Triggering Adaptation via Contextual Metamorphic Relations

Byron DeVries and Erik M. Fredericks
School of Computing
Grand Valley State University
Allendale, Michigan, USA
devrieby@gvsu.edu, frederer@gvsu.edu

Abstract—Verifying system behavior before deployment is a necessity, especially with self-adaptive software. However, exhaustive verification is impractical due to both the scale of the input/output space and uncertainties that may not be captured in pre-deployment testing. Run-time monitors have been used to measure operational health after deployment but are unlikely to have a one-to-one correspondence with pre-deployment testing. Problematically, adaptation strategies can be defined for managing adverse run-time measurements due to run-time monitors, however, it is infeasible to provide adaptation triggers for untested behaviors with only static pre-deployment tests. This paper introduces *contextual* metamorphic relations that are applicable to specific contexts of a system's behavior and implementation artifacts. These contextual metamorphic relations are a subset of general metamorphic relations and can be used for both pre-deployment testing and post-deployment run-time monitoring to trigger context specific adaptations in order to improve tolerance to specific faults. We illustrate our approach by triggering adaptation at run-time using contextual metamorphic relations for two real-world inspired responsive cyber-physical systems: a robotic drone system and a set of proximity-enabled responsive systems.

Keywords–metamorphic relation; monitoring; adaptation; self-adaptive; run-time; cyber-physical

1. Introduction

Verifying programs that include or deal with uncertainty remains a difficult proposition [1]. Testing is still an activity that typically occurs before deployment when the impacts of uncertainty are largely unknown and enumeration of all possible test cases is not possible [2]–[5] and is prohibitively difficult to automatically define expected results [6] (i.e., the oracle problem [7]). In this paper we propose a method of detecting run-time system failures without statically defined expected results. We detect these failures via metamorphic relations [6] extended to be used as contextual run-time monitors and adaptation triggers by exploiting relations in system results over time rather than assessing results based on a property measured at a single discrete time for known expected results.

A fixed set of pre-deployment tests is insufficient to ensure desired behavior at run-time, especially in the face of run-time faults [8]. Existing work has used various search-based methods to identify unique [9]–[11] or even adverse [12] behaviors. However, pre-deployment techniques are inherently limited to the implicit assumptions made when formulating the search space and methods. Worse, unique behaviors may not be *adverse* behaviors requiring manual analysis of the set of behaviors found. While metamorphic relations alleviate aspects of the oracle problem [13], they are typically applied to a single function of the application for which the metamorphic relation is valid. Run-time monitors can measure operational health after deployment and trigger adaptation, but useful run-time monitors may be difficult to create due to the oracle problem.

A metamorphic relation measures the correctness of a result when two different inputs to a function produce the same output or two different inputs to a function produce a known relation between the outputs (e.g., one output is less than the other). However, metamorphic relations only apply to the function the relation is based on and not the full set of system behaviors. This paper introduces *contextual* metamorphic relations that are evaluated only within the contexts the relation is expected to hold. That is, the contextual metamorphic relation only measures failure when the context (C) of the metamorphic relation (MR) is applicable:

$$C \implies MR \tag{1}$$

Contextual metamorphic relations alleviate aspects of the oracle problem, can be applied to an entire system at run-time, and can trigger context-specific adaptations due to their run-time assessment.

The contributions of this paper are as follows:

- We introduce *contextual* metamorphic relations,
- We introduce how *contextual* metamorphic relations can trigger adaptation, and
- We illustrate both *contextual* metamorphic relations and context-specific adaptation on a real-world inspired robotic drone system.
- We discuss the general applicability of *contextual* metamorphic relations via an additional example in proximity-enabled response systems.

The remainder of this paper is organized as follows. Section 2 and Section 3 provide an overview of the background information and the approach with a running example, respectively. Section 4 describes our results in several simulated scenarios. We discuss the more general applicability of this approach to a wider variety of problems in Section 5 by discussing proximity-enabled response systems. Related work is described in Section 6 while Section 7 discusses conclusions and future work.

2. BACKGROUND

This section covers background in metamorphic relations, our example robotic system, and a generic proximity-enabled responsive system.

2.1 Metamorphic Relations

Unlike traditional pre-deployment tests that input data into the software and compare the output with a predetermined expected result, metamorphic testing utilizes relations between two behaviors. When two distinct inputs yield identical results, the metamorphic relation is deemed input-driven. Conversely, when two distinct inputs produce different outcomes with a recognized change (such as one result being greater than the other), the metamorphic relation is categorized as output-driven. While metamorphic relations were initially employed for test generation [14], they have since been expanded as a strategy for addressing the oracle problem [13].

Facilitated by metamorphic relations, metamorphic testing allows for testing and test case generation without relying on an oracle. Take, for instance, the classic example of an algorithm designed to determine the shortest path between two points in a graph [13] (denoted as $A(G, s, e)$, where A represents the algorithm, G is the graph, and s and e are the start and end points, respectively). An input-based metamorphic relation would assert that the function A consistently returns the same value regardless of how the two points are ordered. In other words, the metamorphic relation ensures that the calculated distances (i.e., absolute values of the results) remain unchanged even if the start (s) and end (e) points are swapped, as shown in the following equation:

$$|A(G, s, e)| = |A(G, e, s)|. \tag{2}$$

Equation 2 represents an input-based metamorphic relation that we can use to verify the outputs for two distinct inputs based on their expected relation (e.g., that their output is equal). However, even when multiple sets of two points are selected to generate another set of tests that can be verified using the metamorphic relation, it does not guarantee that the program is correct. It only guarantees that the metamorphic relation was not violated for the inputs used in the test cases defined and additional verification may necessitate additional test cases.

2.2 Robotic Drone System

The robotic system used in this paper is a drone represented as a single sphere, comprising an infrared sensor, light, and a powered propeller extending from the top as shown in Figure 1. The drone is inspired by an inexpensive toy drone used by the child of one of the authors. Issues of unintended behavior based on uncertainty have been experienced by both the child and one of the authors and are explored in more depth in Section 4.

The infrared sensor and light both are placed at the bottom of the drone, facing downward. The drone creates infrared light via its light source to be used as a distance measure/object detection strategy. The infrared sensor detects light reflected from surfaces within range. When infrared light is read by the sensor the drone increases throttle to prevent collision with the object below it. When infrared light is not read, the drone reduces its throttle to descend towards an object below it (e.g., the floor). This robotic system is similar to existing real-world commercial systems used as toys by a variety of manufactures. While only vertical positioning can be affected, the drone can be "chased" upward by moving an object or hand underneath it and "caught" as it continues downward to increase its vertical position. The lateral position is entirely dependent on random perturbations that cause a swing of the ball or environmental factors (e.g., wind). Problematically, environmental uncertainty and uncertainty of the physical robot itself is known to cause issues. The authors have had personal experience with an out of control drone that would not return to the ground under its own control. For example, the drone would simply fly away in outdoor environments.

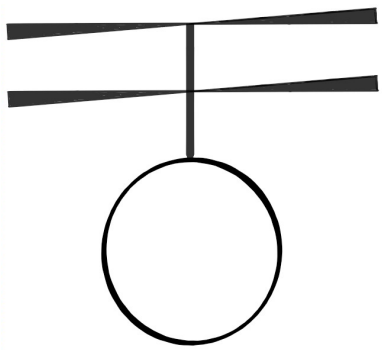

Figure 1. Infrared Ball Drone

Although a simple controller would be advantageous (e.g., a PID controller [15]) for both ascent and descent, proportional feedback of the error is only available within the range of the reflected infrared. That is, only ascent can be based on a proportional error of the reflected infrared. When outside of the reflected infrared range the drone must descend without external information on position until within the reflected infrared range.

2.3 Proximity-Enabled Responsive System

The general proximity-enabled responsive system used in this paper is represented in Figure 2. Starting in the left-most state (i.e., "`Wait`") the system waits until a time passes a specific constant (c_{wait}) until attempting to sense the proximity of a nearby object in the "`Sense`" state. If no proximity is sensed, the system returns to the wait. If proximity is sensed, the system will react in the "`React`" state by performing its operation until a constant time (c_{react}) has passed. The system continues this cycle indefinitely. Once the system reacts, it returns to the "`Wait`" state for a constant time (c_{wait}) before re-sensing for proximity.

Many systems follow this pattern. For example, automatic doors sense proximity and open for a specific amount of time and then close until proximity is sensed. Hands-free faucets in public restrooms detect the proximity of hands and run

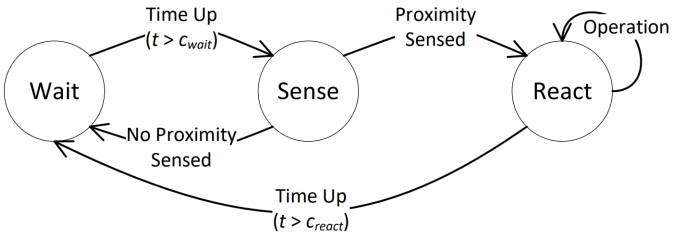

Figure 2. State Diagram of Proximity-Enabled Responsive System

the water for a specific time and then re-sense for proximity. Motion-activated lights sense proximity via motion and turn on for a constant time. In each case, there is a waiting period before proximity can be re-detected. This period is typically short for motion-activated lights, but sufficient enough for the lights to noticeably turn off and back on. Automatic doors, similarly, will start to close and re-open before closing entirely if proximity is detected. Hands-free faucets often include the longest delay of the three resulting in a noticeable wait between water resumes flowing from the faucet after stopping.

3. Approach and Running Example

This section details the steps that comprise our approach along with a running example. First, the system designer manually identifies metamorphic relations. Once applicable metamorphic relations are defined, the context in which the metamorphic relations is applied to create contextual metamorphic relations in the second step. While contextual metamorphic relations could be used for testing pre-deployment, our third step is to define contextual adaptations for detection of run-time failures. In our fourth step we implement our adaptations using a MAPE-K loop [16] via **m**onitoring, **a**nalyzing, **p**lanning and **e**xecuting based on a set of **k**nowledge. Each step, along with a running example based on the robotic drone system defined in the background (Section 2) is detailed below. Finally, limitations are discussed.

3.1 Step (1): Identify MRs

In general, identifying metamorphic relations is a domain-dependent activity [13]. For domain experts identifying relations is not difficult and are most typically identified in an "ad hoc and arbitrary way" [13]. Automatically identified relations [17], [18], however, are more challenging and considered an open research problem [13]. An additional complication for *contextual* metamorphic relations that are used at run time is that it is not possible to choose any inputs for the relation. Instead, the outputs to be compared via the relation must be created based on run-time values during execution. That is, the inputs must occur naturally during execution.

In the case of our running example based on the robotic drone system described in the background (Section 2), there are two distinct modes: increased thrust for ascent and decreased thrust for descent. While ascent may increase and decrease thrust based on reflected infrared (LM), the normal behavior in this mode is to ascend. Two metamorphic relations, one for each mode, are:

- **Ascent Mode**: The reflected infrared should decrease ($\Delta LM \leq 0.0$), due to ascent, or slow its increase ($\Delta\Delta LM \leq 0.0$) measured as a change in acceleration (i.e., jerk or jolt), due to additional thrust to reverse descent.
- **Descent Mode**: The reflected infrared should increase by going from no reflected infrared ($LM = 0.0$) to some amount ($\Delta LM > 0.0$), which would also cause a switch to Ascent Mode, due to an eventual descent.

In the case of each metamorphic relation, the values of the reflected infrared are measured based on at least two applications of the system within the same mode. While metamorphic relations may be described formally, in practice they are described "typically in an informal manner using natural language," [19] as those above. The *Descent Mode* metamorphic relation, for example, requires not just a first- or second-order derivative like the *Ascent Mode* metamorphic relation, but a maximum time before some reflected infrared would re-appear due to a maximum expected operating height and thus a maximum descent time. Worse, a formula-based metamorphic relation (e.g., with derivatives) would not match the discretized version implemented in code. In the remainder of this paper, these two metamorphic relations will be referred to as MR_{ascent} and MR_{descent} for the *Ascent Mode* ($\Delta LM \leq 0.0 \wedge \Delta\Delta LM \leq 0.0$) and *Descent Mode* ($LM = 0.0 \vee \Delta LM > 0.0$) metamorphic relations, respectively.

3.2 Step (2): Identify Contextual MRs

Metamorphic relations are limited to the function over which the relation compares, the path(s) in code over which the function executes when producing comparable output, and range of values applicable to the relation. Some metamorphic relations cover a wider proportion of a system's functionality, while others are more constrained. Consider, for example, the shortest path algorithm described in the background (Section 2). While a metamorphic relation that switches the input for the output ensures that the starting and ending locations do not result in differing output, it is also possible that the same incorrect (i.e., non-optimal) result is returned. A more robust metamorphic relation would choose a third point and ensure that traversing from the start to the new point and finally to the end point would always be greater (i.e., in cases where the point is selected off the optimal start to end path) or equal (i.e., in cases where the point is selected on the start to end path). That is, the output-based metamorphic relation indicates the distance (i.e., absolute value of the results) are greater or equal for a path from start (s) to the middle (m) to the end (e) when compared to just the start to end (s to e):

$$|A(G, s, m)| + |A(G, m, e)| \geq |A(G, s, e)|. \quad (3)$$

This difference in the ability of a metamorphic relation to detect errors extends beyond the relation itself to when the inputs applicable to the relation are used. For example, a shortest path algorithm is only used at specific times within the context of an entire digital map application and a metamorphic

relation related to the shortest path would only be applicable when the shortest path algorithm was in use. The *context* of the metamorphic relation in Equation 3 is when the shortest path algorithm is called.

Similarly, the metamorphic relations defined for our robotic drone system are only applicable in the modes they are defined for. That is, the MR_{ascent} metamorphic relation is only contextually applicable in the *Ascent Mode* while the MR_{descent} metamorphic relation is only contextually applicable in the *Descent Mode*. Since descent mode is applicable whenever the *last* infrared measurement was non-existent, the contexts of MR_{descent} and MR_{ascent} can be defined as $LM_{t-1} = 0$ and $\neg(LM_{t-1} = 0)$, respectively, where LM is infrared light measured and t is time.

The *contextual* metamorphic relations are:

$$(LM_{t-1} = 0) \implies MR_{\text{descent}} \tag{4}$$

$$\neg(LM_{t-1} = 0) \implies MR_{\text{ascent}} \tag{5}$$

Importantly, the *contextual* metamorphic relations only fail if they are contextually relevant *and* in violation of the metamorphic relation. The applicability of the set of contexts (C) can be measured for any specific scenario or even completeness against unsatisfied scenarios when considering all contexts:

$$\bigvee_{i=1}^{|C|} C_i, \tag{6}$$

though just because a metamorphic relation is contextually relevant the relation may not be able to verify all aspects of that functionality (e.g., the first version of the shortest path algorithm metamorphic relation). If any contexts may be satisfied simultaneously, they must be prioritized in order to determine the run-time adaptation. In the case of our *contextual* metamorphic relations, there is no overlap.

Just as metamorphic relations can be used to generate pre-deployment tests, *contextual* metamorphic relations can be used for that purpose as well. Typically, a stochastic search-based method is used to optimize for failures or novelty in the metamorphic relation [12]. *Contextual* metamorphic relations are a subset of metamorphic relations that are more constrained and existing methodologies can be applied pre-deployment. In this paper, however, we focus on triggering run-time adaptation.

3.3 Step (3): Define Contextual Adaptations

A self-adaptive system should adapt when it encounters a failure. Similarly, when a *contextual* metamorphic relation fails, a contextually appropriate adaptation should take place. Adaptations in our approach, just like metamorphic relations, are manually defined and domain specific.

In the case of the *contextual* metamorphic relation defined in Step 2, the following are selected as adaptations from previous behavior:

- $(LM_{t-1} = 0) \implies MR_{\text{descent}}$ fails:
  - Increase thrust to slow descent, and
  - Periodically increase thrust beyond gravitational pull.
- $\neg(LM_{t-1} = 0) \implies MR_{\text{ascent}}$ fails:
  - Revert to descent behavior.

The adaptations for each context are intended to ensure the robotic drone system does not ascend uncontrollably and an unexpectedly long descent does not gain too much velocity. Both cases could be dangerous as high-velocity descent could cause harm and, once the battery was exhausted, uncontrolled ascent would eventually be uncontrolled descent.

3.4 Step (4): MAPE-K Implementation

In this step we use the *contextual* metamorphic relation and their adaptations within a self-adaptive system MAPE-K loop [16] to adapt for runtime fault tolerance. In the following subsections we describe how we employ the monitor, analysis, plan, and execute steps of the MAPE-K loop and our knowledge definition.

3.4.1 Monitor

Consists of reading and storing sensor (external) and system (internal) values for the *contextual* metamorphic relations in order to support the assessment of the relations. For example, our drone system may store previous infrared light measurements (external) and commanded thrust values (internal).

3.4.2 Analyze

Each *contextual* metamorphic relation is assessed, based on the previously stored values. If a relation is contextually appropriate the metamorphic relation portion of the *contextual* metamorphic relation is assessed.

3.4.3 Plan

In the event of a *contextual* metamorphic relation failure, a pre-defined adaptation is selected. In the event of multiple *contextual* metamorphic relation failures adaptations are selected based on pre-defined priorities between the relations.

3.4.4 Execute

Given a single adaptation, the system modifies its behavior to the new behavior defined by the adaptation. Non-adapted behavior is reinstated when the context of the *contextual* metamorphic relation is no longer applicable.

3.4.5 Knowledge

In our case knowledge of the application domain is encoded in the *contextual* metamorphic relation and their adaptations as documented in the system documentation and implemented in software.

3.5 Limitations and Threats to Validity

Our approach is not without known limitations and threats to validity, including the correctness of the contextual metamorphic relations themselves. Specific limitations and threats to validity for contextual metamorphic relations are as follows. First, completeness is limited to scenarios that are visible to the system due to the system only having access to the "shared" actions in the environment. An error could be based on "unshared" actions that are undetectable by the system

due to the set of sensors available [20]. Any system that interacts with an external environment is likely to include this limitation.

Second, both metamorphic relations and the adaptations are defined manually. Based on their manual definition and the accessibility of "unshared" environmental information there is no guarantee that all scenarios have a contextually relevant metamorphic relation that can verify each scenario. Additional future work is planned related to coverage and completeness of *contextual* metamorphic relations, including relations based on both system and environmental monitoring [8].

Third, *contextual* metamorphic relations must be defined over differing inputs that occur while the system is running, based on the number of results related within the relation. In the case of the *contextual* metamorphic relations defined in this paper, two different values in time are used. Standard metamorphic relations can be defined for a wider variety of different inputs, rather than just those that change over time since they are not used at run time. While we acknowledge these limitations, our approach is not negatively impacted in situations where the set of sensors is fixed in an existing system and appropriate metamorphic relations exist over time.

Finally, each metamorphic relation has limited fault detection capabilities, and some are more limited than others (e.g., the source/destination reversal in the metamorphic relation in Equation 2 is less effective than adding a mid-point in Equation 3). *Contextual* metamorphic relations are similarly limited while also being limited to only the scenarios where they are *contextually* relevant. Assessing *contextual* metamorphic relation fault detection capabilities is targeted in future work.

4. Results

In this section we apply a simulation of several run-time scenarios that exercise the system behavior, contextual metamorphic relations, and adaptations of our robotic system. Each scenario includes a surface (e.g., a hand) that reflects infrared light and starts at a height of zero and increases to one meter over the course of a second, then remains at one meter. The initial height of the drone varies depending on the scenario. Please note that the simulation simplifies specific physical aspects, including wind resistance and noise and operates over 100 iterations per simulated second.

4.1 Non-Adaptive Flight

We demonstrate two non-adaptive flights from different heights: 5 meters and 0 meters. Figure 3 shows the height of the reflective surface (black line), the height of the robotic drone system with a 5 meter initial height (black dots), and the height of the robotic drone system with a 0 meter initial height (black dashes).

While the robotic drone system that starts from the ground (dashed line) tracks well with the infrared reflection surface and follows above it due to the infrared reflection, the drone that starts at a higher position (5 meters) fails to adjust its downward speed without any indication of its position relative to the surface reflecting the infrared light until it is within

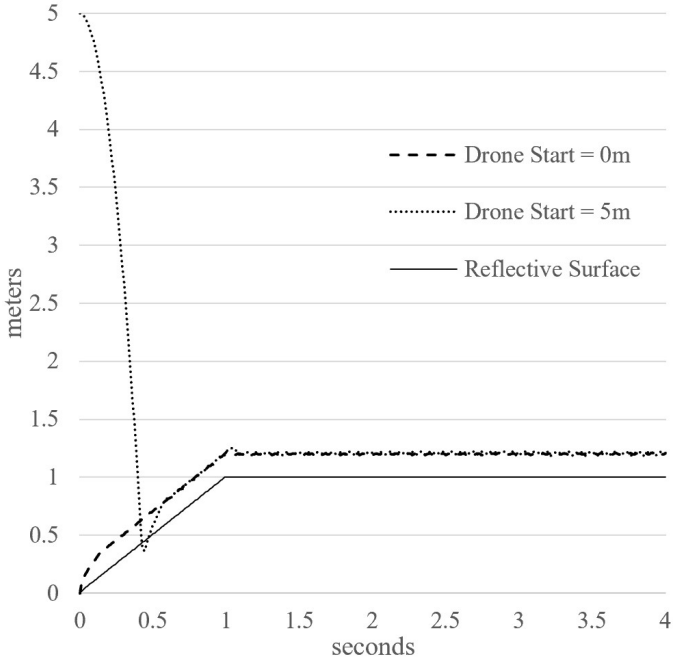

Figure 3. Non-Adaptive Flight from 0 and 5 Meters

range of the reflectively. Worse, we can see that at roughly half a second it goes lower than the infrared reflection surface (i.e., collides with) when it should hover above the surface.

4.2 Uncontrolled Ascent

Uncontrolled ascent may occur with a non-adaptive system in two cases. First, a sensor fault leading to a "stuck on" value indicating continual infrared light reflection, even when no reflective surface nearby. Second, a high amount of natural light can provide infrared light to the sensor that is not reflected from the light on the robotic drone system itself. Each of these cases are examples of expressed uncertainty that could impact the controls of the robotic drone system. Figure 4 shows the height of the reflective surface (black line), the height of the non-adaptive robotic drone system with a 0 meter initial height (black dashes), and the height of an adaptive robotic drone system with a 0 meter initial height (black dots).

Uncontrolled ascent is another impact of uncertainty on our robotic system with a significant effect on both continued use of the system, especially when outside, and subsequent safe return to the ground. When comparing the non-adaptive and adaptive systems in Figure 4 both are unable to read anything but an erroneous large infrared reflection. The non-adaptive robotic drone system continually attempts to fly out of the infrared reflection range. The adaptive system, however, contextually fails the metamorphic relation and adapts to a lower amount of thrust interspersed with periodic increases (causing one second long hops at 1 and 3 seconds). While the adaptive version is not nearly as compelling as a correctly functioning drone, it is at least minimally interactive and *significantly* safer than a continuously ascending drone that

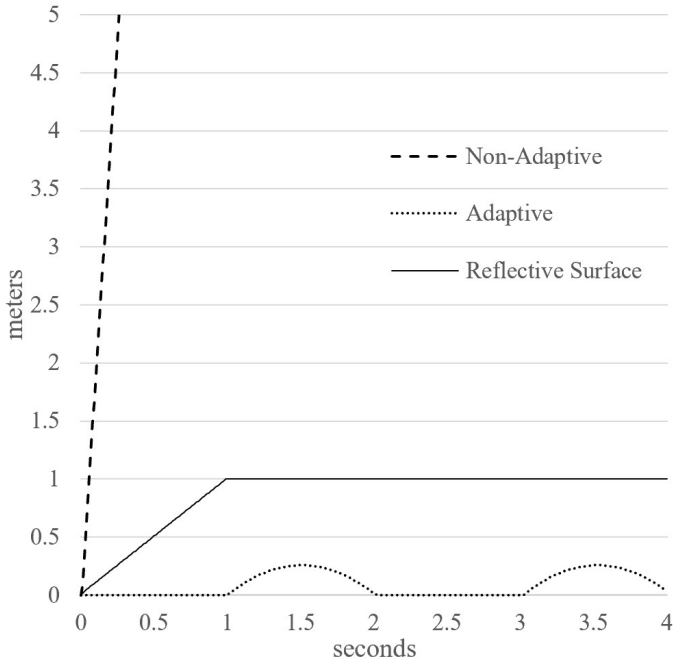

Figure 4. Results for Continuous Infrared Light

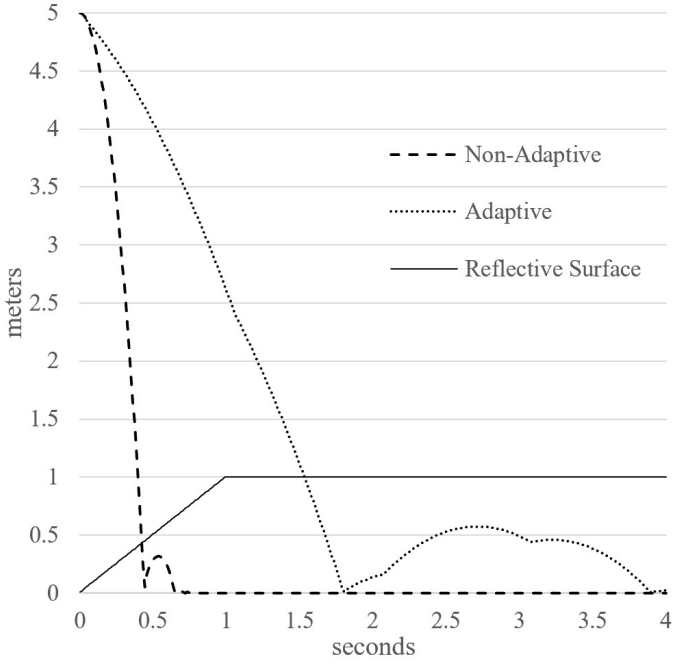

Figure 5. Results for Non-Existent Infrared Light

disappears only to fall at great speed somewhere unexpected when the battery is exhausted.

4.3 Uncontrolled Descent

Uncontrolled descent may occur when a sensor fault leads to a "stuck off" value indicating no infrared light reflection, even when a reflective surface is nearby. This case is an example of expressed uncertainty that could impact the controls of the robotic drone system. Figure 5 shows the height of the reflective surface (black line), the height of the non-adaptive robotic drone system with a 5 meter initial height (black dashes), and the height of an adaptive robotic drone system with a 5 meter initial height (black dots).

A lack of sensed infrared light leads to a less problematic scenario than watching your drone fly away into the clouds, but still results in undesired behavior. The non-adaptive version of the robotic drone system simply reduces thrust to its minimal level searching for reflected infrared light. However, since no infrared light will be detected, the drone continues into uncontrolled descent. In the descent from 5 meters the drone reaches a maximum velocity of just under 50 miles per hour (49.2mph). The adaptive version, after some amount of time not seeing an increase in infrared light compared to previous iterations, adapts to a higher thrust that still allows for descent with periodic increases in thrust. The adaptive version, when dropped from 5 meters reaches a maximum velocity of just under 9 miles per hour (8.9mph).

4.4 Adaptive Flight

Finally, we demonstrate two adaptive flights from different heights: 5 meters and 0 meters. Figure 6 shows the height of the reflective surface (black line), the height of the robotic drone system with a 5 meter initial height (black dots), and the height of the robotic drone system with a 0 meter initial height (black dashes).

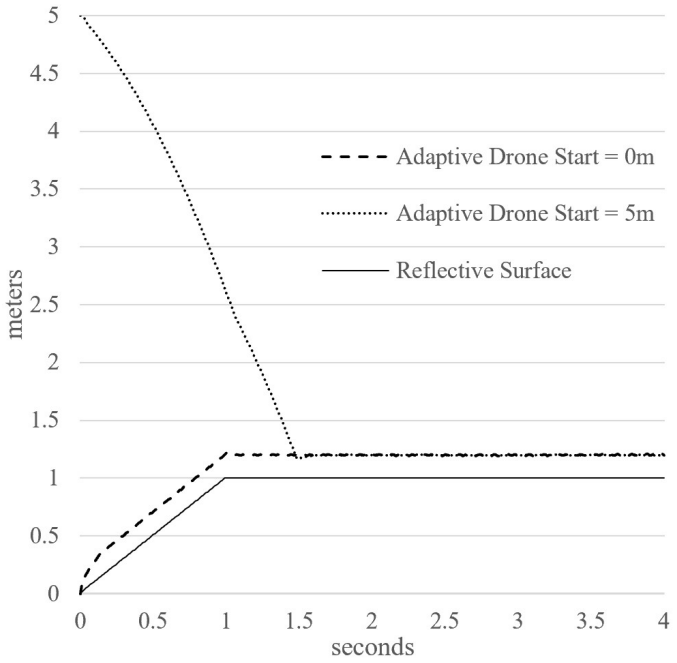

Figure 6. Adaptive Flight from 0 and 5 Meters

In comparison to the previous non-adaptive flights in Figure 3 without sensor or environmental uncertainty, the flights in Figure 6 include the *contextual* metamorphic relations and associated adaptations. While the flight starting at 0 meters behaves the same as before, when starting at 5 meters (about

16.5 feet, the rough equivalent of being dropped out of a second floor window) the descent mode *contextual* metamorphic relation fails and the descent adaptation is enabled. Not only does this reduce the maximum speed of the fall from 47 miles per hour to just under 8 miles per hour (7.8mph), but the transition to following the infrared reflecting surface at roughly 1.5 seconds no longer passes below (i.e., collides with) the reflecting surface, as it did in Figure 3.

4.5 Results Overview

Toy drones, like the one that inspired the robotic drone system in this paper, are not designed for robust behavior. However, the default behavior in the face of uncertainty provides much to be desired. The application of two computationally simple *contextual* metamorphic relations and associated contextual adaptations covered the two major functional modes of this simple robotic drone system and mitigated negative functionality *without* a priori knowledge of the expected results.

5. Applicability

While the drone system described in the previous two sections can make use of *contextual* metamorphic relations to address sensor faults that would cause dangerous scenarios, it is only a single case study. In this section we illustrate the use of *contextual* metamorphic relations on a class of proximity-enabled responsive systems described in the background section (Section 2).

Mirroring our approach (Section 3, we manually identify our metamorphic relations. Then we identify our context to create contextual metamorphic relations and define contextual adaptations to tolerate detected faults detected at run-time. Finally, we implement our adaptations via a MAPE-K loop [16].

5.1 Step (1): Identify MRs

Given the state diagram in Figure 2, we would expect that all states (i.e., "`Wait`", "`Sense`", and "`React`") would be visited over time. That is, the operation should be performed given some sensed proximity (e.g., opening an automatic door for an automatic door system) and that there should be times when no proximity is detected and the door stays closed while in the "`Wait`" state.

A standard pre-deployment test can verify that the operation is performed when proximity is sensed. However, a sensor failure or aberrant environmental scenario may cause undesired behavior due to the fault. For example, if the sensor is damaged or someone stands in front of the automatic door system without entering, then the operation will continue to be performed and the "`Wait`" state will no longer be reachable. The door, in this example, would never fully close for any period of time.

An output-based metamorphic relation would compare the outputs over a time period larger than the operation time frame (i.e., c_{react} in Figure 2) and find they are different, as shown in the following equation:

$$\text{Operation}_t \neq \text{Operation}_{t+c_{react}}, \tag{7}$$

where "Operation" is the state of the system's operation (e.g., opening, closing, open, or closed for an automatic door) and t is a current time.

5.2 Step (2): Identify Contextual MRs

Problematically, the metamorphic relation in Equation 7 is not always valid, as a metamorphic relation should be. For example, if no proximity is sensed the operation will continue to be the same and the metamorphic relation is not valid or applicable. Given the system and environmental context we must add a *contextual* component to create a *contextual* metamorphic relation.

However, the metamorphic relation in Equation 7 is valid for a time period of c_{react} after proximity has been sensed. That is, while the equation:

$$0 < t - t_{proximity} < c_{react} + c_{wait}, \tag{8}$$

where t is the current time, $t_{proximity}$ is the time proximity was sensed, and $c_{react} + c_{wait}$ is the time for the system to perform the operation and wait until the operation can be performed again. The *contextual* metamorphic relation, as defined by the equations above is:

$$\begin{aligned} 0 < t - t_{proximity} < c_{react} \implies \\ \text{Operation}_t \neq \text{Operation}_{t+c_{react}+c_{wait}}. \end{aligned} \tag{9}$$

5.3 Step (3): Define Contextual Adaptations

Based on the contextual metamorphic relation in Equation 9, several adaptations are possible when a fault is detected:

- The c_{wait} time can be decreased to allow for re-engagement of the operation more quickly. For example, the water in a motion-activated faucet could turn on again sooner if it is re-engaged directly after the wait when it turns off.
- The c_{react} time can be increased to allow the operation to take longer. For example, if an automatic door is not open long enough for people to walk through it would automatically re-open after the c_{wait} time period necessitating a longer opening time (i.e., a longer c_{react}).
- The c_{wait} time can be increased to increase the time between operations. For example, a motion-activated paper towel may stop or slow dispensing if it is continually dispensing.

While the adaptations to improve tolerance of specific faults differ by domain, the contextual metamorphic relation is applicable to multiple proximity-enabled responsive systems.

5.4 Step (4): MAPE-K Implementation

The implementation of the MAPE-K loop mirrors that of the implementation described in the approach (Section 3). Regardless of the contextual metamorphic relation, the MAPE-K loop implementation remains the same. In the next subsection we will discuss the general applicability of not just the MAPE-K loop implementation, but of contextual metamorphic relations themselves.

5.5 Contextual Metamorphic Relation Applicability

Identifying metamorphic relations manually is difficult and requires detailed knowledge and understanding of not only the system to be tested, but also the system domain. This challenge is exacerbated as systems become more complex resulting in greater variation in behavior due to both system and environmental factors. While adding an additional contextual expression adds complexity to a metamorphic relation, it also restricts the metamorphic relation to a smaller subset of the system's functionality. This additional term restricts contextual metamorphic relations to a subset of standard metamorphic relations but also makes identifying metamorphic relations for smaller ranges of system functionality more practical.

Despite the wide range of applicable of standard metamorphic relations [6] we have not found contextual metamorphic relations to be less applicable. Several methods of specifying software in individual components (e.g., feature models [21]) or identifying individual modes for testing (e.g., decision table or equivalence class testing [22]) inherently include contextual expressions. We believe that the reconfigurations of adaptive software, as discussed in [23], can be assessed via contextual metamorphic relations where the context is based on the source of the reconfiguration. The oracle problem in [23] is addressed by metamorphic relations specific to reconfiguration when the reconfiguration is deterministic within the context of the contextual metamorphic relation.

Given the logical and structural componentization of software systems, the wide ranging applicability of metamorphic relations are still likely to be applicable for contextual metamorphic relations. However, rather than additional complexity within the metamorphic relation directly to ensure a valid relation the applicability can be defined outside of the metamorphic relation itself.

6. Related Work

Self-adaptive systems have employed a variety of approaches to control autonomous activity including machine learning, hand-crafted policies, multi-agent systems approaches, and control-theoretic approaches as defined in a recent survey by Porter *et al.* [24]. We discuss hand-crafted policies, machine learning and statistical approaches, control-theoretic work, and test-case adaptation below.

Contextual metamorphic relations are a hand-crafted policy, though the relations differ in how they identify scenarios that required adaptation. Hand-crafted policies that use run-time monitors to detect failure [25] or monitor application health via frameworks [26] or utility functions [27]–[29] rely on pre-deployment differentiation of positive and negative behavior. Methods exist to predict behavior to enable resilient run-time monitors [30] manage uncertainty, but still rely on direct measurements of expected behavior. Metamorphic Runtime Checking has been introduced [31], but does not consider metamorphic relations that are not applicable to the state of the program. As far as the authors are aware, no other work uses metamorphic relations to trigger adaptation at run time to ensure tolerance to faults.

Machine learning approaches have been used to successfully learn adaptation [32], reduce the adaptation space [33], and have even been combined with control theory [34]. However, these methods and other machine learning techniques [35] require computational resources not necessary for *contextual* metamorphic relations. Similarly, rigorous statistical methods to assess dynamic behavior of software systems that change behavior (i.e., adapt) during execution have been developed [36], [37]. However, contextual metamorphic relations allow for adaptation due to detected failures at runtime rather than statistical guarantees on the breadth of previously run tests. Such statistical methods could be employed on systems that include adaptation from contextual metamorphic relations for further guarantees.

Typical control-theoretic solutions [15] cannot be used to control both the ascent and descent due to the lack of a measurable positioning error when the drone is too high, even when optimizing the control's parameters [34] or using machine learning [38]. Methods that adapt the control itself to manage behavior may still allow undesired behavior until the controller has been adapted [39]. Control-theoretic solutions that employ machine learning are more likely to mitigate uncertainty by adjusting or adapting the control. The *contextual* metamorphic relations in our approach, however, still trigger adaptation without the computational expense of optimization or machine learning at run-time.

Methods to address uncertainty in runtime verification have utilized test case adaptation [40] with success, while our approach avoids direct comparisons to expected results via metamorphic relations. To the best of the authors' knowledge, this is the only work that addresses the oracle problem by employing metamorphic relations to specific contexts for both pre-deployment verification and contextual runtime adaptation after deployment.

7. Conclusion

In this paper, we have presented *contextual* metamorphic relations, an approach to detect system failure and trigger adaptation to support fault tolerance without explicit expected results. We demonstrated *contextual* metamorphic relations by applying them to a robotic drone system with various forms of uncertainty as well as proximity-enabled responsive systems. We showed that *contextual* metamorphic relations are able to automatically detect scenarios necessitating adaptation at run-time and trigger specific contextual adaptations due to faults without a priori knowledge of expected results.

This demonstration is intended as a proof of concept for *contextual* metamorphic relations though we acknowledge several threats to validity. First, only a single motion of the reflective surface was used across all simulations. Second, both the robotic drone system and the simulation itself are simplified. Third, the time frame of the simulations is short. Despite these limitations, the robot was simulated according to physical laws, the robotic drone is complicated enough to have real-world aberrant behavior in the face of uncertainty, and the aberrant behavior was illustrated within the short

time frame of the simulations. Additionally, we discuss the general applicability to a wider range of systems based on our discussion of proximity-enabled responsive systems.

Future research directions include additional validation with more complex systems and more realistic simulations over greater periods of time. Additionally, we plan to address the limitation of manually defined metamorphic relations and adaptations to allow for automatically defined or derived relations and adaptations based on system models or derived from information gathered at run time. For both manually and automatically defined metamorphic relations and contextual metamorphic relations we intend to explore the applicability of existing test assessment strategies (e.g., code coverage [22] and mutation score [41]) to assess scenario coverage and completeness. Finally, we plan to extend our work to handle fuzzy or RELAXed [42], [43] properties to address additional dimensions of uncertainty and adaptation.

ACKNOWLEDGMENT

This work has been supported in part by grants from Grand Valley State University and the Michigan Space Grant Consortium (80NSSC20M0124). The views and conclusions contained herein are those of the authors and do not necessarily represent the opinions of the sponsors. The authors thank Bryce DeVries for his material support in the form of toy drone that was destroyed while inspiring this work.

REFERENCES

[1] R. Calinescu, R. Mirandola, D. Perez-Palacin, and D. Weyns, "Understanding uncertainty in self-adaptive systems," in *2020 ieee international conference on autonomic computing and self-organizing systems (acsos)*. IEEE, 2020, pp. 242–251.

[2] B. H. C. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*. Springer, 2009, pp. 1–26.

[3] 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 *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 468–483.

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

[5] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, and A. Finkelstein, "Requirements-aware systems: A research agenda for re for self-adaptive systems," in *2010 18th IEEE International Requirements Engineering Conference*. IEEE, 2010, pp. 95–103.

[6] S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen, "Metamorphic testing: Testing the untestable," *IEEE Software*, 2018.

[7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE transactions on software engineering*, vol. 41, no. 5, pp. 507–525, 2014.

[8] B. DeVries, E. M. Fredericks, and B. H. Cheng, "Analysis and monitoring of cyber-physical systems via environmental domain knowledge & modeling," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2021, pp. 11–17.

[9] A. J. Ramirez, A. C. Jensen, B. H. C. Cheng, and D. B. Knoester, "Automatically exploring how uncertainty impacts behavior of dynamically adaptive systems," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 568–571.

[10] M. A. Langford, G. A. Simon, P. K. McKinley, and B. H. C. Cheng, "Applying evolution and novelty search to enhance the resilience of autonomous systems," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 63–69.

[11] M. A. Langford and B. H. C. Cheng, "Enhancing learning-enabled software systems to address environmental uncertainty," in *2019 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2019, pp. 115–124.

[12] B. DeVries and C. Trefftz, "A novelty search and metamorphic testing approach to automatic test generation," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 8–11.

[13] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–27, 2018.

[14] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases. technical report hkust-cs98-01," *Hong Kong Univ. of Science and Technology*, 1998.

[15] K. H. Ang, G. Chong, and Y. Li, "Pid control system analysis, design, and technology," *IEEE transactions on control systems technology*, vol. 13, no. 4, pp. 559–576, 2005.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, no. 1, pp. 41–50, 2003.

[17] T. Y. Chen, P.-L. Poon, and X. Xie, "Metric: Metamorphic relation identification based on the category-choice framework," *Journal of Systems and Software*, vol. 116, pp. 177–190, 2016.

[18] H. Liu, X. Liu, and T. Y. Chen, "A new method for constructing metamorphic relations," in *2012 12th International Conference on Quality Software*. IEEE, 2012, pp. 59–68.

[19] S. Segura, A. Durán, J. Troya, and A. R. Cortés, "A template-based approach to describing metamorphic relations," in *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*. IEEE, 2017, pp. 3–9.

[20] P. Zave and M. Jackson, "Four dark corners of requirements engineering," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 1, pp. 1–30, 1997.

[21] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[22] P. C. Jorgensen and B. DeVries, *Software Testing: A Craftsman's Approach*, 5th ed. CRC Press, 2021.

[23] L. M. Prikler and F. Wotawa, "Challenges of testing self-adaptive systems," in *Proceedings of the 26th ACM International Systems and Software Product Line Conference-Volume B*, 2022, pp. 224–228.

[24] B. Porter, R. Rodrigues Filho, and P. Dean, "A survey of methodology in self-adaptive systems research," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 2020, pp. 168–177.

[25] B. DeVries and B. H. C. Cheng, "Run-time monitoring of self-adaptive systems to detect n-way feature interactions and their causes," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2018, pp. 94–100.

[26] J. Mertz and I. Nunes, "Tigris: A dsl and framework for monitoring software systems at runtime," *Journal of Systems and Software*, vol. 177, p. 110963, 2021.

[27] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das, "Utility functions in autonomic systems," in *Autonomic Computing, 2004. Proceedings. International Conference on*. IEEE, 2004, pp. 70–77.

[28] P. DeGrandis and G. Valetto, "Elicitation and utilization of application-level utility functions," in *Proceedings of the 6th international conference on Autonomic computing*. ACM, 2009, pp. 107–116.

[29] A. J. Ramirez and B. H. C. Cheng, "Automatic derivation of utility functions for monitoring software requirements," in *Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 501–516.

[30] M. A. Nia, M. Kargahi, and A. Abate, "Resilient monitoring in self-adaptive systems through behavioral parameter estimation," *Journal of Systems Architecture*, p. 102177, 2021.

[31] C. Murphy and G. E. Kaiser, "Metamorphic runtime checking of non-testable programs," 2009.

[32] P. Jamshidi, J. Cámara, B. Schmerl, C. Käestner, and D. Garlan, "Machine learning meets quantitative planning: Enabling self-adaptation in autonomous robots," in *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2019, pp. 39–50.

[33] J. Van Der Donckt, D. Weyns, F. Quin, J. Van Der Donckt, and S. Michiels, "Applying deep learning to reduce large adaptation spaces of self-adaptive systems with multiple types of goals," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020, pp. 20–30.

[34] R. D. Caldas, A. Rodrigues, E. B. Gil, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "A hybrid approach combining control theory and ai for engineering self-adaptive systems," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2020, pp. 9–19.

[35] O. Gheibi, D. Weyns, and F. Quin, "Applying machine learning in self-adaptive systems: A systematic literature review," *arXiv preprint arXiv:2103.04112*, 2021.

[36] C. Mandrioli and M. Maggio, "Testing self-adaptive software with probabilistic guarantees on performance metrics," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1002–1014.

[37] ——, "Testing self-adaptive software with probabilistic guarantees on performance metrics: extended and comparative results," *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3554–3572, 2021.

[38] D. Weyns, B. Schmerl, M. Kishida, A. Leva, M. Litoiu, N. Ozay, C. Paterson, and K. Tei, "Towards better adaptive systems by combining mape, control theory, and machine learning," *arXiv preprint arXiv:2103.10847*, 2021.

[39] M. H. Moghadam, M. Saadatmand, M. Borg, M. Bohlin, and B. Lisper, "Adaptive runtime response time control in plc-based real-time systems using reinforcement learning," in *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. IEEE, 2018, pp. 217–223.

[40] 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*. ACM, 2014, pp. 17–26.

[41] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

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

[43] ——, "Relax: a language to address uncertainty in self-adaptive systems requirement," *Requirements Engineering*, vol. 15, no. 2, pp. 177–196, 2010.