 16% of the image.

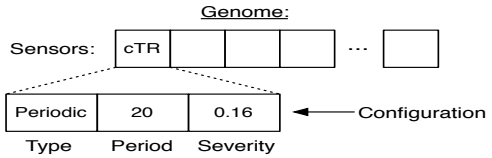

Figure 4. Example genome that specifies sensor noise configuration.

E. Simulation Configuration

Loki extracts the noise specification from the encoded genome (see Figure 4), uses it to configure the monitoring infrastructure of the DAS, and then executes the simulation. The simulation consists of two components. For the first component, the DAS probes its sensors to monitor its execution environment (e.g., detection of target vehicle and its speed); here, the sensor values are detecting simulator-generated random real-world conditions for a given scenario. Loki intercepts this monitoring information, which does not

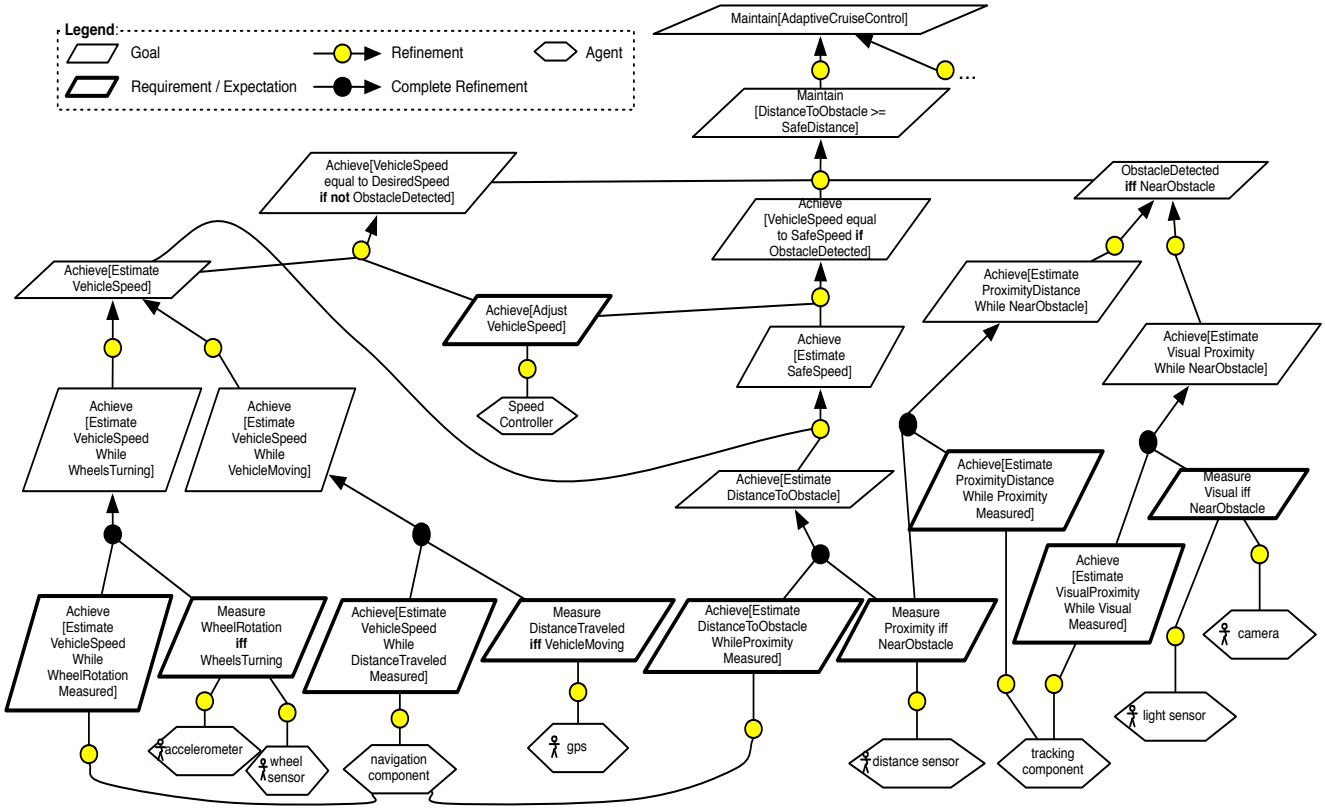

Figure 3. KAOS goal model specifying the goal refinements for adaptive cruise control.

contain "artificially"-inserted noise, before the DAS may process it. Instead, Loki instantiates the previously defined set of utility functions with this monitoring information to compute a set of "real" utility values. Each real utility value reflects how well the DAS satisfied one of its requirements at a specific point in time during the simulation. Note that real utility values differ between simulations depending on the random behavior generated by the simulator for the IVS (e.g., different distances to the target vehicle). These real utility values are stored in a vector of real utility values for further analysis once the simulation is complete.

For the second component, Loki applies noise to the raw sensor values that were intercepted from the DAS's monitoring infrastructure. Specifically, Loki substitutes the original monitoring values with noisy monitoring values that are intended to capture adverse environmental conditions, such as rain occluding the camera images, interference with the GPS signal, faulty distance sensor, etc. The DAS then instantiates the same set of utility functions with the noisy monitoring values to compute a set of "perceived" utility values. Each perceived utility value reflects the DAS's self-assessment of requirements satisfaction based on the noisy monitoring information, where adaptive reconfiguration decisions are made based on how well the requirements are being satisfied (or even violated). In this manner, the DAS performs decision-making based solely on noisy monitoring data. Note that perceived utility values differ between simulations depending on environmental conditions experienced by the IVS and how these affect the IVS's assessment of requirements satisfaction. Lastly, these perceived utility values are stored in a vector of perceived utility values for further analysis once the simulation is complete.

F. Novelty Computation

After the simulation completes, Loki calculates the distance between the utility values of each genome in the population and novelty archive. To compute the distance between the utility values of two genomes, Loki first matches the corresponding vectors of utility values. That is, the distance between real (perceived) utility vectors of a particular genome are evaluated only against the real (perceived) utility vectors of another genome. Loki then calculates the Manhattan distance [19] between each utility value in the vector. While a large divergence between real utility values implies two considerably different behaviors of a DAS, a large divergence between perceived utility values implies two considerably different self-assessments of requirements satisfaction by the DAS. Loki sums the distances between

each pair of vectors and assigns this value as overall the distance between the two genomes.

Loki then ranks the distances between each genome in the population in increasing order, and applies the following novelty metric to calculate the novelty value, $\rho(x)$, of each genome:

$$\rho(x) = \frac{1}{k}\sum_{i=0}^{k} \text{dist}(x, \mu_i)$$

where k (determined empirically) is the number of nearest neighbors that will be compared, x is the genome whose fitness is being assigned, μ_i is the i^{th} neighbor of x, and *dist* is a function that measures the distance between x and its neighbor. Moreover, genomes with the top 10% fitness values in the population are added to the novelty archive at each iteration or generation of the evolutionary algorithm. Once the evolutionary algorithm terminates, genomes in the novelty archive are returned as the set of environmental conditions that produced the most distinct set of behaviors in the DAS. As such, the novelty archive itself serves as a set of test cases for ensuring undesirable behaviors have been resolved.

IV. Experimental Results

In this section, we apply Loki to the IVS application. First, we describe the simulation scenario. We then present the experimental setup for conducting our evaluations. Next, we present and analyze experimental results. Lastly, we present a comparison between Loki and a randomized search algorithm used as a baseline.

A. Simulation Scenario

For this study, the simulation scenario comprises two autonomous vehicles, an IVS and a Lead Vehicle. Initially, the IVS is positioned 900 meters behind the Lead Vehicle in the same driving lane. Both vehicles begin to accelerate at the same time. Specifically, the Lead Vehicle accelerates until it achieves a desired speed of 35 km/h. The IVS, however, continues to accelerate until it achieves a desired speed of 55 km/h. As the IVS approaches the Lead Vehicle from behind, its sensors detect the obstacle and the IVS reconfigures its operational model to achieve and maintain a safe speed to avoid a potential collision. This safe speed prevents the IVS from crossing into the *safe distance zone* (see Figure 2). Shortly thereafter, the Lead Vehicle gradually accelerates until it reaches its new desired speed of 65 km/h, thus increasing its distance from the IVS and enabling the IVS to accelerate, once more, to its desired speed of 55 km/h.

Throughout this experiment, the same scenario is replayed in each simulation. However, different environmental conditions are applied in each simulation as specified by the sensor configurations encoded in the genomes that the genetic algorithm generates. Note that no sensor is considered to be a single point of failure in this scenario. That is, even after applying the maximum permissible amount of noise to each sensor independently, the IVS still continues to satisfy its requirements at run time. Therefore, the objective of this experiment is to discover sets of different latent behaviors and requirements violations in response to different combinations of environmental conditions.

B. Experimental Setup

Table I specifies the configuration of the genetic algorithm and novelty search for this experiment. With a population size of one hundred genomes, and a maximum number of ten generations, this particular configuration evaluates exactly 1000 different combinations of environmental conditions. A Manhattan distance metric is used to compute the difference between the utility vectors associated with genomes in the population and novelty archive. After ranking these distances, the novelty of a genome is assigned by computing the mean distance to the seven nearest genomes in the solution space. At the end of each generation, genomes with a novelty value in the top 10% are added to the novelty archive. Lastly, we conduct 25 trials of this experiment, each with a different seed value that is stored to ensure that results are reproducible.

Table I
Genetic algorithm and novelty search configurations.

Parameter Description	**Value**
Maximum number of generations	10
Population size	100
Mutation rate	0.1
Crossover rate	0.6
Distance metric	Manhattan distance
k-nearest	7
Archive threshold	Top 10%

C. Discovered Behaviors

In general, Loki consistently discovered different sets of behaviors. Specifically, out of 1000 evaluations that were performed in each trial, Loki discovered a mean of 142.42 different behaviors in response to different combinations of environmental conditions. In addition, for each trial, Loki discovered a minimum and maximum of 77 and 201 different behaviors, respectively. These results suggest that 15% of the genomes examined by Loki represented different behaviors. Within this set of behaviors, approximately 7.7% of them involvd a requirements violation. Upon closer inspection, *every* behavior that violated a requirement suffered from a "perception-reality" gap. That is, environmental conditions negatively impacted the monitoring capabilities of the IVS to the extent that it failed to detect conditions that would indicate a requirement violation. The other 92.3% of behaviors satisfied requirements at run time, though some of them exhibited latent behaviors. These results confirm the crucial role that environmental uncertainty plays in determining whether a DAS is able to satisfy its requirements or not.

Requirements Violations. We analyzed the set of behaviors that violated requirements in order to determine which combinations of environmental conditions had the most impact upon goal satisfaction. In particular, the most detrimental combinations of environmental conditions involved mutliple failures of sensors or environmental agents. Although slightly less severe, sensor noise was also detrimental to goal satisfaction. In both cases, a requirements violation occurred because goals were either unfulfilled or their computations unreliable, thereby preventing the IVS from correctly interpreting gathered monitoring data. In contrast, results suggest that periodic and sporadic noise were not as detrimental to goal satisfaction as these had to occur at very precise points in time (i.e., when the IVS is making a key decision) in order to obstruct a goal. Based on this information, a requirements engineer may begin to prioritize the identification and mitigation of obstacles such that sensor failures and sensor noise problems are addressed first.

Different environmental conditions affected three key agents the most: the cameras, distance sensors, and GPS. Of these three agents, the GPS was involved in a significant number of requirements violations. Specifically, the data produced by the GPS was susceptible to system and environmental noise, the slightest of which produced significant errors when computing the IVS's current speed. While the IVS is able to satisfy its requirements when uncertainty is limited to the GPS agent alone, if other agents in addition to the GPS failed or suffered from uncertainty, then the IVS no longer satisfied its requirements. This observation captures the dependency between the navigation component and the tracking component. As such, the goal *"Achieve[Estimate Accurate SafeSpeed]"* in Figure 3 was unsatisfied in the majority of these cases, thus leading to a requirements violation by failing to maintain a safe distance between the IVS and the Lead Vehicle.

To illustrate the range of different behaviors discovered by Loki, we now examine two different behaviors that caused a requirements violation. In both cases, the IVS was unable to accurately estimate the coasting zone distance due to moderate levels of sensor noise across the GPS and distance sensor agents. As a result, in the first behavior, the IVS failed to decelerate in time, crossed into the safe distance zone momentarily, and then, by further reducing its speed, re-entered the coasting zone. While the IVS and the Lead Vehicle did not collide as a result of this behavior, the "Maintain[SafeDistance]" requirement was ultimately violated. In the second behavior, the IVS also suffered from the failures of a camera and two distance sensor agents. However, in contrast to the previous example, the IVS now failed to decelerate in time, crossed into the safe distance zone, collided with the Lead Vehicle, departed from its driving lane temporarily because of the collision, and then continued to collide with the Lead Vehicle in order to re-enter the driving lane (in an attempt to satisfy the lane keeping goal), eventually pushing the Lead Vehicle off the road. While the first behavior suggests that a new mitigation should be inserted into the goal model to resolve obstacles that affect the accuracy of speed estimates, the second behavior suggests a subtle and undesirable requirements interaction between the ACC and lane keeping features.

Careful examination of the interactions between agents and goals involved in the different requirements violations discovered by Loki suggest that the root of this failure cascade was the inaccurate computation of the IVS's current velocity. Specifically, although the IVS can choose from two different OR-refinements to estimate its current velocity, if both branches are affected by detrimental environmental conditions, then the obstacle mitigation is insufficient. In either case, the current velocity estimate is also used to compute the safe speed and safe distance values in the IVS. While these inaccuracies may be independently insignificant, when combined they tended to accumulate and propagate across goals, thereby negatively impacting the decision-making capabilities of the IVS. More specifically, these behaviors indicate that the IVS was unable to decelerate in a timely manner, crossed into the safe distance zone, and violated one or more requirements, often resulting in a collision.

Unwanted Latent Behaviors. Most of the different behaviors discovered by Loki satisfied requirements. However, in several instances, Loki also discovered latent behaviors that should be disallowed. For example, in one latent behavior, the IVS decelerated and achieved its safe speed without crossing into the safe distance zone. However, the IVS then began to abruptly accelerate and decelerate in order to maintain its safe speed (i.e., causing a jerking motion). In a different latent behavior, the IVS abruptly decelerated just before crossing into the safe distance zone. However, instead of achieving its safe speed as in the previous behavior, the IVS continued decelerating almost to a complete stop (i.e., stopping in the middle of a highway) before eventually accelerating. In both cases, the latent behaviors discovered by Loki need to be prevented as they negatively impact passenger comfort and may cause a collision with other vehicles trailing the IVS, respectively.

D. Comparison to Randomized Search

We now *reuse* the environmental conditions and resulting utility vectors from the previous experiment in order to compare Loki with a randomized search algorithm that serves as a control.

Experimental Setup. This experiment leverages the same simulation scenario as in the previous experiment, but we replace the novelty search algorithm with a randomized search. We extend the previous experiment by randomly generating 1000 different sensor configurations that specify the type, duration, and severity of noise applied to each

sensor in the IVS during a simulation. We then execute one simulation for each sensor configuration in order to produce the corresponding vectors of utility values. Since each simulation is executed sequentially and independently, the randomized search algorithm has no knowledge about what areas of the solution space have already been explored. Once all simulations complete, we compute the differences between the utility values for each sensor configuration (i.e., the novelty value) by using the same novelty metric as in the previous experiment (i.e., k-nearest neighbors).

Comparing Novelty Values. Once we computed the novelty values of each randomly-generated sensor configuration, we compared them with the results generated by Loki from the previous experiment. Intuitively, the approach that produces a larger set of novelty values has covered a larger portion of the solution space.

In general, randomized search also discovered system and environmental conditions that produced requirements violations. Out of 1000 evaluations per trial, randomized search discovered a mean of 194.1 behaviors that involved a requirements violation. These results confirm that randomized search is a valuable tool for discovering test cases that trigger failures in the system-to-be. While it seems randomized search discovered more behaviors that violated requirements, upon closer inspection, most of the randomly-discovered behaviors were similar to each other. In particular, we found statistically significant differences in the novelty values between Loki and randomized search (Wilcoxon rank sum test, $p < 0.001$, Loki mean = 14.7, randomized search mean = 12.3). Moreover, Loki was able to consistently find larger novelty values than randomized search. For instance, in every trial, Loki discovered several behaviors with novelty values greater than 40, which are considerably larger than the mean values obtained by both approaches. Often, the magnitude of the novelty value correlates with the severity of latent behaviors and requirements violations.

V. Related Work

This section presents related work in obstacle analysis, requirements monitoring, and test case generation. First, we describe approaches for mitigating obstacles in goal-oriented requirements engineering. We then present related work in self-assessment within the context of a DAS. Lastly, we discuss approaches for generating test cases that can discover incomplete and inconsistent requirements.

A. Obstacle Analysis

Obstacle analysis enables the identification and resolution of conditions that can otherwise obstruct the satisfaction of a goal at run time. Van Lamsweerde and Letier proposed a set of heuristics, refinement patterns, and formal techniques for systematically identifying and mitigating obstacles from goal specifications [2], [3]. The authors also present a library of generic refinement patterns grounded within a formalized framework that can be applied to gradually refine and elaborate root goal negations into fine-grained obstacles. Once identified, several obstacle mitigations are possible. In a slightly different approach, Letier and Van Lamsweerde also proposed a probabilistic framework for specifying partial degrees of goal satisfaction [5]. In this approach, non-functional goals are specified in terms of probabilities, which can be obtained directly from stakeholders, derived from operational statistics, and so forth. The satisfaction of each non-functional goal is evaluated with application-specific refinement equations or objective functions. In addition, several heuristics are proposed for deriving objective functions from system functional goals.

Although it is ideal to identify and mitigate every possible obstacle that might obstruct goals in the system-to-be, achieving this objective for many domains may be practically infeasible given the large space of potential obstacles that might arise at run time. Loki both complements and supports the obstacle analysis approaches proposed by Van Lamsweerde and Letier [2], [3], [5]. Specifically, Loki discovers system and environmental conditions that produce latent behaviors and requirements violations. As such, Loki may be applied to augment the set of goals, obstacles, and obstacle mitigations identified through heuristics, refinement patterns, and formal techniques [2], [3]. Additionally, Loki may also be applied to validate and/or refine the probabilistic values assigned to each goal [5].

B. Requirements Monitoring and Self-Assessment

Requirements monitoring determines how well a system satisfies its requirements at run time, and detects conditions conducive to a requirements violation in order to mitigate them as early as possible. Fickas [20], Feather [21], and Robinson [22] developed requirements monitoring frameworks to support the instrumentation, gathering, and analysis of monitoring data. In the same spirit, utility functions have been used as light-weight approaches to perform self-assessment [14] and select which reconfiguration to apply at run time [13]. Within the context of a DAS, these utility functions accept monitoring data as input, apply domain-specific objective functions to process the data, and output a value within the range [0, 1] that is proportional to how well the system satisfies functional and non-functional requirements. Although Loki applies utility functions to determine if a DAS satisfies functional requirements under different system and environmental conditions, it can also leverage traditional requirements monitoring approaches such as those developed by Feather, Fickas, and Robinson. While we manually derive the utility functions for this work, Loki may also benefit from automated utility function derivation techniques [14].

C. Test Case Generation

Lutz and Mikulski identified several mechanisms for discovering and, if necessary, resolving requirements during the software testing phase [7]. The objective of these mechanisms is to resolve incomplete requirements, unexpected requirements interactions, and requirements confusions by testers. Loki supports these mechanisms for requirements discovery and resolution as it provides an automated approach for discovering incomplete requirements in the form of missing or inadequate obstacle mitigations and insufficiently constrained goals. Loki can also be used to identify unexpected requirements interactions in the form of latent behaviors that may be conducive to goal obstructions. In addition, research on software testing has focused, among other things, on optimizing test inputs such that the software or model is exercised as much as possible. For instance, Ledru *et al.* [23] proposed an approach that applies distance metrics to prioritize test cases that are most different from those already examined. Their proposed approach, however, explores test cases at the application code level, whereas Loki explores the impact of system and environmental conditions at the goal and requirements levels.

Recently, genetic algorithms have been applied to generate suites of test cases [8]–[10]. For instance, Nguyen *et al.* [10] proposed an approach for testing the behavior of an autonomous agent in response to, among other things, environmental conditions generated by a genetic algorithm. Nonetheless, to leverage these approaches, developers must extend the genetic algorithm with domain-specific functions to evaluate the quality of generated test cases. Furthermore, the fitness functions in these traditional genetic algorithm-based approaches are often defined to search for specific types of failures, thus making them less able to detect unwanted latent behaviors that the system also satisfies. In contrast, developers need not redefine the novelty function in order to use Loki across different application domains. Similarly, through novelty search, Loki is able to generate a more comprehensive suite of latent behaviors and requirements violations.

VI. Conclusions

In this paper we presented Loki, an evolutionary computation-based approach for automatically discovering environmental conditions that produce unexpected behaviors in a DAS, including goal obstructions and requirements violations. Loki applies novelty search to reduce the entire set of examined behaviors that a DAS might exhibit into a comprehensive set of different behaviors. We presented several examples of requirements violations and latent behaviors automatically discovered by Loki. Experimental results demonstrate that Loki is able to find a significantly larger number of different and unexpected behaviors in response to environmental conditions when compared to randomized testing. The set of environmental conditions that led to these undesirable behaviors can be leveraged not only to identify missing or inadequate obstacle mitigations in a goal model, but also as test cases when revising the implementation of the DAS.

A requirements engineer may apply Loki in several ways, depending on the kinds of behaviors being explored. In particular, the k-nearest parameter in the novelty search algorithm directly affects the fitness value computation by controlling how many other behaviors are considered when establishing clusters of different behaviors. As a result, setting low (high) values for the k-nearest parameter in novelty search causes Loki to discover a larger (smaller) number of different behaviors. In this manner, a requirements engineer who wishes to initially explore a small subset of behaviors might prefer a larger k-nearest parameter in order to generalize solutions. Once a few interesting behaviors are identified, the requirements engineer might wish to explore a broader set of behaviors by reducing the k-nearest value. Approaches such as the one proposed by Nguyen *et al.* [10] might be leveraged at this point to focus on refining a specific behavior of interest.

Future directions for this work include applying Loki to explore how environmental conditions may affect RELAXed goals [6]. In addition, we also plan to explore how Loki can be used with the partial goal satisfaction framework by Letier and Van Lamsweerde [5] in order to refine initial probabilities assigned to goals.

VII. Acknowledgements

We gratefully acknowledge James Gung's help in constructing earlier versions of the goal models for the IVS application.

This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, DBI-0939454, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company, and a Quality Fund Program grant from Michigan State 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 the National Science Foundation, Army, Ford, or other research sponsors.

References

[1] J. Whittle, P. Sawyer, N. Bencomo, Betty H.C. Cheng, and J.-M. Bruel, "RELAX: Incorporating uncertainty into the specification of self-adaptive systems," in *In the Proceedings of the 17th International Requirements Engineering Conference (RE '09)*. Atlanta, Georgia, USA: IEEE Computer Society, September 2009, pp. 79–88.

[2] A. van Lamsweerde and E. Letier, "Integrating obstacles in goal-driven requirements engineering," in *Proceedings of the 20th International Conference on Software Engineering*. Kyoto, Japan: IEEE Computer Society, 1998, pp. 53–62.

[3] ——, "Handling obstacles in goal-oriented requirements engineering," *IEEE Transactions on Software Engineering*, vol. 26, no. 10, pp. 978–1005, October 2000.

[4] R. Darimont and A. van Lamsweerde, "Formal refinement patterns for goal-driven requirements elaboration," *SIGSOFT Software Engineering Notes*, vol. 21, no. 6, pp. 179–190, October 1996.

[5] E. Letier and A. van Lamsweerde, "Reasoning about partial goal satisfaction for requirements and design engineering," in *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Newport Beach, California: ACM, 2004, pp. 53–62.

[6] Betty 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 *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, ser. Lecture Notes in Computer Science. Denver, Colorado, USA: Springer-Verlag, October 2009, pp. 468–483.

[7] R. R. Lutz and I. C. Mikulski, "Requirements discovery during the testing of safety-critical software," in *Proceedings of the 25th International Conference on Software Engineering*. Portland, OR, USA: IEEE Computer Society, 2003, pp. 578–583.

[8] J. H. Andrews, T. Menzies, and F. C. Li, "Genetic algorithms for randomized unit testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 80–94, January 2011.

[9] 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*. San Diego, California, United States: ACM, 2000, pp. 136–140.

[10] C. D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman, and M. Luck, "Evolutionary testing of autonomous software agents," in *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*. Budapest, Hungary: International Foundation for Autonomous Agents and Multiagent Systems, May 2009, pp. 521–528.

[11] 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)*. Cambridge, MA, USA: MIT Press, 2008.

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

[13] S. W. Cheng, D. Garlan, and B. Schmerl, "Architecture-based self-adaptation in the presence of multiple objectives," in *Proceedings of the 2006 International Workshop on Self-adaptation and Self-Managing Systems*. Shanghai, China: ACM, 2006, pp. 2–8.

[14] P. de Grandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *In the Proceedings of the Sixth International Conference on Autonomic Computing (ICAC'09)*. Barcelona, Spain: ACM, June 2009, pp. 107–116.

[15] O. Michel, "Webots: Professional mobile robot simulation," *Journal of Advanced Robotics Systems*, vol. 1, no. 1, pp. 39–42, 2004.

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

[17] E. S. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *Proceedings of the Third IEEE International Symposium on Requirements Engineering*. Annapolis, MD, USA: IEEE Computer Society, January 1997, pp. 226–235.

[18] A. Dardenne, A. van Lamsweerde, and S. Fickas, "Goal-directed requirements acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[19] P. E. Black, *Dictionary of Algorithms and Data Structures*. U.S. National Institute of Standards and Technology, May 2006.

[20] S. Fickas and M. S. Feather, "Requirements monitoring in dynamic environments," in *RE '95: Proceedings of the Second IEEE International Symposium on Requirements Engineering*. Washington, DC, USA: IEEE Computer Society, 1995, pp. 140–147.

[21] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard, "Reconciling system requirements and runtime behavior," in *IWSSD '98: Proceedings of the 8th International Workshop on Software Specification and Design*. Washington, DC, USA: IEEE Computer Society, 1998, pp. 50–59.

[22] W. N. Robinson, "Monitoring software requirements using instrumented code," in *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences*. Hawaii, USA: IEEE Computer Society, 2002, pp. 276–285.

[23] 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*, ser. ASE'09. Auckland, New Zealand: IEEE Computer Society, November 2009, pp. 510–514.