MITIGATING UNCERTAINTY AT DESIGN TIME AND RUN TIME TO ADDRESS ASSURANCE FOR DYNAMICALLY ADAPTIVE SYSTEMS

By

Erik M. Fredericks

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2015

ABSTRACT

MITIGATING UNCERTAINTY AT DESIGN TIME AND RUN TIME TO ADDRESS ASSURANCE FOR DYNAMICALLY ADAPTIVE SYSTEMS

By

Erik M. Fredericks

A dynamically adaptive system (DAS) is a software system that monitors itself and its environment at run time to identify conditions that require self-reconfiguration to ensure that the DAS continually satisfies its requirements. Self-reconfiguration enables a DAS to change its configuration while executing to mitigate unexpected changes. While it is infeasible for an engineer to enumerate all possible conditions that a DAS may experience, the DAS must still deliver acceptable behavior in all situations. This dissertation introduces a suite of techniques that addresses assurance for a DAS in the face of both system and environmental uncertainty at different levels of abstraction. We first present a technique for automatically incorporating flexibility into system requirements for different configurations of environmental conditions. Second, we describe a technique for exploring the code-level impact of uncertainty on a DAS. Third, we discuss a run-time testing feedback loop to continually assess DAS behavior. Lastly, we present two techniques for introducing adaptation into run-time testing activities. We demonstrate these techniques with applications from two different domains: an intelligent robotic vacuuming system that must clean a room safely and efficiently and a remote data mirroring network that must efficiently and effectively disseminate data throughout the network. We also provide an end-to-end example demonstrating the effectiveness of each assurance technique as applied to the remote data mirroring application.

To Natalie and Zoe, thank you for everything.
I couldn't have done this without you.

ACKNOWLEDGEMENTS

I first want to thank Dr. Betty H. C. Cheng for taking me under her wing and guiding me from the bright-eyed graduate student that I was to the sleepy-eyed, yet still optimistic, person I am today. Your willingness to provide feedback and many, many revisions on papers and projects is greatly appreciated. I want to also thank you for taking me along on the ICSE ride. Because of that invaluable experience, I met many wonderful people and helped to make something great, and I cannot thank you enough for that. Special thanks also go to Dr. McKinley for trading me to the highest bidder early on in my academic career. I am fairly certain that I would not have ended up in software engineering otherwise.

I also would like to thank my committee members: Dr. Philip McKinley, Dr. Erik Goodman, Dr. William Punch, and Dr. Xiaobo Tan. Their valuable feedback and willingness to attend overly long presentations throughout this entire process has been greatly appreciated.

Additionally, there are many other people that I would like to thank who have helped me along the way. First of all, I'm sure that, at some point, all members of the SENS lab were pulled into some sort of discussion on software engineering (either willingly or unwillingly). Many thanks to Jared Moore, Tony Clark, Chad Byers, Daniel Couvertier, Andres Ramirez, and Byron DeVries. Outside of academia, I would like to thank Matthew Jermov, Robb Melenyk, and Greg Robertson. Their oft-uncredited discussions and support definitely helped me get where I am today.

I would also like to thank my family and friends for both believing in me and suffering with me through this long and somewhat arduous process. Any social life that I previously had disappeared the moment I entered graduate school, and I appreciate all of the support and understanding that you all have given to me. Lastly, I especially would like to thank my parents. Thank you for giving me the drive and desire to accomplish whatever I set my mind to.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Cyber-physical systems are being increasingly implemented as safety-critical systems [74]. Moreover, these software systems can be exposed to conditions for which they were not explicitly designed, including unexpected environmental conditions or unforeseen system configurations. For domains that depend on these safety-critical systems, such as patient health monitoring or power grid management systems, unexpected failures can have costly results. In response to these issues, dynamically adaptive systems (DAS) have been developed to constantly monitor themselves and their environments, and if necessary, change their structure and behavior to handle unexpected situations. Providing run-time assurance that a DAS is consistently satisfying its high-level objectives and requirements remains a challenge, as unexpected environmental configurations can place a DAS into a state in which it was not designed to execute. For this dissertation, environmental uncertainty refers to unanticipated combinations of environmental conditions. In contrast, system uncertainty encompasses imprecise or occluded sensor measurements, sensor failures, unintended system interactions, and unexpected modes of operation. As such, we developed new techniques to address assurance in the face of both types of uncertainty at different levels of abstraction, including requirements, code, and testing.

While software engineering techniques have historically focused on design-time approaches for developing a DAS [47, 59, 114] and providing design-time assurance [48, 50, 78, 85], increasingly, more efforts are targeting assurance and system optimization at run time [15, 36, 39, 78]. As cyber-physical systems continue to proliferate, techniques are needed to ensure that their continued execution satisfies their key objectives. For example, agent-based approaches have been developed to provide continual run-time testing [78, 79] in order to verify that the system is behaving as expected. However, these approaches tend to run in a parallel, sandboxed environment. Conversely, we focus on addressing assurance concerns within the production environment.

This dissertation presents a suite of techniques at different levels of abstraction that collectively addresses assurance concerns for a DAS that is experiencing both system and environmental uncertainty. Specifically, our techniques provide automated assurance that a DAS continually satisfies its requirements and key objectives within its requirements, implementation, and testing levels of abstraction. Where applicable, we leverage search-based software engineering heuristics to augment our techniques.

1.1 Problem Description

The field of software engineering strives to design systems that continuously satisfy requirements even as environmental conditions change throughout execution [21, 94, 113]. To this end, a DAS provides an approach to software design that can effectively respond to unexpected conditions that may arise at run time by performing a self-reconfiguration. Design and requirements elicitation for a DAS can begin with a high-level, graphical description of system objectives, constraints, and assumptions in the form of a goal model that can be used as a basis for the derivation of requirements [21]. Adaptation strategies for performing system reconfiguration can then be defined to mitigate previously identified system and environmental conditions in order to effectively respond to known or expected

situations [22, 47, 81]. The DAS then monitors itself and its environment at run time to determine if the identified conditions are negatively impacting the DAS, and if so, performs a self-reconfiguration to mitigate those conditions.

In addition to design-time assurance, verification and validation activities are required to provide a measure of assurance that the DAS succeeds in satisfying key objectives and requirements. However, standard testing techniques must be augmented when testing a DAS as it may exhibit unintended behaviors or follow unexpected paths of execution in the face of uncertainty [45]. Moreover, due to the complex and adaptive nature of a DAS, run-time techniques are necessary to continually validate that the DAS satisfies its requirements even as the environment changes [19, 41, 43, 44, 48]. These concerns imply that assurance techniques for a DAS must not only be applied at design time, but also at run time.

As a result, we have identified the following challenges that currently face the DAS research community:

- Provide effective assurance that the DAS will satisfy its objectives in different operating conditions at run time.
- Anticipate and mitigate unexpected sources of uncertainty in both the system and environment.
- Leverage and extend traditional testing techniques to handle the needs of a complex adaptive system.

1.2 Thesis Statement

This research is intended to explore methods for addressing uncertainty with a focus on providing assurance at different levels of abstraction in a DAS. In particular, techniques for facilitating requirements adaptation, exploration of DAS behaviors expressed as a result of exposure to different combinations of system and environmental uncertainty, and performing run-time testing are examined to provide assurance for the DAS.

1.2.0.0.1 Thesis Statement.

Evolutionary techniques can be leveraged to address assurance concerns for adaptive software systems at different levels of abstraction, including at the system's requirements, implementation, and run-time execution levels, respectively.

1.3 Research Contributions

The overarching objective of this dissertation is to provide assurance for a DAS at different levels of abstraction. We address this goal with three key research objectives. First, we provide a set of techniques that mitigate uncertainty by providing assurance for a DAS's requirements, implementation, and run-time behavior. Second, automation is a key driving force to designing, performing, and analyzing the particular issues presented by each assurance technique. Third, we provide a feedback loop to enable testing of a DAS at run time. To accomplish these key objectives, we developed the following set of techniques:

1. At the requirements level, we provide an automated technique for mitigating uncertainty at the DAS requirements level by exploring how goals and requirements can be made more flexible by automatically introducing RELAX operators [21, 113], each of which map to a fuzzy-logic membership function. We also introduce a method for optimizing the exploration process of goal RELAXation for different environmental contexts. To this end, we introduce AutoRELAX [42, 88] and AutoRELAX-SAW [42], respectively. AutoRELAX provides requirements-based assurance by exploring how RELAX operators can introduce flexibility to a system goal model. AutoRELAX-SAW, in turn, extends AutoRELAX by balancing the competing concerns that manifest when the DAS experiences different combinations of environmental conditions.

2. At the implementation level, we develop an automated technique to explore and optimize run-time execution behavior of a DAS as it experiences diverse combinations of

operating conditions. In particular, we introduce Fenrir [45], a technique that leverages novelty search [64] to determine the specific path of execution that a DAS follows under different sets of operating conditions, thereby potentially uncovering unanticipated or anomalous behaviors.

3. At the run-time execution level, we define and implement a feedback loop that provides run-time assurance for a DAS through online monitoring and adaptive testing. Specifically, we introduce the MAPE-T feedback loop [44], a technique that defines how run-time testing is enabled via *Monitoring*, *Analyzing*, *Planning*, and *Executing*, linked together by *Testing knowledge*, thereby taking a proactive approach in ensuring that a DAS continually exhibits correct behaviors. Furthermore, we introduce techniques to facilitate run-time adaptation of test suites [41] and test cases [43] to ensure that assurance is continually provided even as environmental conditions change over time.

Figure 1.1 presents a high-level description of how each technique facilitates assurance at the requirements, implementation, and testing levels. First, AutoRELAX (1) and AutoRELAX-SAW (2) assess and mitigate both system and environmental uncertainty at the requirements level. Next, Fenrir (3) explores how program behavior, as defined by DAS code, is affected by system and environmental uncertainty. Techniques (1), (2), and (3) are intended to be performed at design time. Next, the MAPE-T feedback loop (4) provides run-time assurance for both the DAS's requirements and implementation by performing run-time testing in the face of uncertainty. The MAPE-T loop is supported by Proteus (5) and Veritas (6), where Proteus and Veritas each address assurance by ensuring that test suites and test cases, respectively, remain relevant as environmental conditions change and the DAS transitions to new system configurations. As the operating context evolves, Proteus and Veritas ensure that test suites and test cases correctly evolve in turn. Techniques (4), (5), and (6) are intended to be performed at run time.

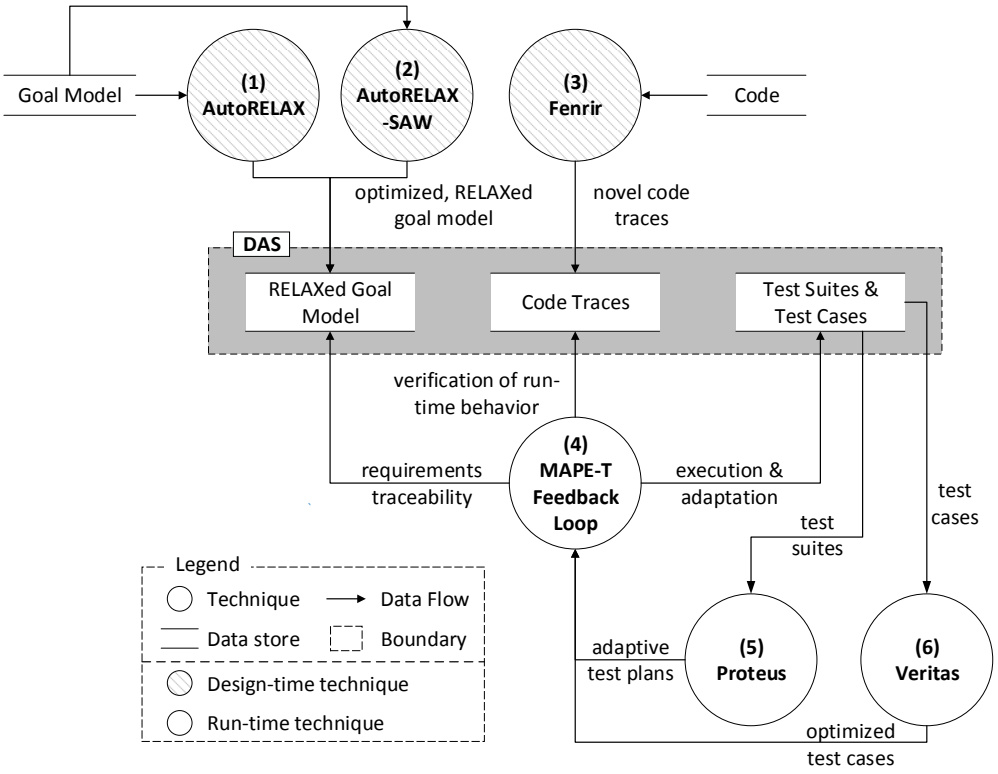


Figure 1.1: High-level depiction of how our suite of techniques impacts a DAS at varying levels of abstraction.

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 provides background information on enabling technologies used for our research, including DASs, goal-oriented requirements modeling, evolutionary computation, the RELAX requirements specification language, and software testing. Chapter 2 also presents background information on our case studies used to validate our techniques: intelligent vacuum systems and remote data mirroring. Next, Chapter 3 presents AutoRELAX and AutoRELAX-SAW, techniques for mitigating the effects of environmental uncertainty at the system requirements level. Then, Chapter 4 describes Fenrir, our technique for exploring the impact of environmental uncertainty on DAS behavior at the code level. Next, Chapter 5 introduces the MAPE-T feedback loop and describes the key elements of the feedback loop. Chapter 6 then describes a real-

ization of MAPE-T with two techniques, Proteus and Veritas, that perform online adaptation of test suites and test cases, respectively, to ensure testing activities remain relevant to the operating context even as the DAS reconfigures and the environment changes over time. Following, Chapter 7 discusses the impact that a run-time testing framework can have on a DAS and also presents techniques for optimizing run-time testing. Chapter 8 then presents an end-to-end example where each of our techniques were applied to the RDM case study in a stepwise process. Finally, Chapter 9 presents our conclusions and summarizes the research contributions presented in this dissertation and then discusses future directions for this line of work.

Chapter 2

Background and Application

This chapter provides relevant background information on the topics discussed within this dissertation: dynamically adaptive systems (DAS), smart vacuum systems (SVS), remote data mirroring (RDM) networks, requirements engineering, and evolutionary computation (EC). First, we overview the components of a DAS and its self-reconfiguration capabilities. Next, we describe the SVS and RDM applications, including the implementation details of each as they are used throughout this dissertation as motivating examples. Following, we describe requirements engineering from the perspective of system goal models and also overview the RELAX specification language. We then present EC and how it can be used to explore complex search spaces. Finally, we describe software testing and highlight common approaches for performing testing.

2.1 Dynamically Adaptive Systems

The state-space explosion of possible combinations of environmental conditions that a system may experience during execution precludes their total enumeration [21, 113]. Moreover, system requirements may change after initial release, thus potentially requiring a new software release or patch. A DAS provides an approach to continuously satisfy requirements by changing its configuration and behavior at run time to mitigate changes in its require-

ments or operating environment [68, 81]. As such, we consider a DAS to comprise a set of non-adaptive configurations connected by adaptive logic [114]. Figure 2.1 illustrates our approach for implementing a DAS. Specifically, the example DAS comprises n configurations ($C_{1..n}$), each of which is connected by adaptive logic (A). Each configuration C_i satisfies requirements for a given operating context and each path of adaptation logic defines the steps and conditions necessary to move from a *source* DAS configuration to a *target* DAS configuration (e.g., from C_1 to C_2).

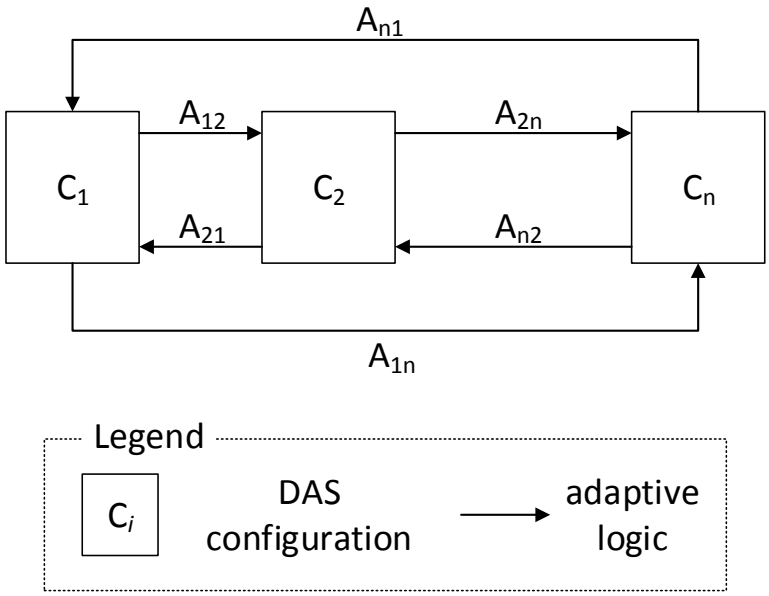


Figure 2.1: Structure of dynamically adaptive system.

Some DASs are embedded systems and achieve dynamic adaptations via mode changes [77]. Mode changes can enable run-time adaptation in situations in which it is either not safe or practical to upload or offload software on executing systems, and thus, mode changes are required to capture the effects of dynamic reconfiguration. In particular, mode changes enable a DAS to self-reconfigure by selecting discrete modes of operation, where a mode is characterized by a particular configuration of system resources and parameters. For example, an autonomous wheeled robot may be characterized by different pathfinding modes and can transition from an *exploration* mode to a *wall-following* mode based on input from monitored sensors and a central controller.

2.2 Smart Vacuum System

This section describes the smart vacuum system (SVS) application that is used as a case study throughout this dissertation. First, we overview SVSs in general, present our derived goal model of the SVS, and then discuss our implementation of the SVS.

2.2.1 Overview of Smart Vacuum System

SVSs are currently available in the consumer market, with a notable example being iRobot's Roomba.[1] An SVS must clean a desired space by using sensor inputs to balance path planning, power conservation, and safety concerns. Common sensors available to an SVS include bumper sensors, motor sensors, object sensors, and internal sensors. Bumper sensors provide feedback when the robot collides with an object, such as a wall or a table. Motor sensors provide information regarding wheel velocities, suction speed, and power modes. Object sensors, for example infrared or camera sensors, can be used to detect and identify different types of entities near the SVS. Internal sensors provide feedback regarding sensor health or the overall state of the SVS. A robot controller processes data from each sensor to determine an optimal path plan, conserve battery power as necessary, avoid collisions with objects that may damage the SVS or the object itself (e.g., a pet or a child), and avoid objects which, if vacuumed, could damage the internal components of the SVS (e.g., liquid or large dirt particle).

Due to its relative level of sophistication, an SVS can also be modeled as an adaptive system [7, 8]. Specifically, the SVS can perform mode changes at run time [77] as a means to emulate the self-reconfiguration capabilities of a DAS. Each mode provides the SVS with the capability to select an optimal configuration of system parameters to properly mitigate uncertainties within the system and environment. An example of system uncertainty is noisy or untrustworthy sensor data, and an example of environmental uncertainty is the

[1] See `http://www.irobot.com/`

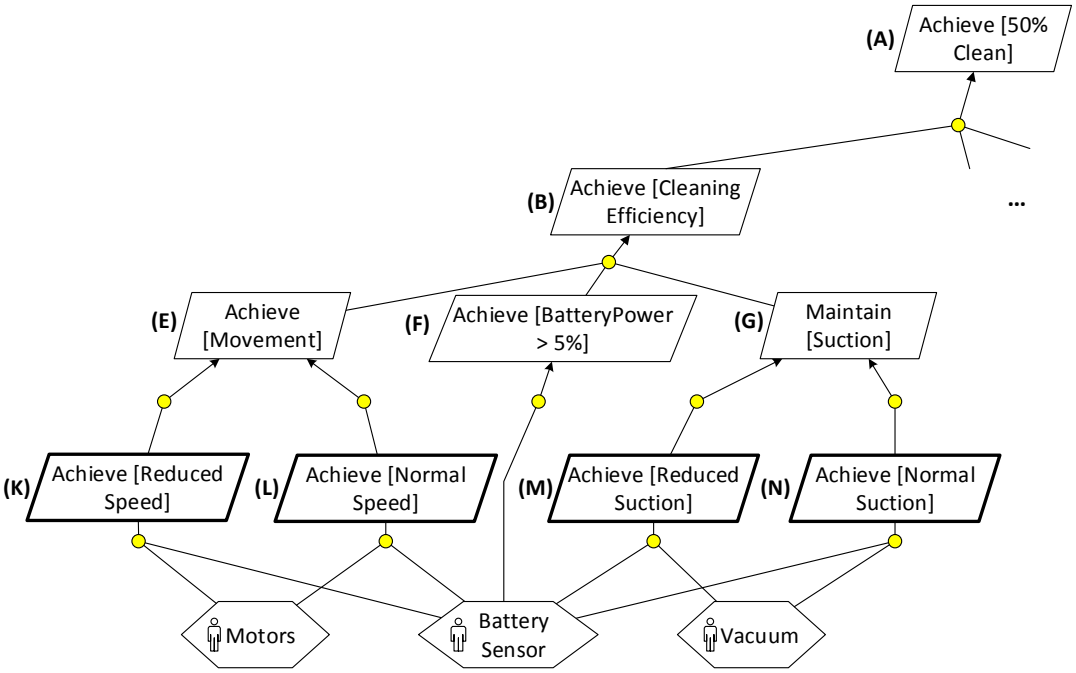
possibility for a liquid to have been spilled in the room in which the SVS is operating. Possible SVS modes include different pathfinding algorithms, reduced power consumption modes, and obstacle avoidance measures. Each mode of operation can be configured in a different manner, leading to an explosion of possible configuration states.

Figure 2.2 presents a KAOS goal model for the SVS application. The SVS must successfully clean at least 50% of the small dirt particles within the room (A). To do so, the SVS must operate efficiently (B) to conserve battery power (F) while still providing both movement (E) and suction (G) capabilities. The SVS can operate in a normal power mode for speed (L) and suction (N), or lower its power consumption (F) by operating in a reduced power mode for speed (K) and/or suction (M). The SVS must also clean the room effectively (C) by selecting an appropriate path plan. The SVS can both clean and explore the room by selecting either a random (O) or straight (P) path for 10 seconds (H), or focus on a smaller area by selecting the 20 second (I) spiral (Q) path plan. Moreover, the SVS must also satisfy safety objectives (D). If a safety violation occurs, then the SVS must activate a failsafe mode (J). Safety violations include collisions with specific obstacles (e.g., pets or children), falling down stairs (R), or collisions with objects that can damage the SVS (S).

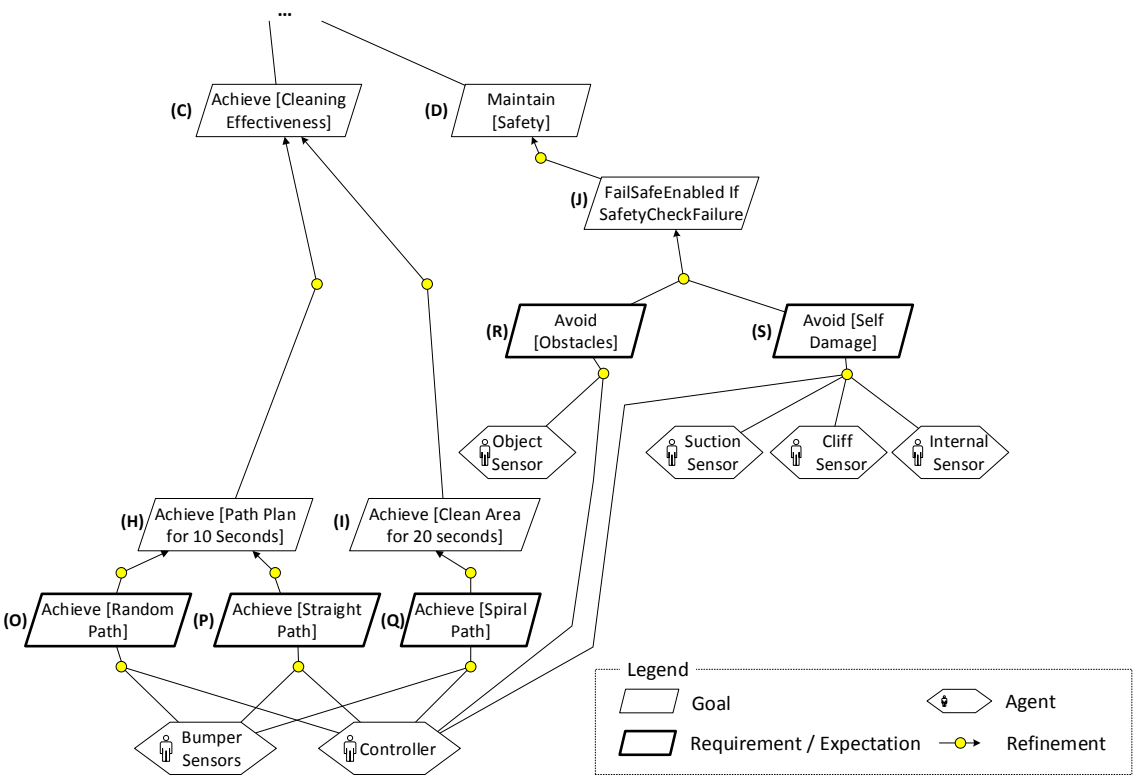
2.2.2 Smart Vacuum System Implementation

This section describes the implementation of the SVS that is used throughout this dissertation. The SVS was configured as an autonomous robot that must efficiently, effectively, and safely vacuum dirt particles within a room while avoiding obstacles and objects that may cause harm to the SVS. Figure 2.3 provides a screenshot of the SVS simulation as implemented within the Open Dynamics Engine physics platform.[2] The screenshot shows the SVS within a simulated room, containing small dirt particles (i.e. small dark cubes), large dirt particles (i.e., large dark cubes), a liquid spill (i.e., small yellow disc), and two obstacles (i.e., thin red pillars) that the SVS must avoid.

[2]See `http://www.ode.org`.



(A) Left half of smart vacuum system goal model.



(B) Right half of smart vacuum system goal model.

Figure 2.2: KAOS goal model of the smart vacuum system application.

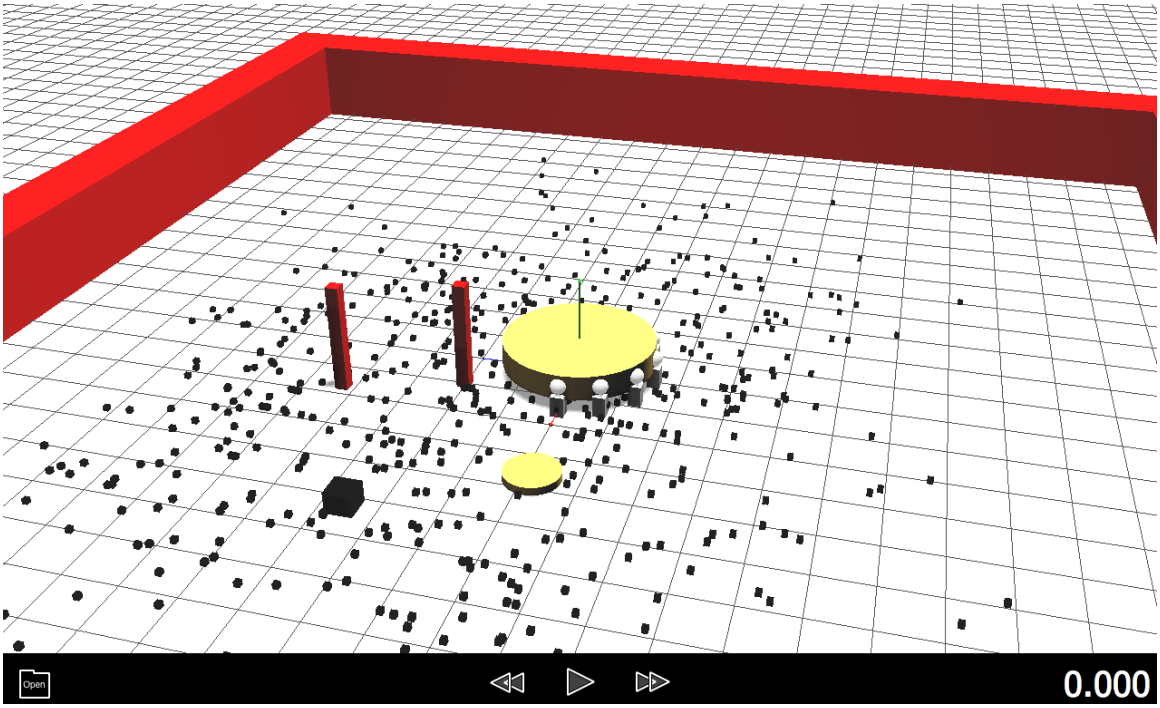


Figure 2.3: Screenshot of SVS simulation environment.

The SVS (i.e., large yellow disc) comprises a controller and a set of available sensors, including an array of bumper sensors (i.e., small gray spheres attached to SVS), an array of cliff sensors (i.e., gray rectangles attached to SVS), an object sensor, an internal sensor, and wheel and suction sensors. The bumper sensor array provides feedback upon contact with a wall or object. The cliff sensor array detects downward steps and is intended to prevent damage from falling. The object sensor detects the distance between the SVS and nearby objects and also provides information regarding the type of object that was detected. The internal sensor monitors the SVS to determine if any damage has occurred, such as vacuuming liquid or large dirt particles. Wheel and suction sensors provide feedback regarding the velocity and suction power of the wheel and suction motors, respectively, and also provide information regarding the health of each. Lastly, a controller aggregates the information from all sensors to determine an appropriate path plan for the robot to follow, as well as a power consumption plan to determine if the SVS must take measures to conserve battery power.

To illustrate our implementation, we now describe an example of the SVS performing mode changes. At the start of the simulation, the SVS selects a *SPIRAL* path plan that is to be executed for 20 seconds. Within 10 seconds, the cliff sensor detects that the SVS is near a downward step and therefore changes its mode to a cliff avoidance plan, pausing the *SPIRAL* path plan while the SVS begins to move in reverse, away from the step. The *cliff avoidance* mode runs for 5 seconds to ensure that the SVS has avoided the step, and then resumes the *SPIRAL* path plan mode for the remaining 5 seconds. At 20 seconds, the SVS selects a new path plan mode (e.g., *STRAIGHT*, 10 seconds) and begins to move in a straight line, as opposed to the spiraling path it followed previously. After executing for several minutes, the SVS's internal sensor detects that the amount of available battery power is falling below a predefined threshold (e.g., 50% remaining). The SVS then selects a *power conservation* mode and reduces power to its wheels, slowing the overall velocity of the SVS while effectively extending its battery life. This process of mode changes reflects how an onboard system-based DAS adapts and selects new configurations at run time.

2.3 Remote Data Mirroring

This section describes the remote data mirroring (RDM) application that is used as a case study throughout this dissertation. First, we overview RDMs in general, present our derived goal model of the RDM, and then discuss our implementation of the RDM.

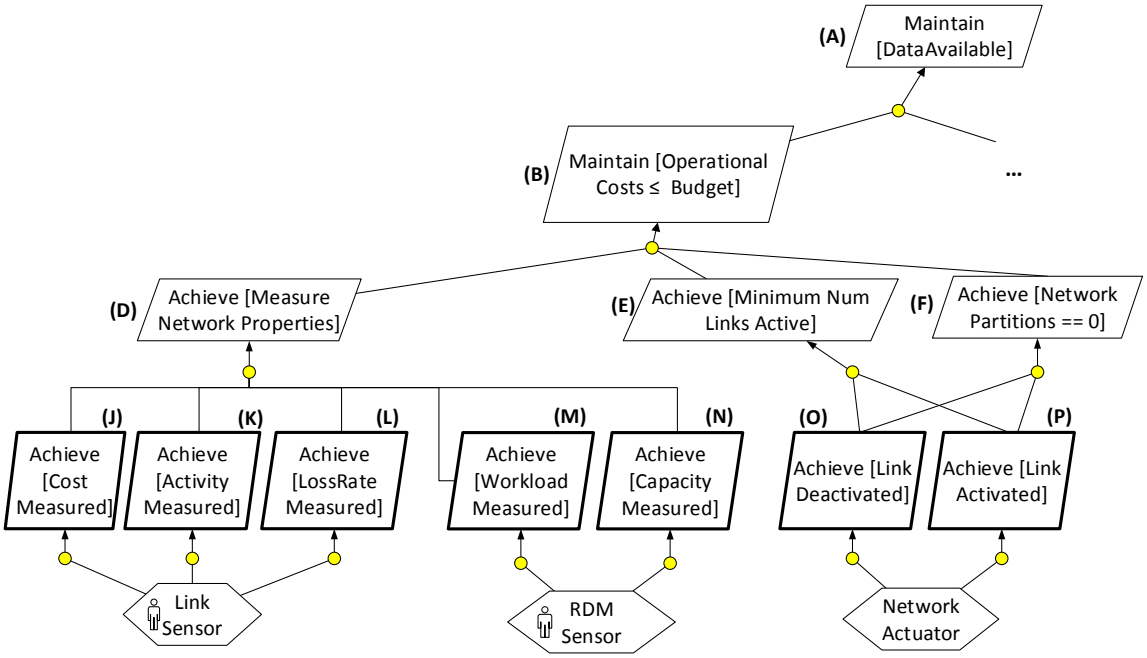
2.3.1 Overview of Remote Data Mirroring Application

RDM is a data protection technique for maintaining data availability and preventing data loss by storing copies (i.e., replicates) on servers (i.e., data mirrors) in physically remote locations [56, 60]. By replicating data on remote data mirrors, an RDM can provide continuous access to data and moreover ensure that data is not lost or damaged. In the event of an error or failure, data recovery can be facilitated by either requesting or reconstructing the

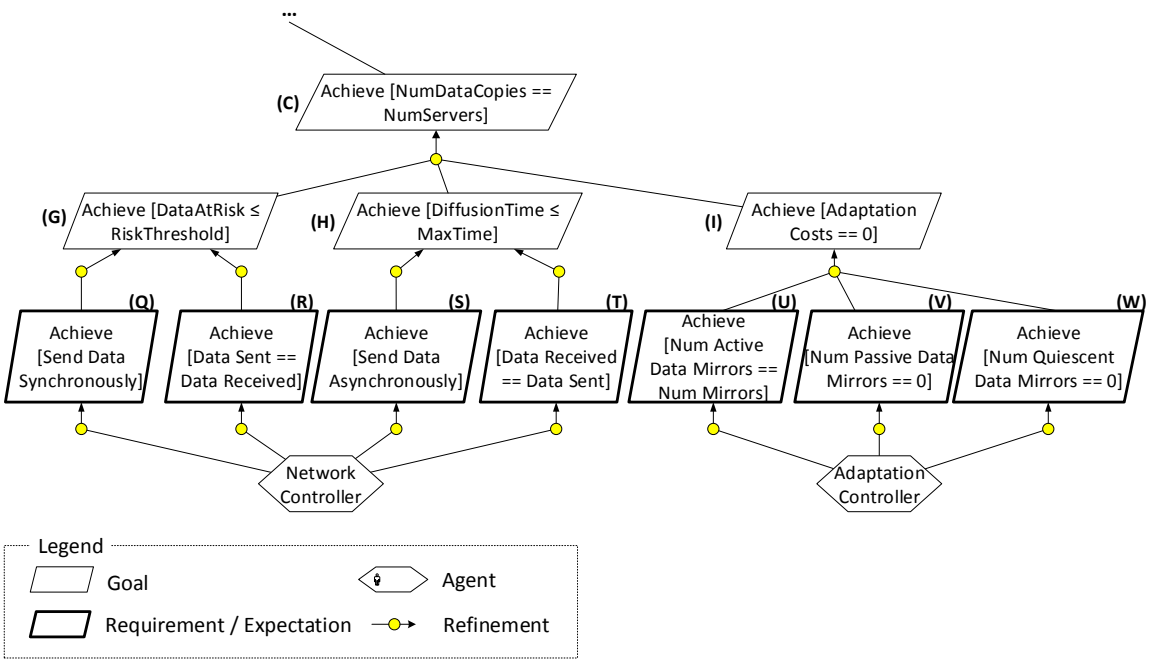
lost or damaged data from another active data mirror. Additionally, the RDM network must replicate and distribute data in an efficient manner by minimizing consumed bandwidth and providing assurance that distributed data is not lost or corrupted.

The RDM can reconfigure at run time in response to uncertainty, including dropped or delayed messages and network link failures. Furthermore, each network link incurs an operational cost that directly impacts a controlling budget and also has a measurable throughput, latency, and loss rate. Collectively, these metrics determine the overall performance and reliability of the RDM. To mitigate unforeseen issues, the RDM can reconfigure in terms of its network topology and data mirroring protocols. Specifically, the RDM can selectively activate and deactivate network links to change its overall topology. Furthermore, each data mirror can select a remote data mirroring protocol, defined as either *synchronous* or *asynchronous* propagation. *Synchronous* propagation ensures that the receiving or secondary data mirror both receives and writes incoming data before completion at the primary or sending site. Batched *asynchronous* propagation collects updates at the primary site that are periodically transmitted to the secondary site. Given its complex and adaptive nature, the RDM application can be modeled and implemented as a DAS [89].

Figure 2.4 provides a KAOS goal model of the RDM application. Specifically, the RDM must maintain remotely stored copies of data (A). To satisfy this goal, the RDM must maintain operational costs within a fixed budget (B) while ensuring that the number of disseminated data copies matches the number of available servers (C). To satisfy Goal (B), the RDM must be able to measure all network properties (D) while ensuring that both the minimum number of network links are active (E) and that the network is unpartitioned (F). To satisfy Goal (C), the RDM must ensure that risk (G) and time for data diffusion (H) each remain within pre-defined constraints, and moreover, the cost of network adaptation must be minimized (I). To satisfy Goals (D) – (I), RDM agents, such as sensors and actuators, must be able to measure and effect all available network properties, respectively.



(A) Left half of remote data mirroring goal model.



(B) Right half of remote data mirroring goal model.

Figure 2.4: KAOS goal model of the remote data mirroring application.

2.3.2 Remote Data Mirroring Implementation

This section describes the implementation of the RDM application used for case studies within this dissertation. Specifically, we modeled the RDM network as a completely connected graph, where each node represents an RDM and each edge represents a network link. In total, the RDM network comprises 25 RDMs with 300 network links. Each link can be activated or deactivated, and while active, can be used to transfer data between RDMs. An operational model previously introduced by Keeton *et al.* [60] was used to determine performance attributes for each RDM and network link. The RDM application was simulated for 150 time steps. During each simulation, 20 data items were inserted into randomly selected RDMs at different times in the simulation. The selected RDMs were then responsible for distributing the data items to all other RDMs within the network.

The RDM network is subject to uncertainty throughout execution. For example, unpredictable network link failures and dropped or delayed messages can affect the RDM at any point during the simulation. In response, the network can self-reconfigure to move to a state that can properly mitigate these setbacks. To this end, each RDM implements the dynamic change management (DCM) protocol [63], as well as a rule-based adaptation engine, to monitor goal satisfaction and determine if a reconfiguration of topology or propagation method is required. Upon determining that self-reconfiguration is necessary, a target network configuration and set of reconfiguration steps are generated to ensure a safe transition to the new configuration.

2.4 Requirements Engineering

This section presents background information on goal-oriented requirements engineering, goal-oriented requirements modeling, and the RELAX specification language.

2.4.1 Goal-Oriented Requirements Engineering

An integral part of software engineering is in eliciting, analyzing, and documenting objectives, constraints, and assumptions required for a system-to-be to solve a specific problem [105]. In the 4-variable model proposed by Jackson and Zave [55], the problem to be solved by the system-to-be exists within some organizational, technical, or physical context. As a result, the system-to-be shares a boundary with the area surrounding the problem, interacting with that world and its stakeholders. As a result, the system-to-be must monitor and control parts of this shared boundary to solve the problem.

Goal-oriented requirements engineering (GORE) extends the 4-variable model with the concept of a goal. Specifically, a goal guides the elicitation and analysis of system requirements based upon key objectives of the system-to-be. Furthermore, the goal must capture stakeholder intentions, assumptions, and expectations [105]. Several types of goals exist: a *functional* goal declares a service that a system-to-be must provide to its stakeholders; a *non-functional* goal imposes a quality constraint upon delivery of those services; a *safety* goal is concerned with critical safety properties of a system-to-be; and a *failsafe* goal ensures that the system-to-be has a fallback state in case of critical error. Additionally, a functional, safety, or failsafe goal may be declared invariant (i.e., must always be satisfied; denoted by keyword "Maintain" or "Avoid") or non-invariant (i.e., can be temporarily unsatisfied; denoted by keyword "Achieve"). Goals may also be satisficed, or satisfied to a certain degree, throughout execution [24].

2.4.2 Goal-Oriented Requirements Modeling

The GORE process gradually decomposes high-level goals into finer-grained subgoals [105], where the semantics of goal decomposition are captured graphically by a directed acyclic graph. Each node within the graph represents a goal and each edge represents a goal refinement. Figure 2.5 presents a KAOS goal model [25, 105] based on the SVS that is

used as a case study throughout this dissertation. KAOS depicts goals and refinements as parallelograms with directed arrows that point towards the higher-level (i.e., parent) goals. KAOS also supports AND/OR refinements, where an AND-decomposition is satisfied only if all its subgoals are also satisfied, and an OR-decomposition is satisfied if at least one of its subgoals is satisfied. Generally, AND-refinements capture objectives that must be performed in order to satisfy the parent goal, and OR-refinements provide alternative paths for satisfying a particular goal.

Goal decomposition continues until each goal has been assigned to an agent capable of achieving that goal. An agent represents an active system component that restricts its behavior to fulfill leaf-level goals (i.e., requirements/expectation goals) [105]. There are two types of agents: system and environmental. A system agent is an automated component controlled by the system-to-be, and an environmental agent is often a human or some component that cannot be controlled by the system-to-be.

For example, Figure 2.5 presents an example of a partial KAOS goal model that defines a subset of the high-level goals for the SVS. In particular, Goal (B) is decomposed into Goals (E), (F), and (G) via an AND-decomposition. Goal (B) can only be satisfied if and only if Goals (E), (F), and (G) are also satisfied. Furthermore, Goal (G) is decomposed via an OR-decomposition into Goals (M) and (N), implying that Goal (G) is satisfied if at least one subgoal is also satisfied. Leaf-level goals (K), (L), (M), and (N) represent low-level requirements or expectations. Lastly, the hexagonal objects (e.g., Motors, Battery Sensor, Vacuum) represent agents that must achieve the leaf-level goals.

2.4.3 RELAX Specification Language

RELAX [21, 113] is a requirements specification language used to identify and assess sources of uncertainty. RELAX *declaratively* specifies the sources and impacts of uncertainty at the shared boundary between the execution environment and system-to-be [55]. Furthermore, a requirements engineer organizes this information into three distinct elements: ENV,

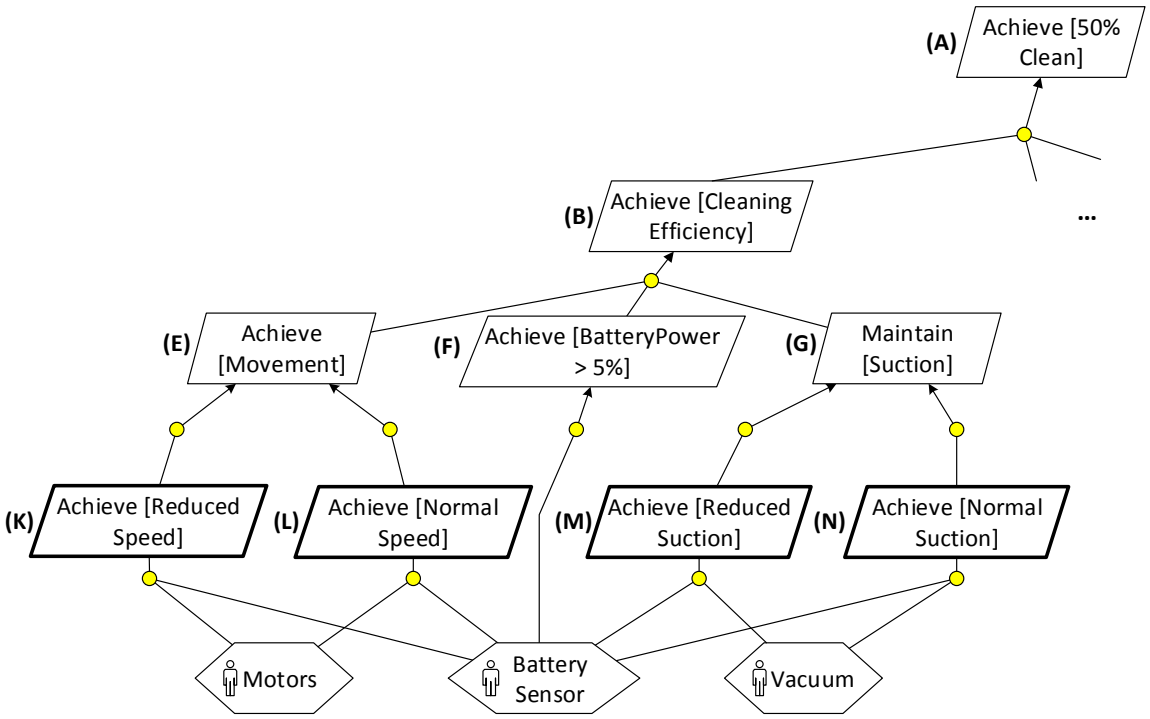


Figure 2.5: Partial KAOS goal model of smart vacuum system application.

MON, and REL. ENV defines environmental properties that can be observed by the DAS's monitoring infrastructure. MON specifies the elements that are available within the monitoring infrastructure. REL defines the method to compute a quantifiable value of ENV properties from their corresponding MON elements.

The semantics of RELAX operators are defined in terms of fuzzy logic and specify the extent that a non-invariant goal can be temporarily unsatisfied at run time [113]. Table 2.1 describes the intent of each RELAX operator. For instance, in Figure 2.5, the RELAX operator `AS CLOSE AS POSSIBLE TO` 0.05 can be applied to Goal (F) to allow flexibility in the remaining amount of battery power for the system.

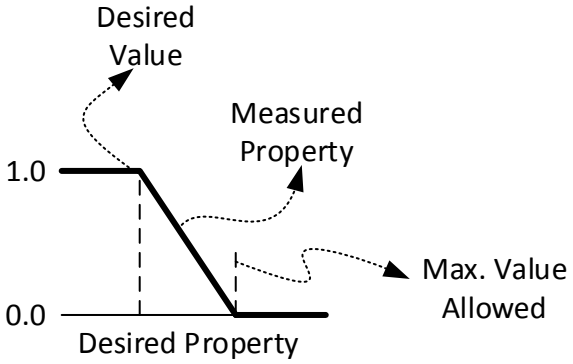
Figure 2.6 gives the fuzzy logic membership functions that have been implemented for RELAX and used within this dissertation. Specifically, the `AS EARLY AS POSSIBLE` and `AS FEW AS POSSIBLE` use the left shoulder function from Figure 2.6(A). The `AS LATE AS POSSIBLE` and `AS MANY AS POSSIBLE` operators use the right shoulder function from Fig-

Table 2.1: RELAX operators [113].

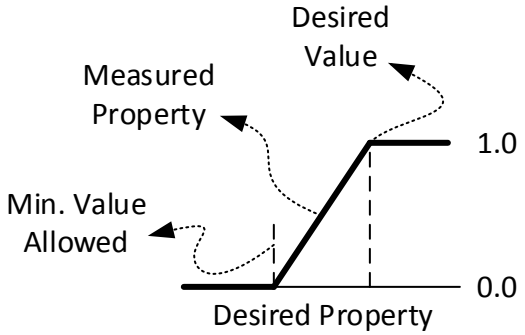
RELAX **Operator**	**Informal Description**	**Fuzzy-logic Membership Function**
`AS EARLY AS POSSIBLE` ϕ	ϕ becomes true as close to the current time as possible.	Left Shoulder
`AS LATE AS POSSIBLE` ϕ	ϕ becomes true as close to time $t = \infty$ as possible.	Right shoulder
`AS CLOSE AS POSSIBLE TO` [$frequency$ ϕ]	ϕ is true at periodic intervals as close to $frequency$ as possible.	Triangle
`AS FEW AS POSSIBLE` ϕ	The value of a quantifiable property ϕ is as close as possible to 0.	Left shoulder
`AS MANY AS POSSIBLE` ϕ	The value of a quantifiable property ϕ is maximized.	Right shoulder
`AS CLOSE AS POSSIBLE TO` [$quantity$ ϕ]	The value of a quantifiable property ϕ approximates a desired target value $quantity$.	Triangle

ure 2.6(B). Lastly, the `AS CLOSE AS POSSIBLE TO` operators use the triangle function from Figure 2.6(C).

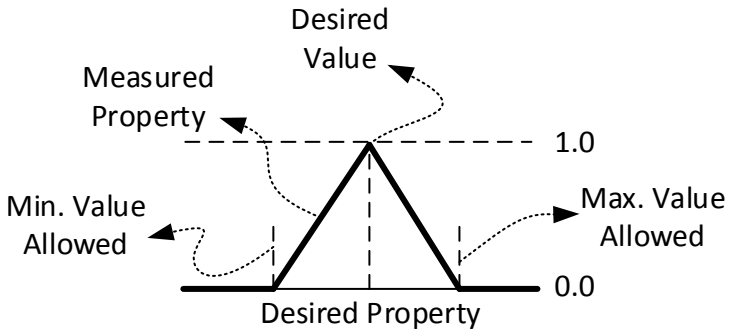
Cheng *et al.* [21] previously proposed a manual approach for applying RELAX operators to KAOS non-invariant goals. A requirements engineer must first specify the ENV, MON, and REL elements necessary to define the sources of uncertainty within the operational context. Next, each goal must be designated as either invariant or non-invariant, where invariant goals are precluded from RELAXation. For each non-invariant goal, the engineer must determine



(A) Left shoulder function



(B) Right shoulder function



(C) Triangle shoulder function

Figure 2.6: Fuzzy logic membership functions.

if any of the defined sources of uncertainty can cause the goal to become unsatisfied. The engineer then applies an appropriate RELAX operator to restrict how the particular goal may be temporarily violated. For a modest-sized goal model with minimal sources of uncer-

tainty, many possible combinations of RELAXed goals are possible, thereby necessitating an automated approach for applying RELAX operators [42, 88].

2.5 Evolutionary Computation

Evolutionary computation (EC) comprises a family of stochastic, search-based techniques that are considered to be a sub-field of artificial and computational intelligence [46, 57]. Genetic algorithms [53], genetic programming [62], evolutionary strategies [96], digital evolution [80], and novelty search [64] are different approaches to EC. These techniques are generally used to solve problems in which there is a large solution space, such as complicated optimization or search problems within software engineering [20, 42, 49, 50, 51, 52], robotics [62], and land use management [23]. EC techniques typically implement evolution by natural selection as a means to guiding the search process towards optimal areas within the solution space.

To implement an EC approach, the parameters that comprise a candidate solution must be fully defined. These parameters must be encoded into a data structure that enables operations specified by the corresponding algorithm [46] to facilitate evolutionary search. As EC is grounded in Darwinian evolution, the use of biologically-inspired terms is relevant. In particular, a *gene* represents an element or parameter directly manipulable by the evolutionary algorithm. A set of genes is known as a *genome* and represents a candidate solution or individual. Lastly, the set of genomes represents a *population*. Evolutionary operations, such as *crossover*, *mutation*, and *selection* can be applied to a genome over a number of *generations*, or iterations, to simulate evolution.

2.5.1 Genetic Algorithms

A genetic algorithm [53] is a stochastic, search-based heuristic grounded in EC that can be used to explore the space of possible solutions for complex optimization problems. A

genetic algorithm typically represents each possible solution (i.e., individual) in an encoded fashion that is amenable to manipulation and evaluation. For example, Table 2.2 illustrates a typical genetic algorithm encoding, wherein a vector of numbers comprises the parameters necessary to represent a particular solution. The data within this table represents k candidate solutions with an encoding of length n. Each parameter value is represented as a floating point number, and directly corresponds to a feature within the candidate solution. Candidate solutions can be encoded in many other formats as well, including bit strings and variable-length vectors.

Table 2.2: Sample genetic algorithm encoding.

Solution	**Parameter A**	**Parameter B**	**...**	**Parameter** n
$Individual_1$	0.2	1.5	...	p_{n1}
$Individual_2$	0.4	1.2	...	p_{n2}
...	...	...	...	...
$Individual_k$	p_{Ak}	p_{Bk}	...	p_{nk}

A set of candidate solutions must undergo an evolutionary process to guide the search towards an optimal solution. To do so, a set of evolutionary operations is executed upon the population until a termination criterion, typically a specific number of generations, is performed. These operations include population generation, performing crossover and mutation, and fitness evaluation with respect to predefined fitness criteria. Each of these evolutionary operations is next described in turn.

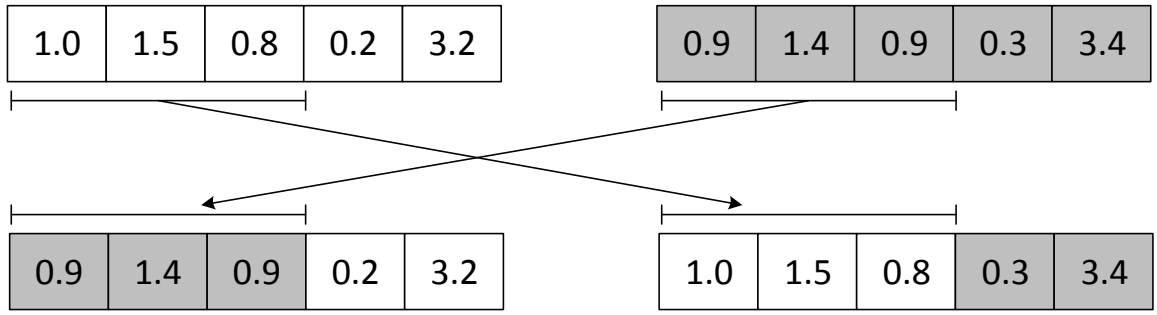
2.5.1.1 Population Generation

At the beginning of the genetic algorithm, a population of individuals with completely randomized parameter values is generated. Throughout each successive generation, new individuals are created via the crossover and mutation operations. At the end of each

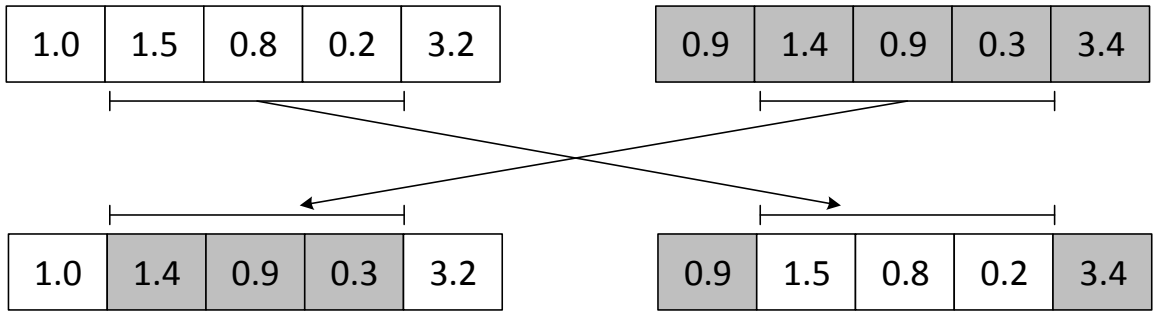
generation, the highest performing, or elite, individuals may be retained to protect the "successful" individuals that represent a particular area of the solution space.

2.5.1.2 Crossover

The crossover operation creates new individuals by combining genetic information from two existing individuals. Crossover is commonly applied via one-point or two-point crossover. As demonstrated by Figure 2.7(A), one-point crossover selects a random gene within the genome and then exchanges the genes before and after that point to create two new children. Two-point crossover, shown in Figure 2.7(B), selects two random genes and exchanges the genes between those two points with another individual, thereby creating two new children as well.



(A) One-point crossover example.



(B) Two-point crossover example.

Figure 2.7: Examples of one-point and two-point crossover.

2.5.1.3 Mutation

The mutation operation creates a new individual by selecting a random gene from a random individual and mutating the selected gene. Mutation provides an approach for exploring different areas of the solution space that may not have been explored by the genetic algorithm. Figure 2.8 provides an example of mutation, where a gene from the parent individual is randomly mutated to create a new child individual.

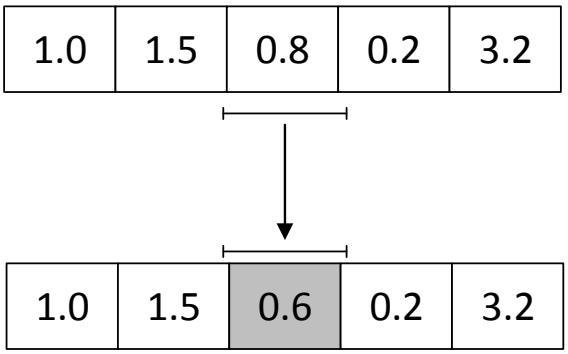


Figure 2.8: Example of mutation.

2.5.1.4 Fitness Evaluation

Each candidate solution's fitness must be calculated to determine which areas of the solution space should be explored by retaining the highest-performing individuals. Specifically, predefined fitness criteria are applied to each candidate solution within the population, thereby quantifying each individual's performance. If multiple fitness functions are necessary due to domain constraints, then each fitness sub-function can be combined to form a single aggregate fitness function. One approach for combining fitness sub-functions uses a linear weighted sum, where each weight provides a metric for the relative impact of each sub-function.

2.5.2 Stepwise Adaptation of Weights

Fitness sub-functions calculate a quantifiable performance metric for specific concerns, the aggregation of which forms an overall fitness value. Adding a weighting coefficient to each sub-function in turn determines the relative importance, or impact, of that particular sub-function on an aggregate fitness value (i.e., a higher weight value gives preference to a particular fitness sub-function in the overall fitness) [31]. However, the definition of weight values often depends upon domain knowledge of the DAS engineer or is based on observed or calculated metrics. As a result, an automated approach to calculating fitness sub-function weights is necessary.

The stepwise adaptation of weights (SAW) [31, 104] is a hyper-heuristic [13] for optimizing the weighting coefficients for fitness sub-function equations. Specifically, the weights are adjusted over time and the aggregate fitness value is monitored to determine if a particular combination of weights improves overall fitness. This approach can be implemented either *offline* or *online*. An offline SAW adjusts sub-function weights following execution of an evolutionary algorithm (EA), whereas an online SAW implementation adjusts weights dynamically throughout execution of the EA.

For the purposes of this dissertation, we have used the online SAW approach [104] as it provides a fast and efficient method for optimizing fitness sub-function weights. Within online SAW, weights are seeded with values identified as optimal by a requirements engineer. Throughout execution, feedback from the controlling EA is analyzed periodically to determine which weight requires adjustment. The fitness sub-function that performs the worst (i.e., yields the lowest fitness value) has its associated weight increased to provide preference to that particular fitness sub-function. The remaining weights are then normalized to ensure that all weights sum to a value of 1.0.

2.5.3 Novelty Search

Novelty search [64] is a branch of genetic algorithms that explicitly searches for *unique* solutions, as opposed to optimal solutions. The intention of this approach is to avoid converging to a locally optimal solution and instead explore the entire solution space to find a set of solutions that may contain a globally optimal solution. Novelty search often replaces or augments the traditional fitness function with a novelty function that implements a distance metric, such as a Euclidean distance [10], to determine the distance between a candidate solution and its nearest neighbors in the solution space. A novelty archive tracks candidate solutions considered to be the most novel throughout the evolutionary process and is used to guide the search towards unexplored areas within the solution space.

2.5.4 (1+1)-ONLINE Evolutionary Algorithm

Generally, EC is a computationally-intense technique that must be performed offline. However, complex optimization problems with an enormous solution space can exist at run-time, thereby necessitating the need for an online technique to solve these problems. The (1+1)-ONLINE EA [12] is a lightweight approach to providing run-time evolution on systems with limited processing capabilities. (1+1)-ONLINE EA sacrifices searching power for performance by providing a population with only two individuals: a parent and a child. In this approach, the parent individual is evaluated for a set amount of time. At the end of its allotted time, the child individual asserts control of the system and is then evaluated for the same amount of time. Upon completion of its execution cycle, the performance of both the parent and child are compared, with the higher-performing individual retained for the following generation. A new child is then created by mutating the retained individual, and the process is then repeated until either a satisfactory individual is found or a specified number of generations has occurred.

The (1+1)-ONLINE EA can also search locally or globally within the solution space by specifying a mutation parameter σ. Upon evaluation of each individual, the (1+1)-ONLINE EA determines if the fitness has stagnated (i.e., no significant increase in value), where stagnation implies that the search procedure is trapped within a local optima. The mutation value σ determines the severity of the mutation, where higher severities search other regions of the solution space, and lower severities explore nearby regions of the search space. By varying σ as necessary, the search process has a better chance of avoiding convergence to a non-optimal solution.

2.6 Software Testing

Software testing is an assurance technique that determines if a software system is operating according to its requirements or design specification [72]. Many techniques exist for performing software testing at design time [9] as well as approaches for extending common testing techniques to the search-based and evolutionary domain [52]. A subset of the most common techniques [14, 82] are structural testing, functional testing, unit testing, integration testing, and regression testing. Each of these techniques is next described in turn.

2.6.1 Structural Testing

Structural testing, typically known as white-box testing, often is concerned with validating coverage metrics such as branch or data flow coverage. Test engineers generally have access to the source code or other abstractions of system implementation to target implementation-specific aspects for verification and validation. For example, a test engineer can write a set of test cases that triggers difference branches of an `if-else` construct to ensure that each branch is accessible within the program.

2.6.2 Functional Testing

Functional testing, otherwise known as black-box testing, generally validates a system based solely on a test specification. In this approach, a test engineer does not have implicit knowledge of the inner workings of a system. At a high-level, functional testing is concerned with verifying that defined features function correctly within a software implementation. For example, a functional test case could be defined to validate that an autonomous robot successfully actuates a motor to a particular velocity.

2.6.3 Unit Testing

Unit testing is concerned with testing individual source code modules, and often are considered the "smallest" test that can be performed when verifying a system. Moreover, unit tests are generally performed by the software developer to ensure that each code module performs according to key objectives or requirements, thereby catching bugs early in the software development cycle. For example, a unit test verifies that the output of a particular function matches a pre-defined expected value.

2.6.4 Regression Testing

Regression testing is an assurance technique for verifying a system following a change in software or operating context. Specifically, regression testing validates that the system continues to satisfy previously validated requirements. For example, a regression test ensures that a new software release does not introduce any new issues or side effects against a previously executed test case.

Chapter 3

Addressing Requirements-Based Uncertainty

This chapter introduces our techniques for automatically providing requirements-based assurance for a DAS. Specifically, we explore how RELAX operators can be automatically applied to a goal model using AutoRELAX and how the fitness sub-function weights that guide the creation of automatically RELAXed goal models can be optimized with AutoRELAX-SAW [42, 88]. First, we motivate the need for requirements-based assurance. Next, present AutoRELAX and AutoRELAX-SAW. We then present two empirical studies across different application domains for each technique. Lastly, we describe related work, summarize our findings, and propose future directions for this research.

3.1 Motivation

Uncertainty experienced by the DAS at run time can adversely affect its ability to satisfy requirements. While uncertainty can be mitigated by performing a self-reconfiguration, introducing adaptations can incur operational costs in both preparing for and executing the adaptation. To this end, RELAX [21, 113] has been introduced to provide flexibility in requirements satisfaction as a means to reduce the number of adaptive reconfigurations.

However, the number of possible combinations for defining and applying RELAX operators to a goal model exponentially increases with the number of non-invariant goals and sources of uncertainty, thereby necessitating the need for an automated approach. For example, the RDM goal model (c.f., Figure 2.4) comprises 21 non-invariant goals, each of which may have one of the 6 presented RELAX operators applied (c.f., Table 2.1). Moreover, each RELAX operator can be designated as active or inactive when applied to the goal model. As such, there exist 12^{21} possible goal models comprising combinations of applied RELAX operators that must be examined. To enable exploration of this enormous search space, we introduce AutoRELAX and AutoRELAX-SAW.

AutoRELAX and AutoRELAX-SAW support our overarching objective to providing assurance in exploring how RELAX operators can provide flexibility at the requirements level. Specifically, AutoRELAX automatically generates RELAXed goal models while minimizing the number of DAS adaptations and the number of RELAXations to enable the DAS to temporarily tolerate adverse uncertainty. AutoRELAX-SAW automatically balances the competing concerns of adaptation minimization and minimization of applied RELAX operators by tailoring fitness sub-function weights to the operating context being experienced by the DAS.

3.2 Introduction to AutoRELAX

In this section, we present AutoRELAX, an evolutionary approach for automatically applying RELAX operators to a goal model to mitigate system and environmental uncertainty at the requirements level. First, we introduce the assumptions that are needed to implement and apply AutoRELAX to a given problem. Next, we describe the inputs and the expected outputs of AutoRELAX. Following, we describe each step in the AutoRELAX approach in detail. Lastly, we describe how SAW, a hyper-heuristic for optimizing the weights for a set of fitness

sub-functions, can be applied to AutoRELAX to tailor fitness sub-functions to a particular environment, thereby yielding a higher overall fitness value.

Thesis statement: *It is possible to automatically introduce flexibility into system requirements to mitigate system and environmental uncertainty.*

3.2.1 Assumptions, Inputs, and Outputs

In this section we present the required assumptions for implementing AutoRELAX, as well as the necessary inputs and expected outputs.

3.2.1.1 Assumptions

Three assumptions must hold true for AutoRELAX to successfully generate an optimized set of RELAXed goal models to mitigate different combinations of system and environmental configurations. In particular, we assume that:

- System and environmental data monitored by the DAS is accessible to AutoRELAX's genetic algorithm.
- The requirements engineer has identified a representative set of system and environmental uncertainties.
- The requirements engineer has constructed a KAOS goal model that is representative of the DAS requirements specification.

3.2.1.2 Inputs and Outputs

AutoRELAX requires three elements as input. First, a goal model of the DAS must be provided to capture high-level requirements. Next, a set of utility functions must be derived to provide a quantifiable measure of requirements monitoring [28, 87, 109]. Lastly, an executable specification or prototype of the DAS is required to simulate different types of

system and environmental conditions. A set of RELAXed goal models is provided as output. Each input and output is next described in detail.

3.2.1.2.1 Goal Model

AutoRELAX requires as input a KAOS goal model [25, 105] of the DAS to capture the functional requirements of a system and resolve obstacles to goal satisfaction. Moreover, goals within a KAOS goal model can be specified as either invariant (i.e., cannot be violated) or non-invariant (i.e., can be temporarily unsatisfied without adverse consequences). AutoRELAX can then introduce RELAX operators to non-invariant goals, as invariant goals are precluded from adaptation as they must always be satisfied.

3.2.1.2.2 Utility Functions

Utility functions provide an approach for quantifying and measuring run-time satisfaction of software requirements [28, 87, 109]. Specifically, utility functions can yield a quantifiable metric that a DAS can use to determine if its goals are being satisfied. Each utility function comprises a mathematical relationship that maps data monitored by the DAS to a scalar value between 0.0 and 1.0. Utility functions can be derived either manually [109], automatically by using a goal model [87], or statistically inferred based upon observed DAS behaviors [28]. For example, the satisfaction of Goal (B) in the RDM goal model presented in Figure 2.4 is evaluated with a utility function that returns 1.0 if accrued operational costs remain less than or equal to a pre-determined budget, and 0.0 otherwise. AutoRELAX leverages utility functions to evaluate how applied RELAXations either positively or negatively impact DAS behavior at run time.

For this research, we manually derived the utility functions based upon the RDM goal model (see Figure 2.4). Specifically, the utility functions were derived according to the Athena approach previously developed by Ramirez and Cheng [87]. Athena uses a goal model and a mapping of environmental conditions to elements that monitor those conditions to

derive either state, metric, or fuzzy logic-based utility functions. State-based utility functions monitor goals that are Boolean in nature. Both metric and fuzzy logic-based utility functions monitor the *satisficement* of goals. A RELAX operator is then used to specify an appropriate membership function for fuzzy logic goals to derive a utility function. In the context of the RDM (c.f., Figure 2.4), a state-based utility function may monitor determine if invariant goal (A) is either satisfied or not satisfied by monitoring if data is always accessible by the network; a metric-based utility function may measure the satisficement of non-invariant goal (F) to determine how many network partitions have been introduced; and a fuzzy logic-based utility function can determine the satisficement of a goal to which a RELAX operator has been applied, for instance, Goal (C) can be RELAXed with the `AS CLOSE AS POSSIBLE TO` operator to provide flexibility in the number of data copies with respect to the number of servers.

3.2.1.2.3 Executable Specification

An executable specification or prototype for the DAS is used by AutoRELAX to evaluate the effects of different combinations of system and environmental conditions on DAS behavior. First, a requirements engineer identifies sources of uncertainty that can affect a DAS at run time and the executable specification is then initialized with those uncertainties. For instance, the RDM can be initialized according to the probability that certain network links will fail at any given point during the simulation. Varying the sources of uncertainty, including their likelihood of occurrence and impact, will ideally trigger the DAS to self-reconfigure, thereby leading to different types of RELAXed goal models. The executable specification then uses the input utility functions to evaluate the satisfaction of requirements at run time.

3.2.1.2.4 RELAXed Goal Models

Given this information, AutoRELAX generates a set of RELAXed goal models that capture optimal configurations of applied RELAXations. Each goal model in the output set represents

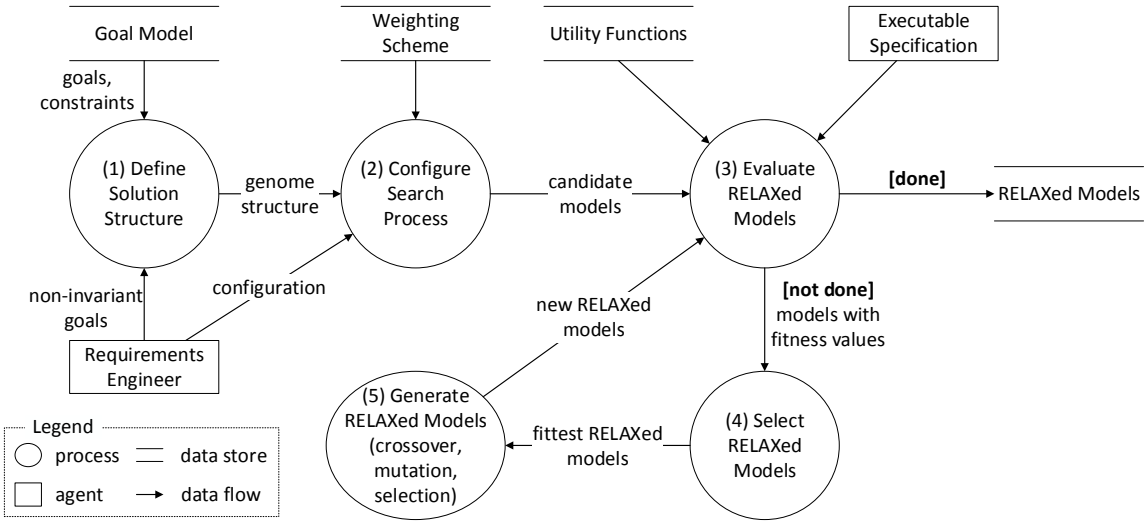


Figure 3.1: Data flow diagram of AutoRELAX process.

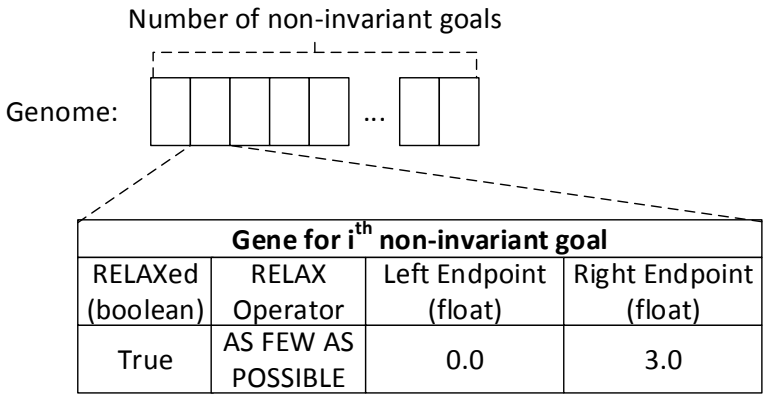
a combination of goal RELAXations that maximize run-time goal satisfaction for a particular set of system and environmental conditions.

3.2.2 AutoRELAX Approach

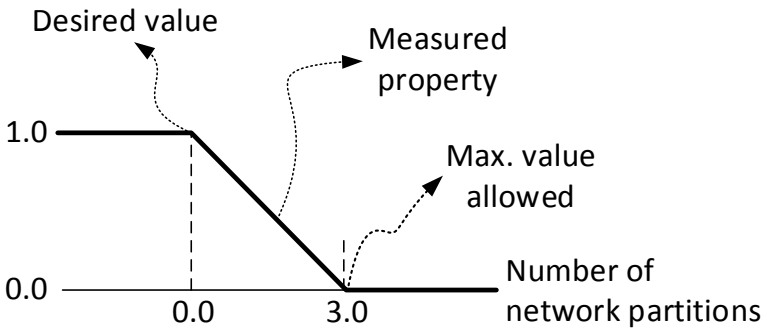
This section describes the AutoRELAX approach. Figure 3.1 presents a data flow diagram (DFD) that overviews AutoRELAX. We now present each step in detail.

(1) Define Solution Structure: First, an encoding must be specified for candidate solutions. Each solution comprises a vector of n elements or *genes*, where n is equal to the total number of non-invariant goals specified by the input goal model. Figure 3.2(A) presents the gene structure. Specifically, each gene comprises a Boolean variable that specifies if each non-invariant goal is to be RELAXed, a corresponding RELAX operator (see Table 2.1), and two floating-point values that specify the left and right boundaries of the fuzzy logic membership function, respectively.

Figure 3.2(B) illustrates how each gene is then mapped to its corresponding fuzzy logic function to evaluate the satisfaction of that particular goal. This example maps Goal (F) from Figure 2.4 to a *left shoulder* fuzzy logic membership function, corresponding to the `AS FEW AS POSSIBLE` RELAX operator. This function returns 1.0 if the network is connected



(A) AutoRELAX solution encoding.



(B) Mapping a gene to a RELAXed goal.

Figure 3.2: Encoding a candidate solution in AutoRELAX.

(i.e., the number of network partitions equals zero) and 0.0 otherwise. However, as long as the network partition is *transient* (i.e., minor and temporary), then data can still be diffused among connected data mirrors. While the RDM contains between 0 and 3 transient network partitions, the downward slope from the apex to the right endpoint reflects values that may be temporarily tolerated at run time, resulting in a quantifiable satisfaction value between 1.0 and 0.0.

(2) Configure Search Process: The genetic algorithm must then be configured by a requirements engineer. Specifically, a population size, crossover and mutation rates, and a termination criterion must be defined. The population size determines how many candidate RELAXed goal models are available each generation; crossover and mutation rates specify

how AutoRELAX generates new goal models; and the termination criterion defines a stopping point for AutoRELAX, and is typically represented as a generational limit.

(3) Evaluate RELAXed Models: Each RELAXed goal model is evaluated based on the performance of the DAS throughout execution, where performance is measured based on the satisfaction of invariant goals and the fitness sub-functions defined in Equations (3.1) and (3.2). Minimizing the number of DAS adaptations and number of RELAXed goals is also emphasized within the fitness calculation. To evaluate a RELAXed goal model, AutoRELAX first maps the RELAX operators encoded in an individual goal model to their corresponding utility functions based on Step (1). AutoRELAX then executes the prototype and records the satisfaction of each goal and the number of adaptations performed by the DAS throughout the simulation. Two fitness sub-functions consume this information to emphasize individuals that minimize the number of introduced RELAXations and minimize the number of run-time DAS reconfigurations.

The first fitness sub-function, FF_{nrg}, rewards candidate individuals that minimize the number of RELAXed goals. The purpose of this fitness sub-function is to limit an excess of unnecessary flexibility provided by the controlling goal model. FF_{nrg} is defined as follows:

$$FF_{nrg} = 1.0 - \left(\frac{|relaxed|}{|Goals_{non-invariant}|}\right) \tag{3.1}$$

where $|relaxed|$ and $|Goals_{non-invariant}|$ are the number of RELAXed and non-invariant goals in the goal model, respectively.

The second fitness sub-function, FF_{na}, rewards candidate individuals that minimize the number of reconfigurations performed by the DAS throughout execution in order to reduce overhead incurred on performance and cost. FF_{na} is defined as follows:

$$FF_{na} = 1.0 - \left(\frac{|adaptations|}{|faults|}\right), \tag{3.2}$$

where $|adaptations|$ measures the total amount of reconfigurations performed by the DAS, and $|faults|$ measures the total number of adverse environmental conditions introduced throughout the simulation, thereby reducing the number of passive and quiescent components [63].[1] A *passive* component can service transactions from other components, however it cannot initiate new transactions. *Quiescent* components cannot initiate new transactions or service incoming transactions. For the RDM, the operational cost of maintaining the network varies due to the number of nodes placed into a passive or quiescent state to ensure system consistency. Reducing the amount of the passive and quiescent components minimizes the impact of adaptation on operational costs. RELAX operators can increase a system's flexibility and ability to tolerate uncertainty, resulting in fewer nodes placed into a passive or quiescent state.

The fitness sub-functions are then combined into a single fitness calculation with a linear weighted sum in Equation (3.3):

$$Fitness\ Value = \begin{cases} \alpha_{nrg} * FF_{nrg} + \alpha_{na} * FF_{na} & \text{iff invariants true} \\ 0.0 & \text{otherwise} \end{cases} \tag{3.3}$$

where α_{nrg} and α_{na} are weights that reflect the priority of each fitness sub-function with respect to the calculated fitness value. The sum of the two weights must equal 1.0. Fitness of a RELAXed goal model also depends on the satisfaction of all goals designated as invariant. Specifically, any violation of an invariant goal results in a fitness value of 0.0, as AutoRELAX must only produce *viable* RELAXed goal models.

(4) Select RELAXed Models: Next, AutoRELAX selects the highest-performing individuals from the population based on their calculated fitness value to guide the search process towards that particular area of the solution space. AutoRELAX then applies tourna-

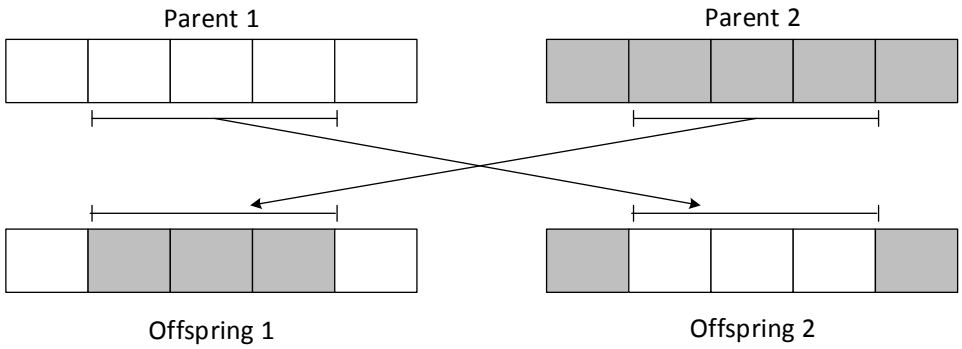
[1] A component refers to an executable element, such as a software component or network node.

ment selection [53], a technique that randomly selects k individuals from the population and competes them against one another. For each group of competing individuals, AutoRELAX selects the RELAXed goal model with the highest fitness to survive into the next generation. The remaining individuals do not survive and are effectively removed from consideration.

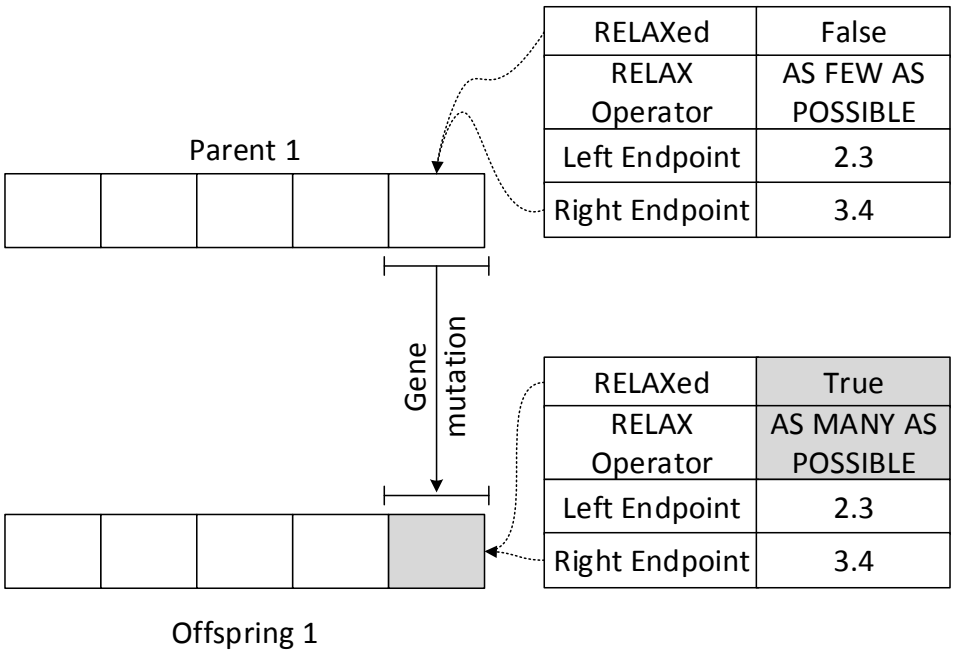
(5) Generate RELAXed Models (crossover, mutation, selection): AutoRELAX uses two-point crossover and single-point mutation to create new RELAXed goal models. The rates for crossover and mutation were specified to be 50% and 40%, respectively.[2] Figure 3.3(A) demonstrates two-point crossover. Specifically, two individuals are selected at random from the population and designated as *parents*. Two new individuals, designated as *children*, are produced by exchanging genes that lie between two randomly selected points along the genome. Conversely, Figure 3.3(B) demonstrates single-point mutation. In this example, a single individual is selected at random, and a single point along its genome is randomly selected and then mutated. Specifically, the mutation operator in this example changes the RELAX operator from `AS FEW AS POSSIBLE` to `AS MANY AS POSSIBLE`, and moreover sets the gene to enable RELAXation. Mutation is precluded from affecting the fuzzy logic function boundaries, as that would alter the meaning of a fully satisficed goal. In summary, the crossover operation attempts to generate new individuals that combine good elements from their parents, and mutation introduces a random element to facilitate exploration of the solution space.

Output RELAXed Models: AutoRELAX iteratively applies Steps (3) through (5) until the termination criterion (i.e., generational limit) is reached and then outputs the set of RELAXed goal models with the highest fitness values.

[2]The crossover and mutation rates were determined based upon the rate of convergence found from empirical results.



(A) Two-point crossover.



(B) Single-point mutation.

Figure 3.3: Examples of crossover and mutation operators in AutoRELAX.

3.2.3 Optimizing Fitness Sub-Function Weights with SAW

While AutoRELAX can automatically provide requirements-based assurance for a DAS, the generated combination of RELAX operators as applied to the goal model may not be optimal across different operating contexts. SAW is a hyper-heuristic for optimizing a given weighting scheme by iteratively exploring how different combinations of weights can improve overall fitness within a given environment [104]. Fitness sub-functions often use weighted coefficients to define the relative importance of each sub-function and can therefore be optimized with SAW. Specifically, SAW explores the space of weight combinations within a particular operating context to determine which weighting scheme is appropriate (i.e., yields a higher fitness value) for a specific environmental configuration.

Weights are typically assigned to fitness sub-functions based mainly on the domain knowledge of a requirements engineer, coupled with trial and error and limited empirical evidence. As a result, the combination of weighting coefficients may not be optimal for the specific operating context in which the DAS is executing and therefore an algorithmic approach is necessary to determine the optimal weighting scheme. SAW iteratively updates each weight over the course of an EA to explore how different combinations of weights affect the fitness of an individual, resulting in a combination of weights optimized for a specific environment.

AutoRELAX-SAW extends AutoRELAX with the SAW approach to optimize weights in the fitness function. Specifically, we have added the *online* method of SAW optimization [104] to AutoRELAX. This online approach can be performed in parallel to DAS execution. Specifically, weights are initially seeded with values specified by a requirements engineer. Throughout DAS execution, feedback is provided to AutoRELAX-SAW to determine which fitness sub-functions are providing the smallest fitness value compared to the other sub-functions. The poorer performing sub-function has its associated weight increased to indicate its importance

to that specific sub-function, and the remaining weights are then normalized to ensure that all weights sum to 1.0.

To accommodate SAW within AutoRELAX, we augment Step (3) as shown in Figure 3.1 to incorporate the online SAW implementation. Fitness sub-function calculations are examined periodically (i.e., for this study, fitness was examined every 5^{th} generation, where this value was determined based upon empirical evidence to provide SAW enough time to explore different weighting schemes). To guide the AutoRELAX process towards combinations of RELAX operators that mitigate the uncertainties exhibited by the current operating context, the weight associated with the poorest-performing fitness sub-function is incremented. Incrementation of this particular weight pressures the AutoRELAX process to select appropriate RELAX operators for the current environment based on the shortcomings identified by AutoRELAX-SAW.

3.3 Case Studies

This section describes experimental results from two case studies in which AutoRELAX and AutoRELAX-SAW were both applied to the RDM and SVS applications, respectively, to demonstrate the effectiveness of these techniques in different application domains. First, we introduce the configuration of each simulation environment, including sources of uncertainty and possible reconfiguration states. Next, we present experimental results that compare and evaluate unRELAXed, manually RELAXed, and AutoRELAXed goal models. We then describe how SAW can augment AutoRELAX to determine an optimal weighting scheme that further improves AutoRELAX results. For each experiment, we state and evaluate our experimental hypotheses and resulting goal models.

3.3.1 RDM Case Study

For this experiment, AutoRELAX and AutoRELAX-SAW were applied to the RDM application. The RDM was configured as described in Section 2.3.2. The fitness functions described in Section 3.2.2 are used to guide the generation of RELAXed goal models.

3.3.1.1 RDM Uncertainty

We now compare and evaluate goal models that have been automatically RELAXed with two goal models: one in which we manually applied RELAX operators and one that was not RELAXed. The goal model used as a base for each experiment was previously presented in Figure 2.4. A small group of people with domain knowledge relevant to the RDM independently created first drafts of a manually RELAXed goal model and then collaborated to form a single, common goal model to be used in this experiment. The following RELAX operators were applied to the goal model: Goal (C) was RELAXed to tolerate dropped messages throughout the simulation; Goal (F) was RELAXed to allow for up to three simultaneous transient network partitions; Goal (G) was RELAXed to allow larger exposure to data loss; and Goal (H) was RELAXed to allow for extra time when diffusing data. These operators are summarized in Table 3.1.

We define two null hypotheses for this experiment. The first, $H1_0$, states that "there is no difference in fitnesses achieved by a RELAXed and an unRELAXed goal model." The second, $H2_0$, states that "there is no difference in fitnesses achieved between RELAXed goal models generated by AutoRELAX and those manually created by a requirements engineer." Also, we specified the fitness sub-function weights, α_{nrg} and α_{na}, to be 0.3 and 0.7, respectively. These values were chosen based on empirical evidence that it is nearly twice as important to minimize the number of adaptations when compared to minimizing the number of RELAXed goals.

Table 3.1: Summary of manually RELAXed goals for the RDM application.

Goal	Description	RELAX Operator	Boundaries
C	Achieve $[DataSent] == [NumberofDataCopies]$	`AS CLOSE AS POSSIBLE TO [quantity]`	$\pm$5.0 copies
F	Achieve $[NetworkPartitions] == 0$	`AS CLOSE AS POSSIBLE TO [quantity]`	$\pm$1.0 partitions
G	Achieve $[DataAtRisk] <= [RiskThreshold]$	`AS FEW AS POSSIBLE`	$\pm$5.0% risk
H	Achieve $[DiffusionTime] <= [MaxTime]$	`AS FEW AS POSSIBLE`	$\pm 5.0s$

Due to the variety of configurations available to the RDM at run time, not all adaptations result in the same fitness. Figure 3.4 presents boxplots that capture the fitness values achieved by AutoRELAXed goal models, a manually RELAXed goal model, and an unRELAXed goal model (the small circle on the plot indicates an outlier for the unRELAXed fitness values), where each experimental treatment was performed 50 times to achieve statistical significance. Despite the fitness boost that unRELAXed goal models have (i.e., 0 applied RELAXations for FF_{nrg}), goal models with applied RELAX operators achieve significantly higher fitness than unRELAXed goal models ($p < 0.001$, Welch Two-Sample t-test). These results enable us to reject $H1_0$ and conclude that RELAX reduces the number of necessary adaptations in the face of uncertainty.

Furthermore, Figure 3.4 also presents the difference in fitnesses between goal models that have been automatically generated and those manually generated. Specifically, AutoRELAX-generated goal models also achieve significantly higher fitness than those that are manually RELAXed ($p < 0.001$, Welch Two-Sample t-test), indicating that an automated approach for assigning RELAX operators to a system goal model provides an overall higher fitness value.

This result indicates that an AutoRELAXed goal model can better manage uncertainty than those that are manually RELAXed or unRELAXed.

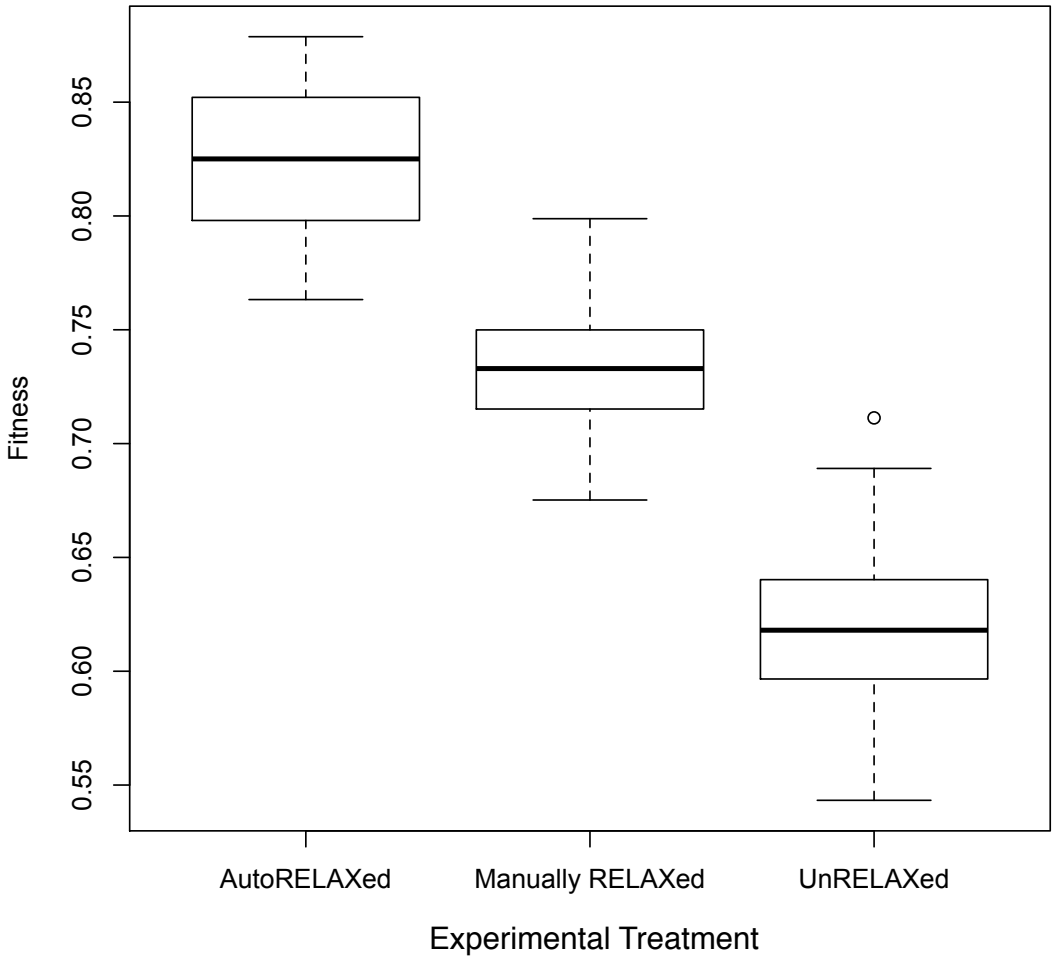


Figure 3.4: Fitness values comparison between RELAXed and unRELAXed goal models for the RDM.

Lastly, Figure 3.5 presents information regarding the types of goal model configurations generated by AutoRELAX for varying degrees of uncertainty. There are two specific types of uncertainty represented: low uncertainty and high uncertainty. The low uncertainty environment specifies a small probability that failure can occur, such as failed network links and dropped or delayed messages. The environment with high uncertainty specifies the opposite. As such, Figure 3.5 plots the *sorted* number of RELAXed goals per trial, where the first environment specifies a low degree of uncertainty and the second environment speci-

fies a high degree of uncertainty, and each point in the plot represents the average number of RELAXations applied throughout the RDM simulation. In all but one trial, AutoRELAX introduced greater than or equal amounts of RELAX operators to the goal model executed under the environment with high uncertainty than those executed under low amounts of uncertainty. Moreover, the positive correlation between the two curves indicates that AutoRELAX introduced RELAXations as necessary to mitigate increasing amounts of uncertainty, suggesting that unnecessary goal model flexibility is avoided.

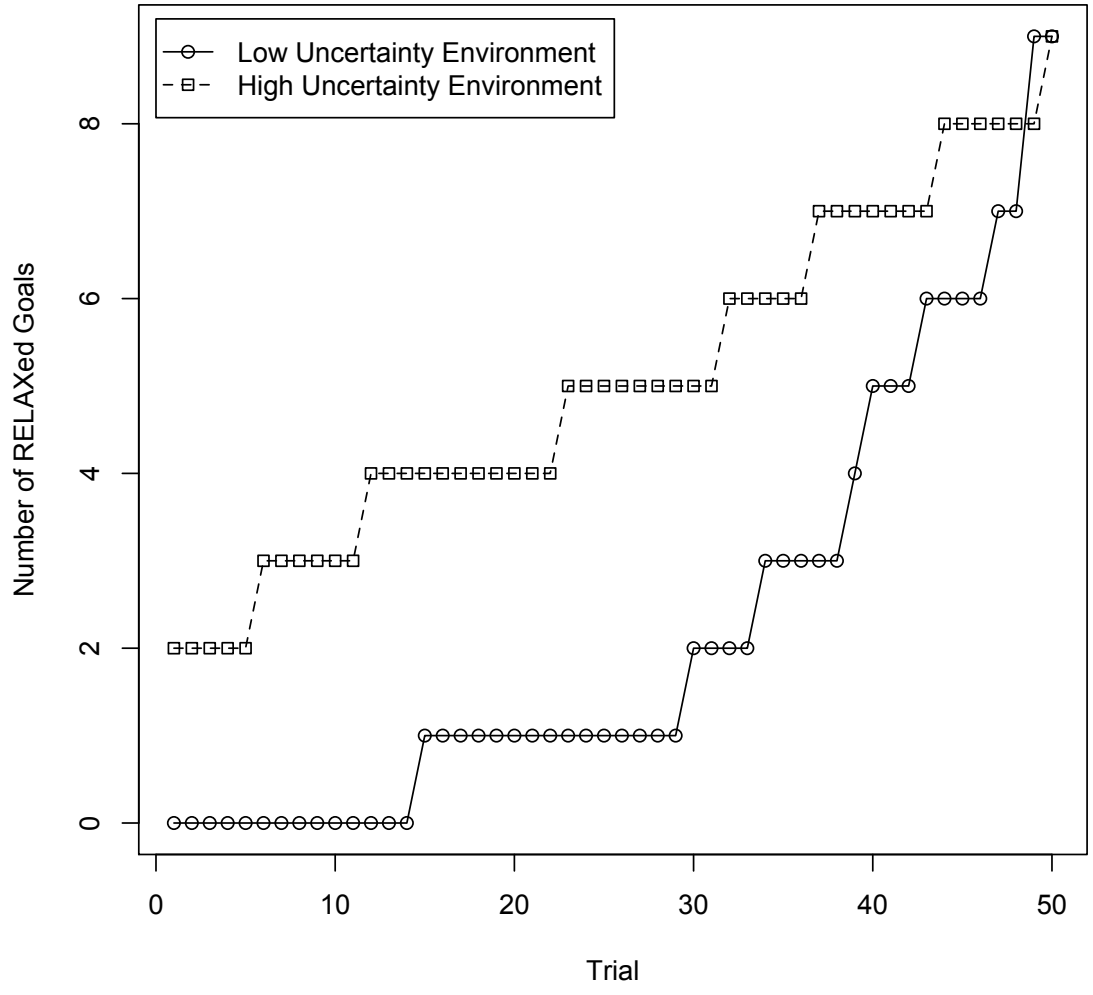


Figure 3.5: Mean number of RELAXed goals for varying degrees of system and environmental uncertainty for the RDM.

3.3.1.2 Dynamic Weight Adjustment

Next, we evaluate the effectiveness of optimizing the fitness sub-function weights. Specifically, we compare the calculated fitness values provided by AutoRELAXed goal models with those generated by AutoRELAX-SAW. AutoRELAX and AutoRELAX-SAW were each executed 50 times to achieve statistical significance. The fitness sub-function weights for AutoRELAX were set as defined in Section 3.3.1.1, and the weights for AutoRELAX-SAW were also seeded with these values.

We define two hypotheses for this experiment. First, $H1_0$ states that "there is no difference between an automatically RELAXed goal model with static weights and an automatically RELAXed goal model that uses a SAW-optimized weighting scheme across different environments." The second null hypothesis, $H2_0$, states that "there is no difference between RELAXed goal models with optimized and unoptimized weights in a static environment."

As such, Figure 3.6 presents two boxplots with fitness values generated by AutoRELAXed goal models and AutoRELAX-SAW-generated goal models. As is demonstrated by these plots, goal models that use an optimized weighting scheme achieve significantly higher fitness values than those that use a static set of weights ($p < 0.001$, Welch Two-Sample t-test). These results enable us to reject $H1_0$ and conclude that optimizing the fitness sub-function weighting scheme directly impacts AutoRELAX fitness.

For a small subset of environments, AutoRELAX-SAW converged to weights of $\alpha_{nrg} = 1.0$ and $\alpha_{na} = 0.0$, ensuring that the number of RELAXed goals is minimized. This result implies that RELAX operators are not always necessary if the goal model can sufficiently handle uncertainty without extra flexibility. Comparing the two populations, we can also reject $H2_0$ ($p < 0.001$, Welch Two-Sample t-test) and state that SAW-optimized goal models generally perform better than goal models without optimized weights.

Furthermore, AutoRELAX-SAW was compared to AutoRELAX in a static environment. Figure 3.7 presents boxplots of fitness values found by AutoRELAX-SAW over 50 separate trials, where the horizontal line represents the best AutoRELAX fitness value for that particular

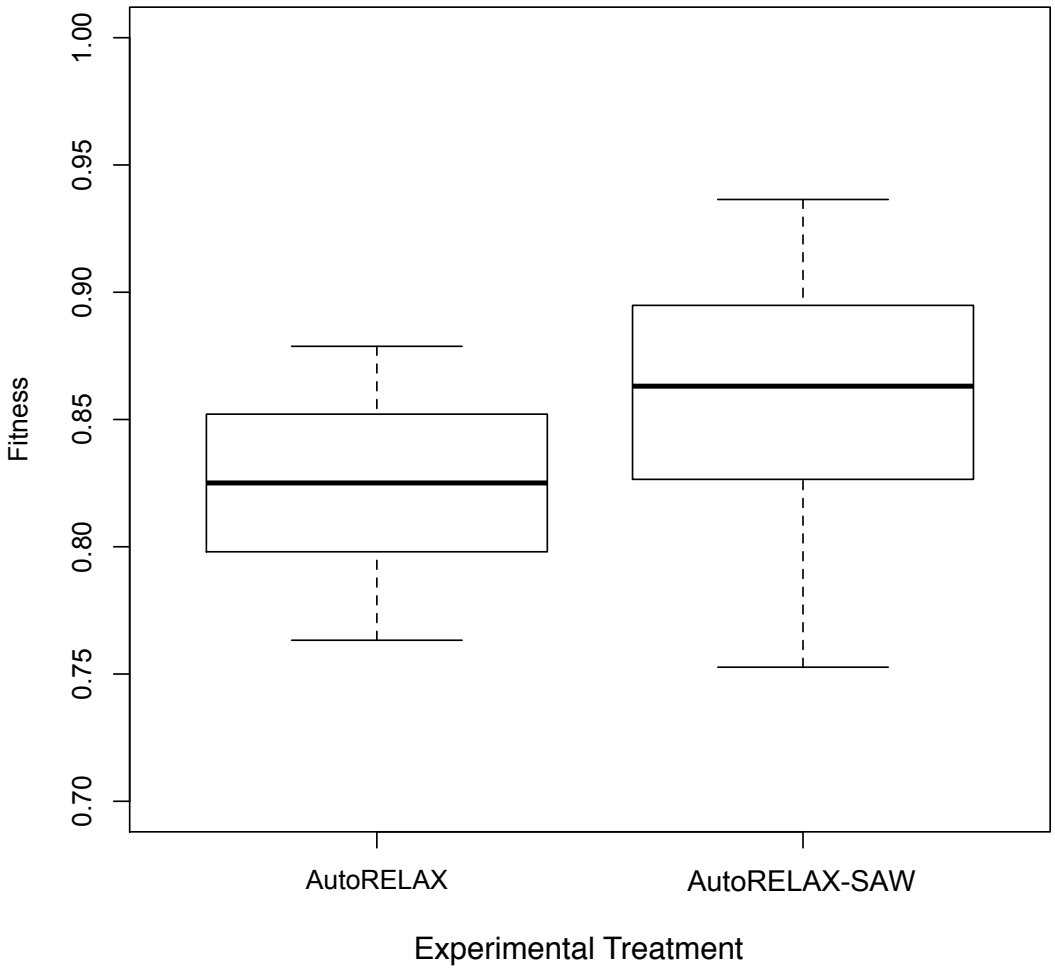


Figure 3.6: Fitness values comparison between AutoRELAXed and SAW-optimized AutoRELAXed goal models for the RDM.

environment. The results show that while AutoRELAX found a higher fitness value than the median fitness found by AutoRELAX-SAW, optimizing with SAW enabled the achievement of 100% fitness in this particular environment. However, AutoRELAX-SAW also found fitnesses lower than those found by AutoRELAX, indicating that not all weighting schemes are appropriate for a given environment. Each boxplot in Figure 3.8 represents the first 25 trials of a single environment experiment, where each boxplot presents the fitness over 50 replicates.

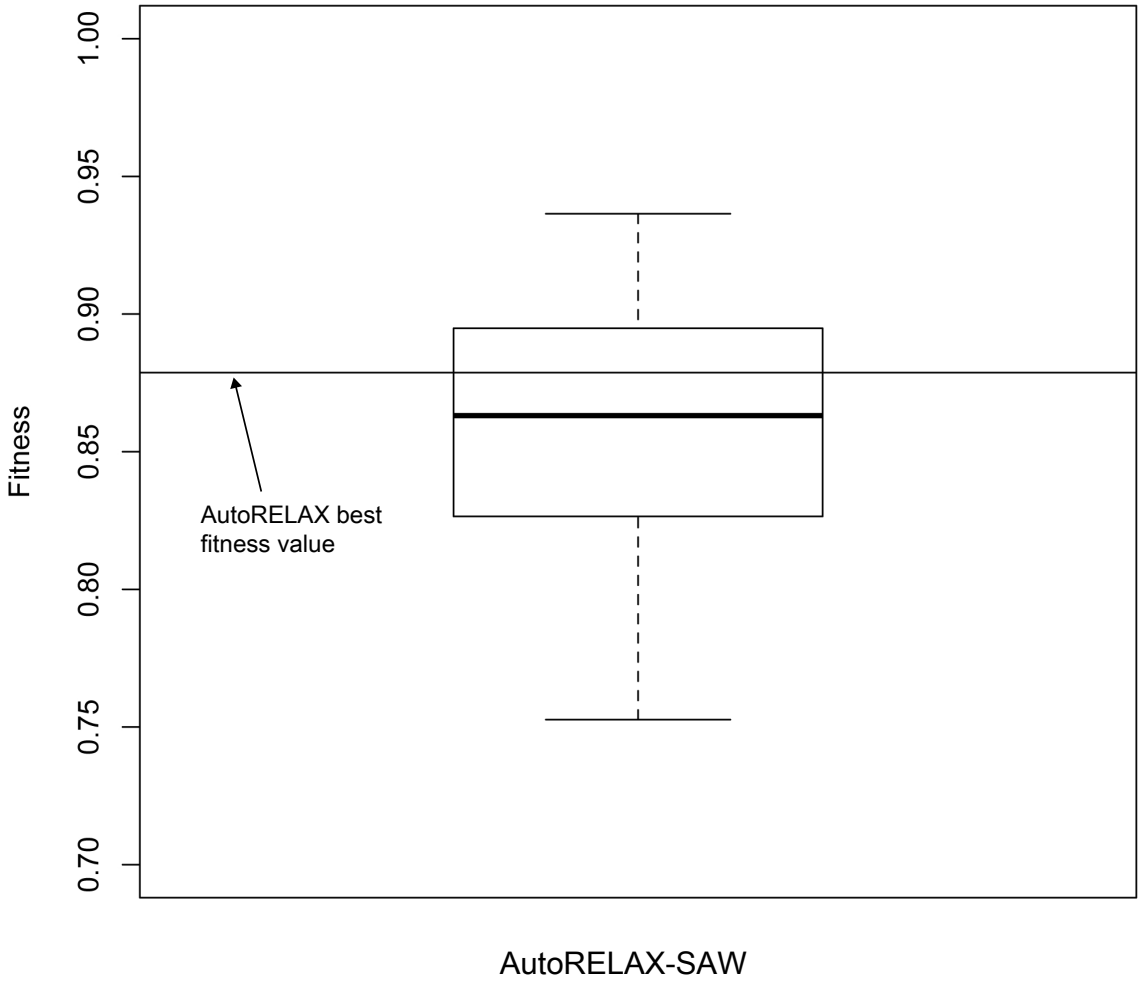


Figure 3.7: Fitness values of SAW-optimized AutoRELAXed goal models in a single environment.

3.3.2 SVS Case Study

We now describe a case study with AutoRELAX and AutoRELAX-SAW that uses the SVS as an application domain. The SVS was implemented as a fully autonomous robot tasked with cleaning a room, as described in Section 2.2.2. For this particular study, the SVS monitored utility values for each goal (c.f., Figure 2.2) at each time step to determine if a self-reconfiguration is necessary. The following section describes our experimental results from the SVS case study.

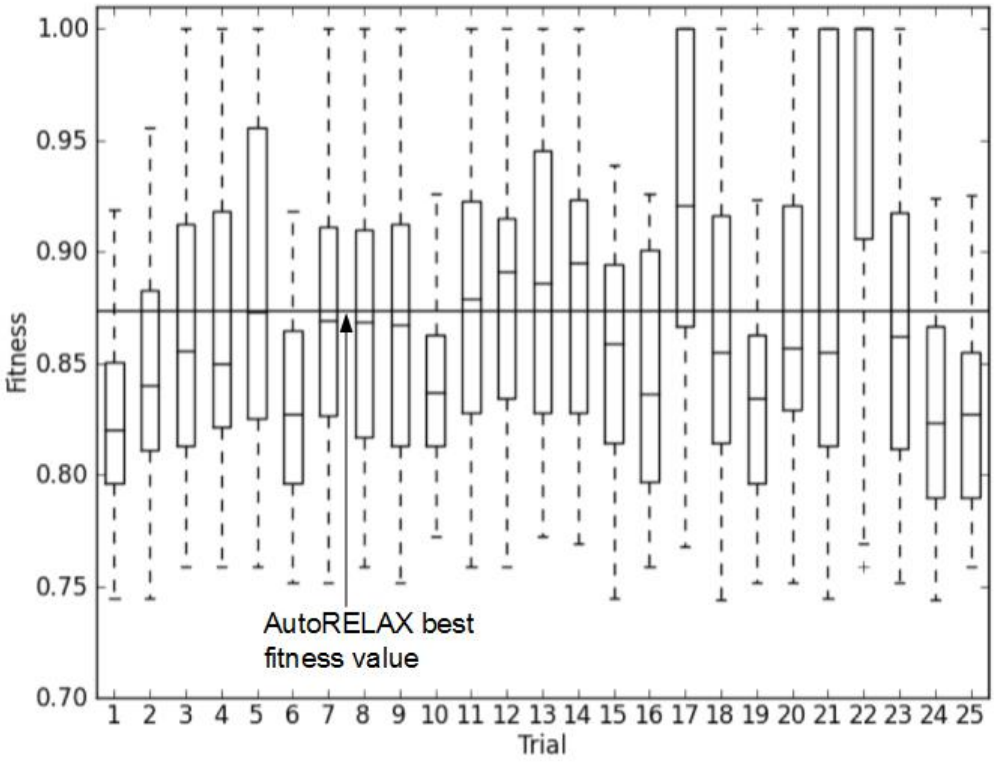


Figure 3.8: Comparison of SAW-optimized fitness values from different trials in a single environment.

3.3.2.1 SVS Uncertainty

The SVS case study also evaluates and compares unRELAXed, manually RELAXed, and AutoRELAXed goal models as a basis for experimentation. The manually RELAXed goal model had RELAX operators applied to: Goal (A) to allow the amount of remaining dirt to be larger than specified; Goal (B) to allow flexibility in the amount of remaining battery power; and Goals (F) and (G) to provide flexibility in the timing of each path plan. Table 3.2 describes the RELAX operators and fuzzy logic boundaries for the manually RELAXed goals.

The fitness function required an update for the SVS application. Specifically, two additional fitness sub-functions were added to maximize the cleanliness of the room (i.e., FF_{clean}) and minimize the number of faults encountered (i.e., FF_{faults}). The first additional sub-function, presented in Equation (3.4), maximizes the amount of dirt cleaned by the SVS:

Table 3.2: Summary of manually RELAXed goals for the SVS application.

Goal	**Description**	RELAX **Operator**	**Boundaries**
A	Achieve [90% Clean]	`AS MANY AS POSSIBLE`	±5.0%
B	Achieve [> 5% Power]	`AS MANY AS POSSIBLE`	±5.0%
F	Achieve [Path Plan for 30 seconds]	`AS CLOSE AS POSSIBLE TO [quantity]`	±15.0 seconds
G	Achieve [Clean Area for 1 minute]	`AS CLOSE AS POSSIBLE TO [quantity]`	±15.0 seconds

$$FF_{clean} = \left(\frac{|dirt\ removed|}{|amount\ of\ dirt|}\right) \tag{3.4}$$

where $|dirt\ removed|$ is the number of dirt particles vacuumed by the SVS and $|amount\ of\ dirt|$ represents the total number of initialized dirt particles. The second additional sub-function, Equation (3.5), rewards candidates that minimized the number of faults, such as running out of battery power before the simulation completed:

$$FF_{faults} = \left(\frac{1.0}{|faults|}\right) \tag{3.5}$$

where $|faults|$ represents the total number of faults encountered by the SVS during simulation. All fitness sub-functions can be combined into a linear weighted sum as follows in Equation (3.6):

$$Fitness\ Value = \begin{cases} [\alpha_{nrg} * FF_{nrg}] & + & \% \text{ number of RELAXed goals} \\ [\alpha_{na} * FF_{na}] & + & \% \text{ number of adaptations} \\ [\alpha_{clean} * FF_{clean}] & + & \% \text{ percentage of dirt removed} \\ [\alpha_{faults} * FF_{faults}] & + & \% \text{ number of faults} \end{cases} \tag{3.6}$$

where α_{clean} and α_{faults} are the weights associated with the additional sub-functions. Again, the sum of all weights must sum to 1.0. For this particular experiment, we have set $\alpha_{nrg} = 0.1$, $\alpha_{na} = 0.4$, $\alpha_{clean} = 0.2$, and $\alpha_{faults} = 0.3$.

For comparison purposes, we reuse the null hypotheses proposed in the RDM case study. Specifically, we define $H1_0$ to state that "there is no difference in fitnesses achieved by a RELAXed and an unRELAXed goal model," and $H2_0$ to state that "there is no difference in fitness values between RELAXed goal models generated by AutoRELAX and those manually created by a requirements engineer."

Figure 3.9 presents three box plots that demonstrate the overall fitness values calculated for AutoRELAX-generated goal models, a manually RELAXed goal model, and an unRELAXed goal model, respectively. Each experiment was performed 50 times to achieve statistical significance. As is demonstrated by the results, RELAXed goal models achieve significantly higher fitness values than unRELAXed goal models ($p < 0.001$, Welch Two Sample t-test), enabling us to reject $H1_0$ and conclude that applying RELAX operators to non-invariant goals can significantly reduce the number of encountered faults while mitigating uncertainty. Figure 3.9 also demonstrates that AutoRELAXed goal models achieve significantly higher fitness than manually RELAXed goal models ($p < 0.001$, Welch Two Sample t-test), allowing us to reject $H2_0$ and conclude that AutoRELAX can generate goal models that better mitigate system and environmental uncertainty than those created manually.

Figure 3.10 illustrates the impact that environmental uncertainty has on the application of RELAX operators. Specifically, this figure shows the *sorted* number of RELAXed goals for each trial in two separate environments. Each data point represents the mean number of RELAXed goals generated throughout the evolutionary process for each goal model. The first environment contained a low degree of uncertainty, and the second contained a high degree of uncertainty. Low uncertainty implies a low probability for the occurrence of failures, and high uncertainty implies the opposite. In all trials, AutoRELAX generated goal models with more RELAX operators in the environment with higher uncertainty. The gradual increase in

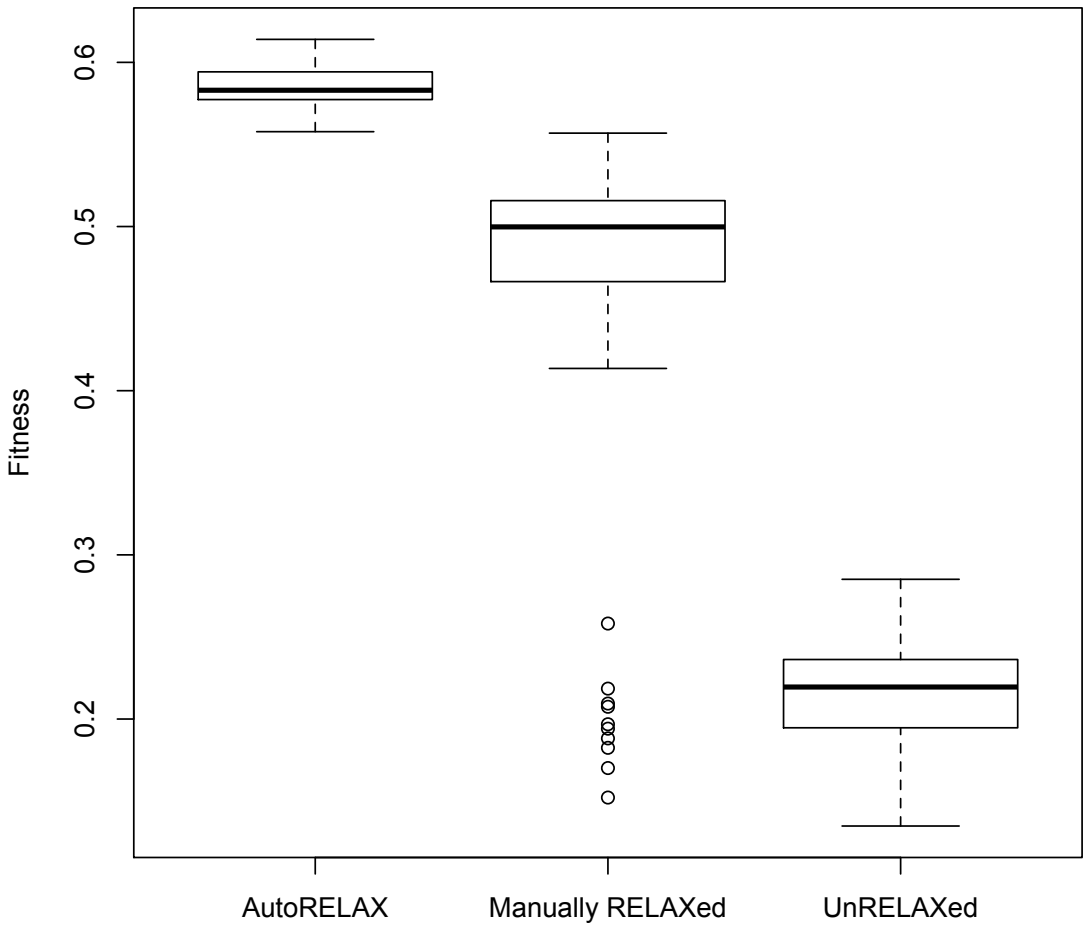


Figure 3.9: Fitness values comparison between RELAXed and unRELAXed goal models for the SVS.

slope of the curve again suggests that the application of goal RELAXations increases with the amount of uncertainty within the DAS and environment.

3.3.2.2 Dynamic Weight Adjustment

This next experiment explores how the weighting of fitness sub-functions impacts fitness as a result of environmental uncertainty. As such, we reuse the experimental setup as described for the SVS in the previous section. Furthermore, we define our first null hypothesis

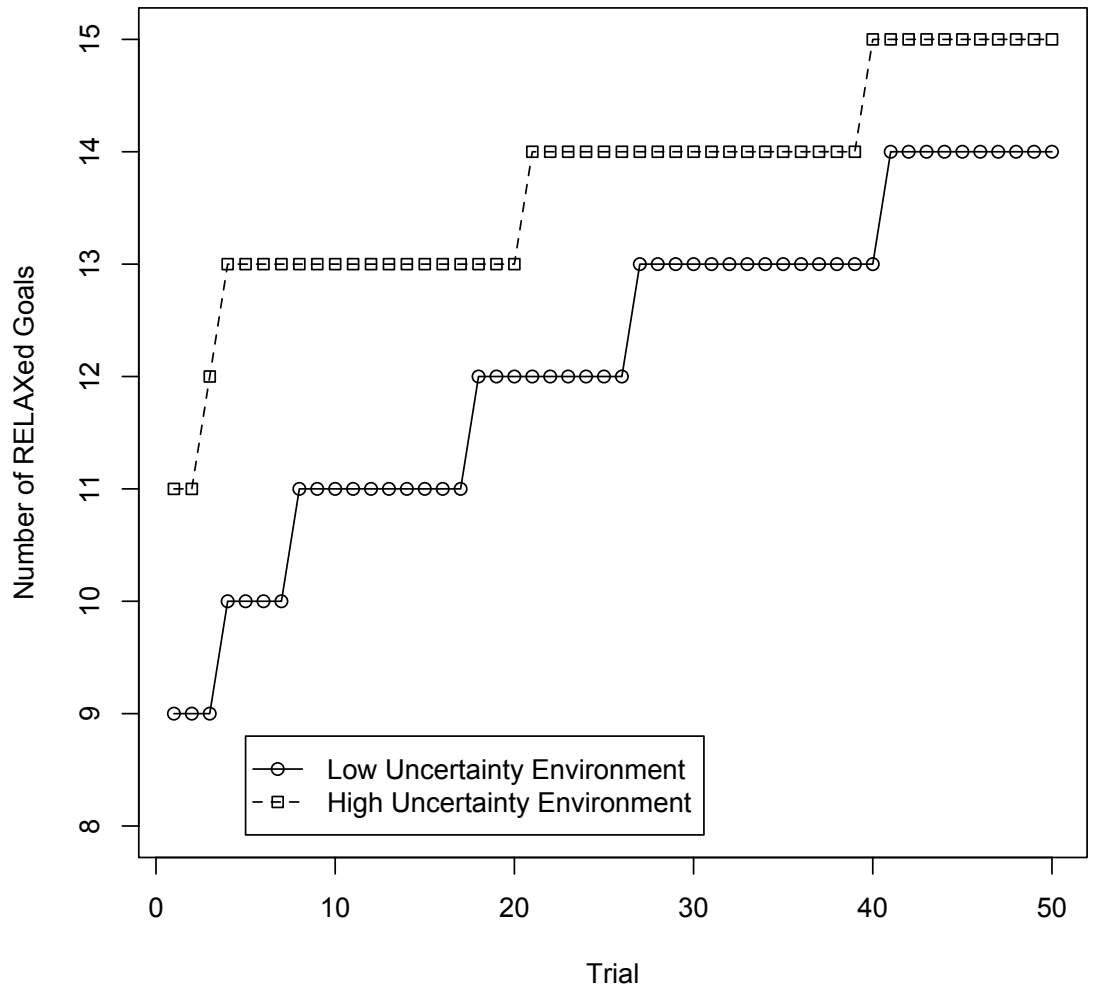


Figure 3.10: Mean number of RELAXed goals for varying degrees of system and environmental uncertainty for the SVS.

$H1_0$ to state that "there is no difference between an automatically RELAXed goal model with static weights and one that uses a SAW-optimized weighting scheme across different environments." We define a second null hypothesis $H2_0$ to state that "there is no difference between RELAXed goal models with optimized and unoptimized weights in a static environment." The SVS fitness sub-function weights were defined to emphasize the importance of reducing the number of encountered faults (see Equation (3.5) while minimizing the number

of performed reconfigurations (see Equation (3.4). Prior to dynamic adjustment, the weights were initialized as follows: $\alpha_{nrg} = 0.1$, $\alpha_{na} = 0.4$, $\alpha_{faults} = 0.3$, and $\alpha_{clean} = 0.2$.

Figure 3.11 presents boxplots that show fitness values calculated from AutoRELAXed and goal models generated with AutoRELAX-SAW, respectively. As such, AutoRELAXed goal models with optimized weights have a significantly higher fitness ($p < 0.001$, Welch Two-Sample t-test) than those without weight optimization. We can reject $H1_0$ and state that the weighting scheme has a direct impact on aggregate fitness for a goal model.

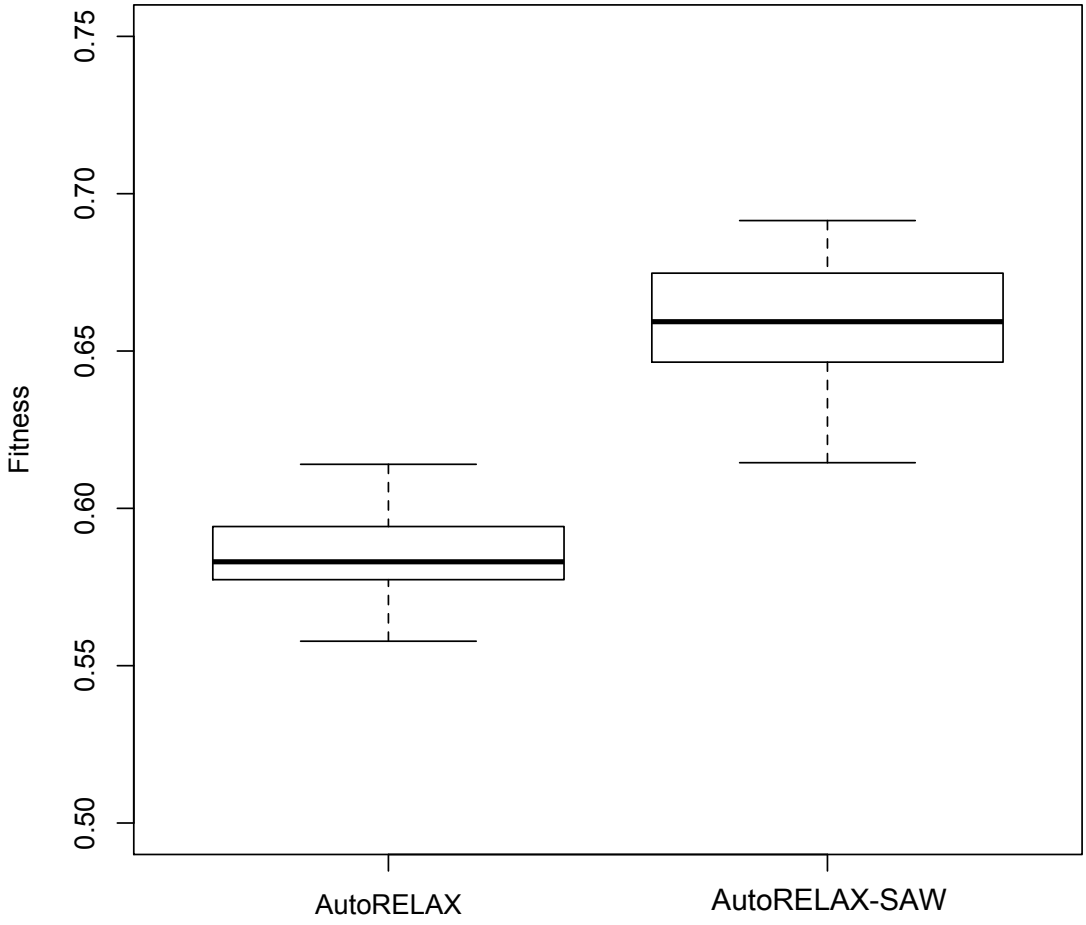


Figure 3.11: Fitness values comparison between AutoRELAXed and AutoRELAX-SAW goal models for the SVS.

Lastly, Figure 3.12 demonstrates a range of fitness values calculated from goal models with weights optimized for a static environment using AutoRELAX-SAW over 50 separate trials, where only the first 25 trials are depicted for presentation purposes. The solid horizontal line (at approximately 0.62) represents the best possible fitness generated by AutoRELAX over 50 separate trials. These results demonstrate that the environment directly impacts the weighting scheme of the fitness sub-functions, particularly in that a single weighting scheme may not be optimal for all environments. Moreover, Figure 3.12 demonstrates that different combinations of fitness sub-function weights yield different fitness values, indicating that AutoRELAX-SAW can be used to search for an optimal combination of weights, thereby finding an optimal RELAXed goal model for the particular environment.

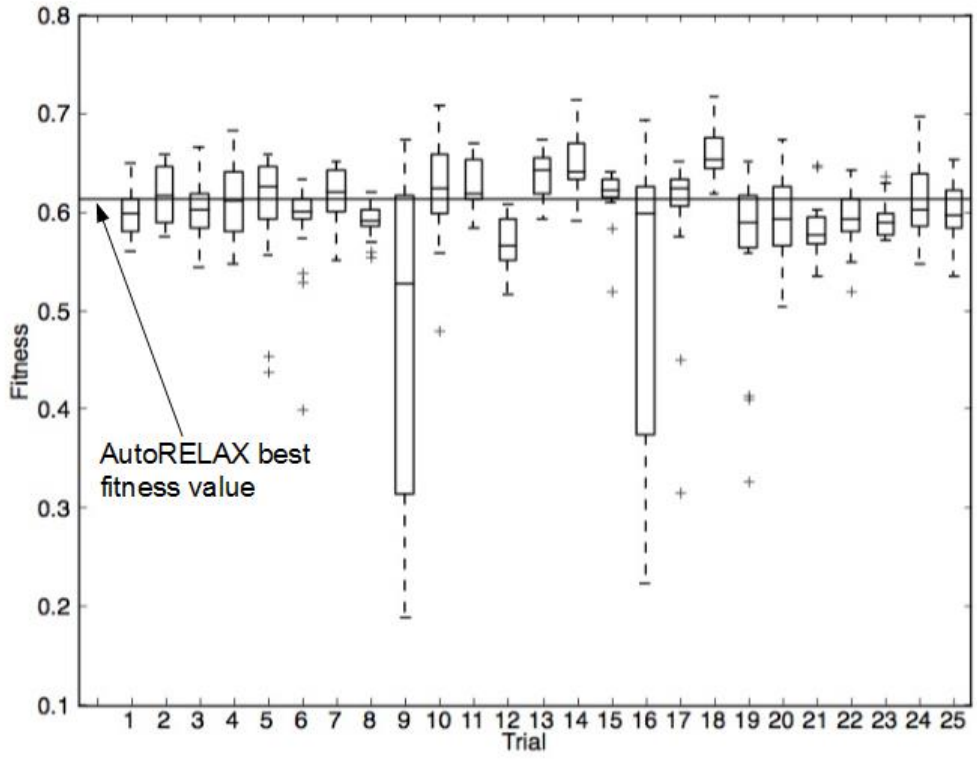


Figure 3.12: Comparison of SAW-optimized fitness values from different trials in a single environment.

3.4 Related Work

This section presents related work in specifying, detecting, and mitigating obstacles to the satisfaction of DAS requirements. In particular, we discuss related research for expressing uncertainty in requirements, performing requirements monitoring and reflection, and providing obstacle mitigation strategies.

3.4.1 Expressing Uncertainty in Requirements

Research into determining the impact of uncertainty, including assurance, affects the design, implementation, and execution of a DAS has been recently been the focus for the software engineering community [5, 8, 21, 32, 34, 35, 66, 86, 94, 111, 112], specifically in identifying and documenting sources of uncertainty. For instance, a *Claim* [111, 112] is a marker of uncertainty in requirements and design based upon possibly incorrect assumptions. Likewise, an analytical framework [34, 35] has been proposed that leverages mathematical approaches for modeling and assessing uncertainty from a risk-management perspective. Uncertainty in model checking [110] has also been studied, specifically in the realm of model variations. Probabilistic model checking [38] has also been explored to quantify unpredictable changes in models to mitigate unsatisfied requirements. While these approaches often require a requirements engineer to evaluate the sources and results of uncertainty, AutoRELAX can automate the identification and application process of RELAX operators to mitigate those sources of uncertainty.

3.4.2 Requirements Monitoring and Reflection

Requirements monitoring frameworks can detect the occurrence of obstacles in satisfying requirements [37]. In certain circumstances, such frameworks can also suggest mitigation strategies as necessary. Requirements have also recently been promoted to first-class, runtime entities whose satisfaction directly impact the execution of adaptive systems [8, 94]. Fur-

thermore, an awareness requirements-based feedback loop has also been introduced, where awareness requirements perform a meta-level monitoring of system requirements [100]. In contrast, these approaches do not directly support the management and run-time monitoring of RELAXed requirements.

3.4.3 Obstacle Mitigation

Existing strategies have been proposed to identify, analyze, and resolve obstacles that prevent requirements from being satisfied [105, 106], however the majority of these strategies focus on revising goals to either prevent or negate the effects of obstacles. For example, Letier *et al.* [65] have recently proposed an approach for designing software requirements while explicitly considering obstacles posed by uncertainty using multi-objective optimization and decision analysis. AutoRELAX can complement the proposed strategies by automatically determining the extent to which a non-invariant goal can be unsatisfied at run time, thereby enabling the design of a system that can continually function while exposed to obstacles that occur as a result of uncertainty.

3.5 Conclusion

This chapter described AutoRELAX, a technique for automatically generating RELAXed goal models, and AutoRELAX-SAW, a technique for automatically balancing fitness sub-function weights in response to environmental uncertainty while executing AutoRELAX. AutoRELAX leverages a genetic algorithm to explore how RELAX operators can be automatically applied to a goal model to provide an optimal combination of RELAXations that can effectively mitigate uncertainty in both the system and environment. Similarly, AutoRELAX-SAW also uses a genetic algorithm for optimization, however it, instead, uses AutoRELAX as a basis for evaluation. Both AutoRELAX and AutoRELAX-SAW were applied to the RDM and SVS applications for validation.

AutoRELAX experimental results indicate that RELAXations tend to correspond with specific types of uncertainty. For the RDM, Goals (F) and (T) were RELAXed most often, depending on whether network links had a higher probability of failure than it did for dropping messages, and vice versa. RELAX operators applied to Goal (J) tended to add extra flexibility with respect to the available time for goal satisfaction. For the SVS, Goals (D) and (E) tended to be RELAXed more often, as RELAXing those goals effectively extends the available operating time for the SVS. Adding the SAW optimization for both case studies tended to improve overall goal model fitness for a given environment, specifically, we determined that the weighting scheme is directly impacted by the environmental configuration in that fitness sub-function weights can pressure the search for an optimal RELAXed goal model for each given environment.

Potential future directions for AutoRELAX and AutoRELAX-SAW include exploring different approaches to handling multiple concerns when calculating fitness, such as multi-objective optimization [27], as well as exploring how different genetic operators can impact the search procedure. Lastly, exploring different types of fuzzy logic membership functions for RELAX operators is another future direction.

Chapter 4

Exploring Code-Level Effects of Uncertainty

This chapter presents Fenrir, a technique for exploring how uncertainty impacts the execution behavior of a DAS by examining the paths of execution taken throughout system execution, thereby enabling the identification of unexpected or anomalous behaviors [45]. Anomalous behaviors can then be corrected by applying bug fixes, augmenting target configurations, or updating system requirements. First, we motivate the need to explore DAS run-time behavior. Next, we introduce Fenrir, an evolutionary computation-based approach for exercising a DAS implementation by generating a novel set of operational contexts. We then describe the assumptions, required inputs, and expected outputs of Fenrir. Following, we present our case study in which Fenrir was applied to the RDM application. Lastly, we present related work and summarize our conclusions.

4.1 Motivation

A DAS can self-reconfigure as it executes to mitigate unexpected changes in the environment by triggering adaptive logic to switch between configurations. By doing so, the DAS can continually satisfy its requirements and high-level objectives even as its operational

context changes. However, if the DAS reconfigures in response to an unexpected situation, its resulting behavior can also be unexpected and may no longer satisfy its requirements. To this end, we describe an approach for exploring how a DAS reacts to both system and environmental uncertainty at the code level.

Fenrir supports our approach for mitigating uncertainty by providing code-level assurance. Specifically, Fenrir examines the resulting behaviors of a DAS experiencing system and environmental uncertainty to determine the paths followed by the DAS throughout execution. The amount of total execution paths is very large due to the adaptive nature of the DAS, as an adaptation can yield a completely unexpected DAS configuration. To explore the space of possible execution paths, Fenrir leverages novelty search to search for a representative set of operational contexts within the solution space. Fenrir then executes the DAS within each discovered operating context to determine its resulting behavior. The generated execution traces can then be analyzed by a DAS engineer to determine the effect that different types of uncertainty can have on the DAS, thereby enabling the identification of unanticipated or anomalous behaviors.

4.2 Introduction to Fenrir

In this section, we present and describe Fenrir, our approach to exploring how uncertainty affects a DAS within its implementation. First, we introduce the assumptions required to use Fenrir, as well as its required inputs and expected outputs. Next, we describe each step of the Fenrir approach in detail, illustrated by a DFD of the process. Lastly, we describe the results of a case study in which Fenrir was applied to the RDM application and compared with random search. Finally, we present related work in this domain, summarize our findings, and present future directions for research into Fenrir.

Thesis statement. *Complex software applications can follow unexpected paths of code execution when faced with run-time uncertainty.*

4.2.1 Assumptions, Inputs, and Outputs

In this section, we present the assumptions needed for implementing Fenrir. We then describe the required inputs to Fenrir and its expected outputs in detail.

4.2.1.1 Assumptions

We assume the following when instrumenting a DAS with Fenrir to explore the different paths of execution taken at run time:

- The DAS source code is available and can be instrumented with logging statements.
- Instrumenting the DAS does not significantly affect run-time behavior or performance of the DAS.
- The logging statements should provide a measure of code coverage and should monitor possible exceptions or error conditions.

4.2.1.2 Inputs and Outputs

Fenrir requires two elements as input. First, a prototype or executable specification of a DAS must be provided and instrumented with logging statements. Second, the domain engineer must specify all possible sources of uncertainty exposed to the DAS at run time. Fenrir then provides an archive containing a set of pairs as output, where each pair comprises a unique operating context and operational trace of DAS execution. Each input and output is next described in detail.

4.2.1.2.1 Instrumented Executable Specification

Fenrir requires an instrumented DAS to execute in order to trace its run-time behavior. Specifically, logging statements must be introduced into the DAS code to ensure that all possible adaptation paths and representative behavioral paths are represented (i.e., to

provide a base measure of DAS behavioral coverage). Furthermore, variables identified as relevant to the DAS operating context should also be monitored by the logging statements to capture the state of the operating context that led to each expressed behavior.

4.2.1.2.2 Sources of Uncertainty

The DAS engineer must specify the expected sources of uncertainty to simulate while executing Fenrir (e.g., severed network links or sensor failures). Specifically, the probability of occurrence and the severity of each uncertainty must be defined (e.g., [5%, 8%], [10%, 15%], ..., [$probability_n$, $severity_n$]), as they are required when configuring the novelty search algorithm. The probability of occurrence defines the chance that the source of uncertainty will manifest and the severity defines how much the source of uncertainty will impact the DAS. For example, a source of uncertainty may impact the ability of an RDM sensor to monitor a data mirror. In this case, we can state that this source of uncertainty defines the number of RDM sensors that may fail during execution. As such, we define the probability of occurrence to be 3%. This probability states that, for each timestep of execution, there is a 3% random chance that this source of uncertainty occurs. Next, we define the severity of occurrence to be 25%. In this case, the severity defines an upper bound on the amount of RDM sensors that can fail. In this case, up to 25% of all RDM sensors may fail when this source of uncertainty is activated.

4.2.1.2.3 Fenrir Archive

Fenrir generates a collection of operational contexts, each with a corresponding execution trace that details the path of execution followed by the DAS. Each operational context specifies the information necessary to recreate the context in simulation, comprising the sources of uncertainty with likelihood of occurrence (e.g., 10% chance of occurrence that network links will be severed) and severity (e.g., up to 15% of all network links affected) that are specific to that particular context. Furthermore, each context may trigger adaptations within

```
(1)  data.size                    = size;
(2)  mirror.remaining_capacity = capacity;
(3)
(4)  // log the variable state for this data mirror
(5)  logger.print('data mirror:[%d], data size:[%f],
(6)                remaining capacity:[%f]', data.id,
(7)                data.size, remaining_capacity);
(8)
(9)  // send data from data mirror
(10) if (data.size <= remaining_capacity)
(11) {
(12)   remaining_capacity -= data.size;
(13)   return true;
(14) }
(15) else
(16)   return false;
```

Figure 4.1: Example code to update data mirror capacity.

the DAS, and as such the trace provides information specific to each explored path. Path information comprises the invoking module, line number, description of intended behavior, and a flag that indicates the presence of an error or exception.

For instance, a module within the RDM application that is concerned with updating the remaining capacity for a data mirror can be instrumented as follows in Listing 4.1:

Executing the RDM under a particular operating context would yield the following order of statement execution: `[(1), (2), (12), (13)]`. In this case, the size of the particular data message is smaller than the remaining capacity of the data mirror, thus enabling a

successful transmission to another node. However, to reduce the size of the resulting trace,[1] only a single logging statement is necessary to capture the intended behavior of this sequence of operations. As such, the sample logging statement would appear as follows (for presentation purposes, only the relevant variable information has been displayed; a full logging statement comprises a unique identifier, the current module name, line number, and description including any relevant variable values):

```
data mirror:[2], data size:[2.5], remaining capacity:[4.0]
```

This particular trace statement enables a DAS developer to determine the behavior of a particular data mirror at a particular point during execution. Each branch of the `if-else` block did not require instrumentation in this case, as DAS behavior can be inferred based upon the monitored variables.

4.2.2 Fenrir Approach

This section details the Fenrir approach. Figure 4.2 shows the DFD that illustrates Fenrir. Each step is next described in detail.

(1) Generate Operational Contexts: Fenrir accepts the sources of uncertainty, as specified by a DAS engineer, and in turn uses novelty search [64] to generate a set of operational contexts, where each operational context specifies the configuration of the sources of uncertainty, the configuration of the RDM, and the configuration of the environment. These contexts are then applied to a DAS that has been instrumented with logging statements to determine how different configurations of uncertainty affect the DAS throughout execution, specifically in which paths of execution the DAS follows as it executes and self-reconfigures in response to those uncertainties.

Each operational context is represented as a genome, where a set of genomes in turn represents a population. Each genome comprises a vector of length n, where n defines the

[1] Full coverage of all possible branches resulted in file sizes greater than 10 GB and required optimization to ensure comparisons between logs would be manageable.

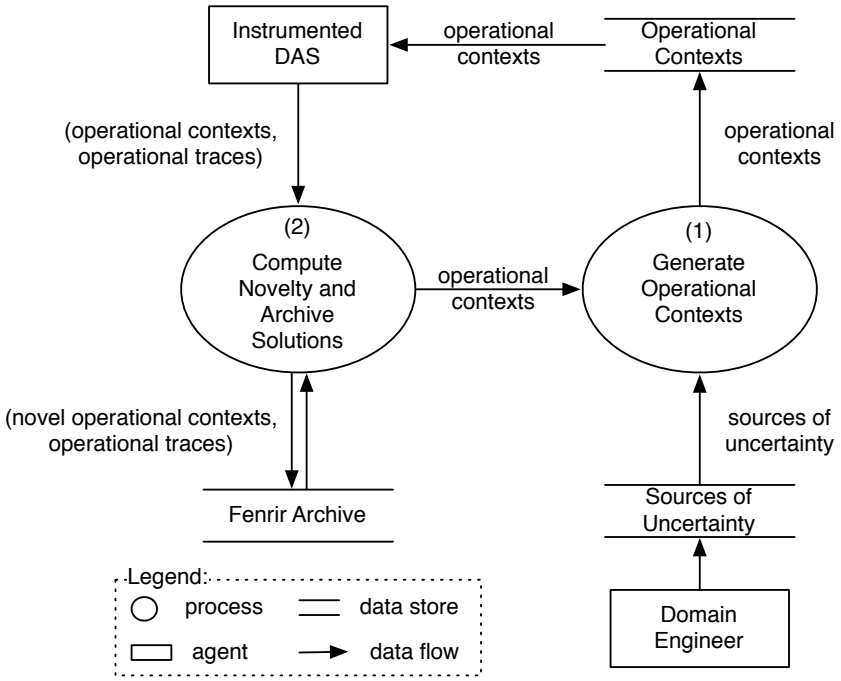


Figure 4.2: Data flow diagram of Fenrir approach.

number of uncertainty sources from the system and the environment. Each gene defines the likelihood of occurrence and the severity of the occurrence, representing how often the particular uncertainty is likely to occur and how much of an impact it will have on the DAS. Figure 4.3 presents a sample Fenrir genome containing three sources of uncertainty specific to the RDM application: dropped message, delayed message, and link failure. In this case, there is a 15% chance that a link failure will occur, and upon occurrence, can affect up to 10% of all network links.

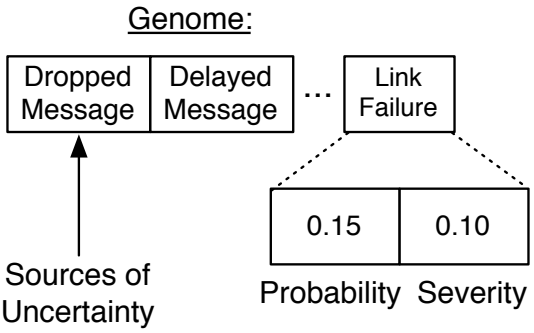


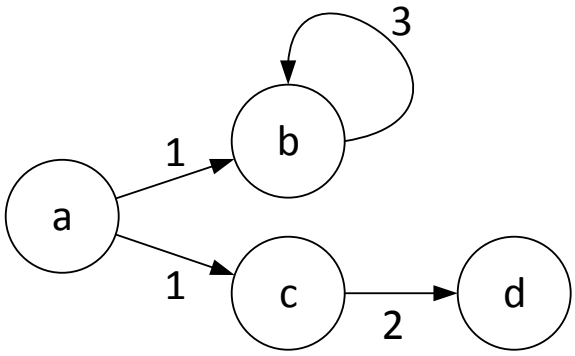
Figure 4.3: Sample Fenrir genome representation for the RDM application.

Following the generation of operational contexts in each generation, Fenrir then applies each operational context to the instrumented DAS, resulting in an execution trace for each context. The set of pairs (*operational context*, *execution trace*) is then provided as input to the novelty value calculation, as is described in the following section.

(2) Compute Novelty and Archive Solutions: Following execution of all operational contexts within a population, Fenrir then calculates a novelty value for each solution. First, Fenrir constructs a *weighted call graph* (WCG) [1] by parsing each execution trace, where a WCG is an extension to a program call graph [90]. Each WCG is represented as a directed graph, where each node has a unique identifier that corresponds to a particular logging statement, the directed edges between nodes symbolize the order in which logging statements were executed, and a weight on each edge represents execution frequency. Figures 4.4(A) and 4.4(B) illustrate a sample WCG with corresponding code statements. In this example, Statement (a) was executed first, and only a single time. Based on the `if` statement within the `wrapper` function, either Statement (b) or (c) can then be executed next. Here, Statement (b) was executed three times, and Statement (c) was executed twice. Lastly, Statement (d) is executed a single time following Statement (c).

The novelty value for an individual is calculated based upon the difference in nodes and edges between two WCGs, as is shown in Equation (4.1), and then by applying the Manhattan distance metric [10] to measure the distance between every other WCG in both the current population and the novelty archive, as shown in Equation (4.2). In Equation (4.1), v represents a vertex or node, e represents an edge, μ_i and μ_j represent the WCGs being compared, and g_i and g_j represent the graphs that define each WCG. In Equation (4.2), k represents the number of WCGs to compare.

Any individual with a novelty value that either exceeds the novelty threshold, or falls within the top 10% of all novelty values (i.e., novelty threshold), is then added to the novelty archive at the end of each generation. The novelty archive, in turn, stores the k most diverse individuals throughout the evolutionary process.



(A) Weighted call graph.

```
   void main() {
a:   wrapper(TRUE);
   }

   void wrapper(bool flag) {
     int i, k = 0;
     if (flag)
b:     while (i < 3) ++i;
     else {
c:     while (i < 2) ++i;
d:     callFunction(i,k);
     }
   }
```

(B) Corresponding code.

Figure 4.4: Example of weighted call graph.

$$dist(\mu_i, \mu_j) = len(\{v \in g_i\} \oplus \{v \in g_j\}) + len(\{e \in g_i\} \oplus \{e \in g_j\})) \tag{4.1}$$

$$p(\overline{\boldsymbol{\mu}}, k) = \frac{1}{k} \sum_{i=1}^{k} dist(\mu_i, \mu_j) \tag{4.2}$$

Following completion of novelty search, Fenrir returns the novelty archive containing a set of pairs (*operational context*, *execution trace*) that, based on their novelty score, comprise

the most diverse individuals found by Fenrir. Each pair provides insight into the behavior of the DAS throughout its execution, such as adaptation triggers, parameters for each instantiated DAS configuration, and any raised exceptions, unnecessary adaptations, or unacceptable behaviors that occurred, where an unnecessary adaptation refers to thrashing between DAS configurations before finally settling on a target configuration. This type of detailed runtime behavior can only be found by reviewing execution traces. Unacceptable behavior can then be corrected by providing bug fixes, augmenting target configurations, updating system requirements, or introducing methods for adding flexibility to system requirements, such as RELAX [21, 113].

4.3 RDM Case Study

This section presents a case study in which Fenrir is applied to the RDM application. First, we introduce the configuration of the simulation environment. Next, we present experimental results in which DAS execution behavior is examined within uncertain environments.

For this experiment, Fenrir was applied the RDM application that was configured as described in Section 2.3.2. Equations (4.1) and (4.2) were used to guide the search procedure towards novel areas of the solution space. The RDM network contained 25 RDMs with 300 network links. Each logging statement added to the DAS comprises a unique identifier and module information such as function name, line number, and description. The RDM was executed for 150 time steps and 20 data items were randomly inserted into the network throughout the simulation. The novelty search algorithm was configured as follows in Table 4.1:

To validate our approach, we compared the results of novelty search with a random search algorithm. We chose random search as we do not know *a priori* which operational contexts will adversely affect the system. As such, we generated 300 random operational contexts to compare against Fenrir. For statistical purposes, we conducted 50 trials of each

Table 4.1: Novelty search configuration.

Parameter	**Value**
Number of generations	15
Population size	20
Crossover rate	25%
Mutation rate	50%
Novelty threshold	10%

experiment and have plotted mean values and error bars. The following section describes the experiment in detail, including our hypotheses and experimental results.

4.3.1 DAS Execution in an Uncertain Environment

To validate our approach, we compared Fenrir with random search to explore the space of possible operational contexts. Specifically, we defined a null hypothesis H_0 to state that "there is no difference in execution traces generated by configurations produced by novelty search and those created by random search." Furthermore, we defined the alternate hypothesis H_1 to state that "there is a difference in execution traces generated by configurations produced by novelty search and those created by random search."

Figure 4.5 presents the results of our experiment. Specifically, boxplots with novelty distances for each member of the novelty archive generated by Fenrir and novelty distances calculated by a randomized search algorithm. This plot demonstrates that Fenrir-generated operational contexts yield execution traces that achieve significantly higher novelty values than are generated by random search ($p < 0.001$, Welch Two-Sample t-test). Furthermore, Fenrir-generated execution traces that exhibited negative kurtosis, suggesting that the distribution of operational contexts was skewed towards larger novelty values. We can then reject H_0 based on these results. Furthermore, we can also accept H_1, as novelty search discovered a larger number of unique DAS execution paths in comparison to random search.

Furthermore, Figure 4.5 also demonstrates that Fenrir can generate a better representation of the search space with fewer operational contexts. This conclusion is attained by the fact that the Fenrir boxplot only contains novelty values from 30 individuals, whereas the random search boxplot contains novelty values from 300 individuals. As a result, a DAS developer can use Fenrir at design time to assess DAS behavior in the face of uncertainty.

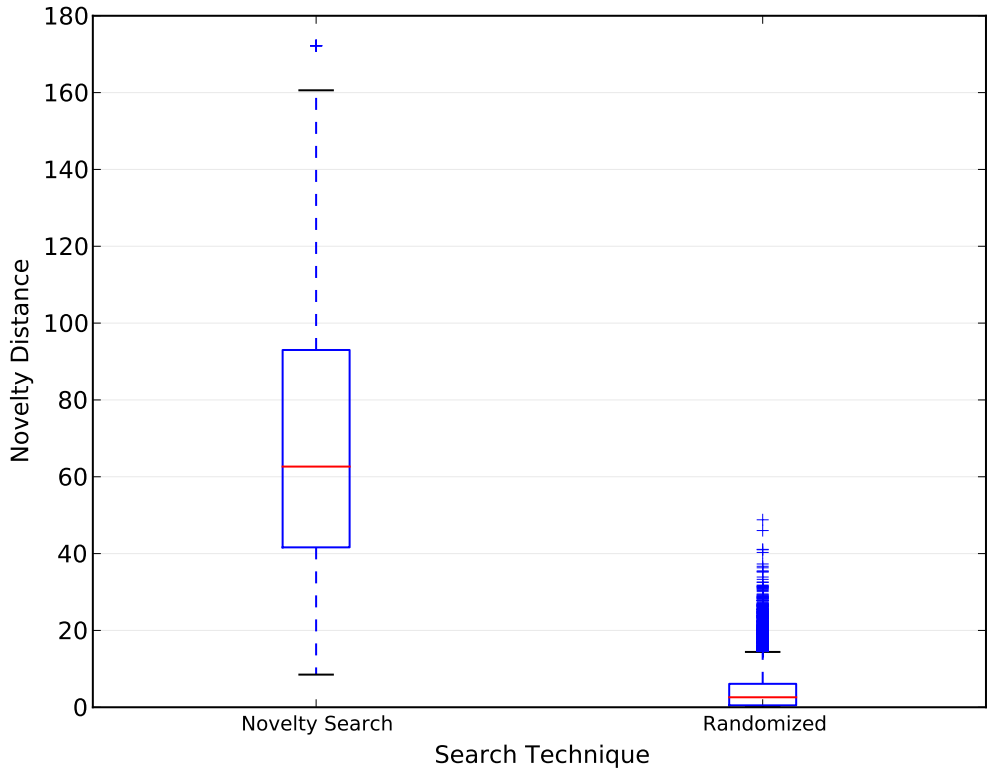
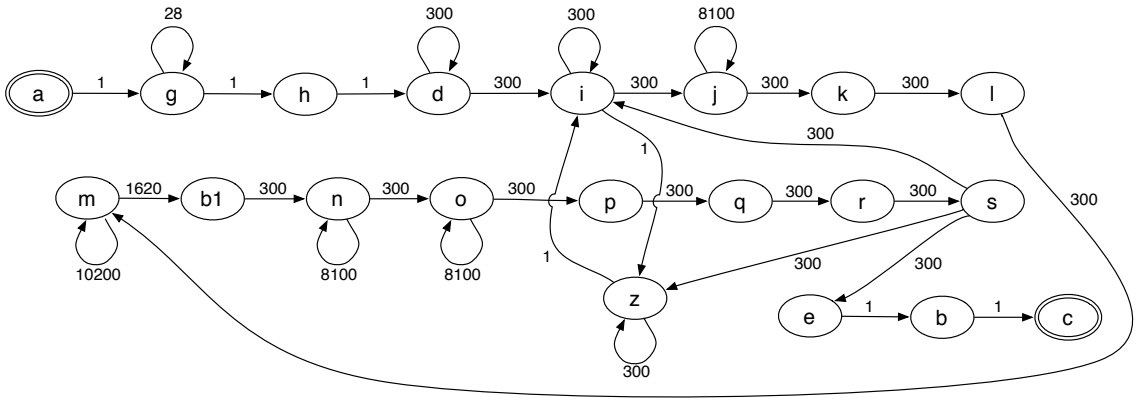


Figure 4.5: Novelty value comparison between Fenrir and random search.

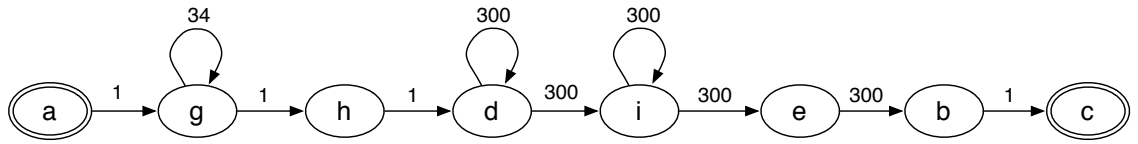
Figure 4.6 presents two sample RDM execution paths produced by execution traces generated by Fenrir-created operational contexts. Each node represents a unique logging statement, each edge represents a sequential execution from one node to the next, and each weight represents the frequency that the statement was executed. For example, Figure 4.6(A) demonstrates that Node (g) was executed 28 times and then Node (h) was executed once, based on the transition between Nodes (h) and (g). Further analysis of Figure 4.6(A) indi-

cates that the RDM network comprised 28 instantiated data mirrors (i.e., Node (g) logs a function that instantiates data mirrors).

Visual inspection of Figures 4.6(A) and 4.6(B) indicate that Fenrir can generate operational contexts that cause a DAS to exhibit wildly different execution paths in the face of uncertainty. The diversity of these traces enables the DAS developer to determine focal points in the DAS functional and adaptation logic to perform optimizations such as reducing complexity, optimizing target configurations, and repairing erroneous behaviors.



(A) Execution Path 1



(B) Execution Path 2

Figure 4.6: Unique execution paths.

4.3.1.1 Threats to Validity

This research was intended to be a proof of concept to determine the feasibility of using DAS execution trace data to examine how the DAS reacts to different sources of uncertainty. Threats to validity include whether this technique will achieve similar results in other application domains, such as embedded systems. To limit the resulting size of

generated trace data, we focused on coverage points in the DAS, rather than providing full branch coverage. As a result, additional logging statements may be required to provide a full representation of DAS run-time behavior.

4.4 Related Work

This section presents related work on research into code coverage efforts, methods for automatically testing distributed systems, and automatically exploring how uncertainty impacts software requirements.

4.4.1 Code Coverage

Code coverage testing provides a measure of assurance by defining a metric to which the different branches of code are verified and validated. Moreover, code coverage can be used to determine how uncertainty impacts a DAS codebase in terms of unexpected adaptations or unwarranted behaviors. Chen *et al.* [18] previously proposed code coverage as a method for enhancing reliability of a software system while testing is being performed, as a system may contain latent errors in areas of code that have not been exercised even following a successful testing phase. However, instrumenting a software system to provide code coverage analysis is non-trivial, as each separate branch of execution must be examined for faults. Previously, Tikir and Hollingsworth [102] introduced a technique for dynamically adding and removing logging calls into a codebase to ease the impact of instrumenting code on the developer. Furthermore, logging optimizations were introduced to reduce the total number of instrumentation points as well as the expected overhead incurred by logging.

While each of these approaches facilitates code coverage testing, Fenrir instead targets code segments that specifically cause the DAS to follow different paths of run-time execution. These segments include branching paths and self-reconfiguration code statements, and can provide a finer-grained representation of DAS behavior throughout execution.

4.4.2 Automated Testing of Distributed Systems

Distributed systems are generally composed of asynchronous processes or applications that can send and receive data asynchronously. Distributed systems are similar to DASs in that they are complex software systems that require constant communication to properly function. As a result, distributed systems can exhibit a staggeringly large number of possible execution paths. Sen and Agha [98] previously introduced the notion of *concolic* execution, or simultaneous concrete and symbolic execution, to determine a partial order of events that occur throughout system execution. Concolic execution has been proven to exhaustively find all possible execution paths of a distributed system. In the DAS domain, concolic execution could be used to examine the space of possible DAS adaptations and adaptation triggers, enabling an in-depth analysis of DAS behavior in uncertain environments. Conversely, Fenrir does not exhaustively examine all possible execution paths, instead opting to examine a representative set of behaviors exhibited by executing a DAS under disparate environments. While concolic execution can find all possible paths of execution, Fenrir can be used to determine a DAS's reaction to different operating contexts and sources of uncertainty.

Automated fault injection is an approach to facilitate fault tolerance by subjecting a distributed system to problems at run time, thereby enabling a system to provide mitigation behaviors to continually satisfy its requirements [26] and ensure dependability [97]. Fenrir instead explores how a DAS handles faults by examining its reaction to various types of operational contexts, rather than directly injecting faults for the system to mitigate.

4.4.3 Automatically Exploring Uncertainty in Requirements

Uncertainty impacts software at all levels of abstraction, including the requirements level. Ramirez *et al.* [86] proposed Loki as an approach to create a set of unique operating conditions that impacts a DAS, thereby attempting to uncover latent or unexpected errors in the controlling requirements specification. This task is accomplished by using novelty search

to generate conditions that cause the DAS to exhibit different behaviors throughout execution. Fenrir extends Loki by exploring uncertainty at the code level, specifically examining the path of execution followed by the DAS in response to different types of uncertainty.

4.5 Conclusion

This chapter described Fenrir, a design-time approach for providing code-level assurance for a DAS in the face of uncertainty. Fenrir leverages novelty search [64] to automatically generate a diverse set of operational contexts that can affect how a DAS executes. Specifically, Fenrir introduces logging statements into the DAS codebase to determine when and how often sections of code are executed. Fenrir calculates the distance between execution paths by translating program logs into WCGs and then measures the difference in nodes and edges to determine a resulting novelty value. Using novelty search, Fenrir searches for a set of operational contexts that provide a broad range of execution traces, effectively exercising the DAS to determine if it exhibits undesirable behaviors or contains inconsistencies between its requirements and implementation.

Fenrir was applied to the RDM application and subjected to uncertainties in the form of dropped or delayed messages and network link failures. Experimental results determined that Fenrir can generate operational contexts that are more novel than can be found by random search, and moreover can generate the set of contexts in an efficient manner. Future directions for this work include application to additional domains, investigating the use of other types of distance metrics for the novelty value calculation, and exploring how other search-based or evolutionary algorithms can be leveraged to explore the operational context solution space.

Chapter 5

Run-Time Testing of Dynamically Adaptive Systems

This chapter proposes a run-time testing feedback loop that provides online testing assurance for continual satisfaction of DAS requirements [44]. First, we motivate the need for run-time testing in DASs. Next, we present MAPE-T, a feedback loop to enable run-time testing. For each element of the MAPE-T loop, we describe in detail the key objectives, expected functionalities, and key challenges that we anticipate to manifest. Moreover, we describe enabling technologies that could be used or extended to realize each element. For each component, we also highlight an example of each component within the context of the SVS application. Next, we overview related work. Finally, we summarize our proposed framework.

5.1 Motivation

While testing a DAS during the design and implementation phases of the software development cycle addresses verification and validation for a DAS, the derived test cases are often static in nature and may not adequately assess unexpected system or environmental conditions [20, 61, 68, 93, 94, 112]. Furthermore, test cases that have been derived for the

anticipated environments may quickly lose relevance as the DAS self-reconfigures to address fluctuations within the environment. For instance, test cases that have been derived to test an SVS under conditions in which large objects must be avoided are only applicable if those large objects exist within the current operational context. Otherwise, executing this particular test case is superfluous and unnecessarily expends system resources.

Traditional testing techniques provide assurance that the DAS is satisfying its requirements by identifying and testing the DAS against anticipated operating conditions. However, it is often unfeasible for an engineer to identify all possible conditions that the DAS may face over its lifetime, leading to possible requirements violations and system failures as a result of untested DAS execution states. Extending traditional testing techniques to the run-time domain enables a DAS to determine if the operating context will cause a negative effect on DAS operation, and if possible, reconfigure the DAS to mitigate the adverse conditions. Moreover, the results of run-time testing could be used to trigger a DAS adaptation, thereby strengthening the DAS against adverse conditions.

Several challenges must be overcome to test a live system as it executes. First, understanding how and when to *generate test cases* is a concern, as providing a representative set of test cases is necessary when validating a system. Next, determining *when to test* is a concern that impacts DAS performance, as testing activities may consume system resources required for normal DAS operation. Then it is necessary to select an appropriate *testing method*, as the type of testing can focus on a particular aspect of system validation (e.g., regression analysis, structural testing, etc.). Finally, determining the *impact* of test results, and moreover, how to *mitigate* the impact is crucial to ensure the continual satisfaction of DAS requirements, as the execution and results of testing activities may cause an unexpected impact to DAS operation. Each of these challenges is next described in turn.

5.1.1 Test Case Generation

A test case is intended to determine if a system requirement, objective, or expectation is being satisfied or violated. Specifically, a test case explores how different ranges, boundaries, and combinations of input values can affect the behavior of software modules. IEEE provides a standard definition of a test case [54] that comprises three parameters: an input value, an expected output value, and a set of conditions necessary for the test case to be properly executed. Table 5.1 presents an example of an SVS test case as defined according to the IEEE standard. In this example, a test case has been defined to monitor the effective sensing radius of a camera sensor. According to a requirements specification, the camera must be able to detect objects within 1.0 m. To this end, the *input value* for this test case is a measurement of a camera's sensing distance. As the resulting value must be 1.0 m for this test case to pass, the *expected output value* is defined to also be 1.0 m. Lastly, to successfully execute this test case, a measurable object must be within sensing range of the SVS camera sensor, at most, 1.0 m.

Table 5.1: Example of test case as defined according to IEEE standard [54].

Parameter	**Value**
Test Case	*Camera has an effective sensing distance of* 1.0 m
Input Value	*Measured sensing distance*
Expected Output Value	1.0 m
Operating Conditions	*A measurable object is placed at* 1.0 m *away from the SVS, and the camera sensor is enabled and able to sense external objects.*

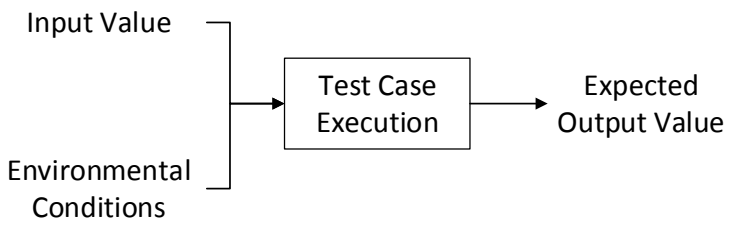
As such, test cases are often defined at design time when environmental conditions are static in nature and do not take into account the impact that a changing operating context can have on a DAS. To facilitate run-time testing, traditional test cases must be

extended to provide adequate flexibility in the input conditions, expected output values, and environmental conditions that define the test case.

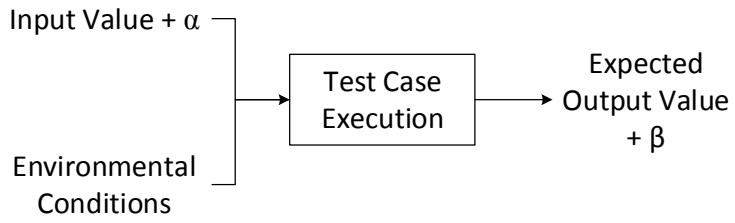
Figure 5.1 presents a block diagram comparing the standard test case definition and our proposed extensions necessary to facilitate run-time DAS testing. Specifically, Figure 5.1(A) demonstrates a standard test case, comprising an input value, corresponding output value, and the environmental conditions necessary to execute the test case. Conversely, Figure 5.1(B) presents an *adaptive test case*. An adaptive test case introduces flexibility in terms of the input value (i.e., α) and expected output value (i.e., β). As environmental conditions change, the input value and expected output value may require an update to properly represent the new operating context. For example, consider the lens of a camera on an SVS. During normal operation, the camera can sense objects within a range of 1.0 m, however during its latest cleaning cycle, the SVS collided with an object that scratched the camera lens, thus reducing its range of visibility to 0.75 m. However, a predefined safety tolerance within the requirements specify that the SVS can safely operate as long as the camera can sense objects within a radius of 0.5 m, thereby allowing continuing operation of the SVS with its currently reduced capability. An adaptive test case that tests the sensing range of the SVS could therefore update its expected sensing range (i.e., expected output value) to be 0.75 m as opposed to its initial definition of 1.0 m. Given that the SVS is still satisfying its requirements, this test case should continue to pass. The impact of making such an adaptation is minor, as the safety tolerance ensures that the SVS still continues to satisfy its safety requirements. However, the SVS can experience requirements violations as its utility functions may be rigidly defined in expecting the initial sensing range. These violations may be reduced by applying RELAX as specified in Chapter 3.

5.1.2 When to Test

Testing non-adaptive systems occurs throughout the software development cycle, including the design, requirements, integration, and maintenance phases. However, testing efforts



(A) Standard test case.



(B) Adaptive test case.

Figure 5.1: Comparison between standard test case and adaptive test case.

may decrease once the software has been released and may be restricted to performing regression testing on bug fixes or code updates to ensure that new or modified functionality does not have an adverse effect on the system. Moreover, post-deployment testing tends to occur on isolated or cloned systems to ensure that there is not a negative impact on the production environment. Conversely, testing a DAS following deployment must be conducted at run time on a live system and often in response to changes in the operating context. Given the need to perform run-time testing on DASs, traditional testing techniques must be augmented to consider an appropriate time to execute the test cases to prevent adverse performance effects on the system under test. Furthermore, assurance that run-time testing activities do not negatively impact DAS performance must also be provided.

For example, continually testing an SVS can have an impact on its ability to perform its intended functionality and process environmental data gathered from its sensors. Specifically, images gathered by the camera may require intensive processing to distinguish and categorize objects within its field of view. As such, executing a set of test cases concurrently with the image processing can overextend the on-board processor, thereby reducing performance for

both operations. Determining an appropriate schedule for testing is therefore a necessity for run-time testing to be feasible.

5.1.3 Testing Methodology Selection

Many established methods exist for performing software testing at design time, with each verifying and validating a system in different ways. These include black-box, white-box, equivalence class, and boundary testing. Black-box testing [82] examines system functionality without having an intimate knowledge of the internal workings of the system, whereas white-box testing [82] examines functionality based upon prior knowledge of the system. Equivalence class testing [14] groups test cases that exhibit similar system responses and then executes a single item from each group to minimize the length of the testing procedure. Finally, boundary testing [14] defines test case input values at and beyond specified limits to determine the system's response.

Each of these testing strategies examines different aspects of system behavior and performance, and moreover can be extended to provide run-time testing capabilities. However, assurance must be provided that run-time testing will not negatively impact the expected functionality and behavior of the system under test. To this end, monitoring run-time requirements satisfaction and only performing testing during periods in which the DAS is not actively satisfying safety or failsafe requirements can assist in this objective.

5.1.4 Impact and Mitigation of Test Results

A failed test case in traditional systems often demonstrates a requirements violation, typically necessitating an update to the software system and further testing. Common repair strategies require a detailed investigation into the cause of the failure and an accompanying resolution that may include code refactoring, patch generation, or bug fixing. Following the repair attempt, the system must then be re-validated to ensure that the resolution strategy fixed the problem and did not introduce any new problems.

Conversely, a DAS can leverage monitoring and testing information to determine if a self-reconfiguration is needed to prevent a requirements violation from occurring. Specifically, the DAS can correlate information gained from monitoring itself and its environment with the results of run-time testing to determine an appropriate mitigation strategy. For instance, the DAS can select a new configuration or require that a live patch be applied to introduce a new configuration to mitigate an unexpected environmental situation. An example of this behavior is that an SVS may determine that a failing test case is due to a fault within an internal sensor tasked with monitoring sensor health. As a result, self-reconfiguration may not be possible, however a software patch can be applied to the internal sensor to resolve the problem.

5.2 Introduction to the MAPE-T Feedback Loop

This chapter introduces MAPE-T, a run-time testing feedback loop that proposes a feedback-based approach for implementing run-time testing in a DAS. MAPE-T is inspired from the MAPE-K autonomic feedback architecture that often guides DAS development [61]. Specifically, MAPE-T provides components for *Monitoring*, *Analyzing*, *Planning*, and *Executing* test cases at run time, with each component linked by *Testing knowledge*.

Thesis statement. *Automated run-time assurance can be facilitated by a testing feedback loop that monitors, analyzes, plans, and executes test cases during execution.*

To facilitate run-time assurance, the MAPE-T loop has four main components. *Monitoring* observes and identifies changes within the DAS and in its environmental context. This information is then transferred to the *Analyzing* component that in turn identifies individual test cases that require adaptation and creates a test plan specific to the current operating conditions. Next, the *Planning* component performs adaptation on test case input values and expected output values and then schedules the execution of the test plan. Then the *Executing* component performs the test plan and analyzes its results. Test case results can

feed back into the DAS MAPE-K loop to be used in determining if the DAS requires reconfiguration. Lastly, *Testing Knowledge* ensures that each element has full awareness of all testing activities.

5.2.1 MAPE-T Feedback Loop

We now describe each component of the MAPE-T feedback loop in detail, identify challenges to realizing each component, discuss candidate enabling technologies that could be leveraged, and present an example of each component within the context of the SVS case study. Figure 5.2 provides a graphical representation of the MAPE-T feedback loop, comprising the *Monitoring*, *Analyzing*, *Planning*, and *Executing* components, each of which are linked together by *Testing Knowledge*.

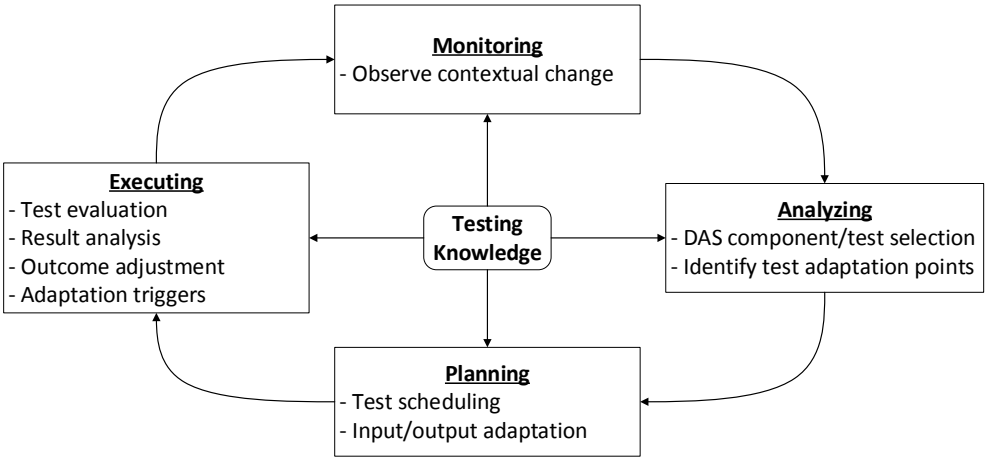


Figure 5.2: MAPE-T feedback loop.

5.2.2 Monitoring

The MAPE-T monitoring component provides run-time information regarding both the DAS and its environment. Moreover, monitoring is considered both an internal and an external process, as monitoring introspects upon DAS components to provide data on sys-

tem execution, and moreover measures properties regarding its execution environment to determine external factors impacting the DAS.

5.2.2.1 Key Challenges

The difficulties in monitoring the operating context of the system executing MAPE-T are similar to those that are encountered within the monitoring component of the MAPE-K feedback loop [61]. The following monitoring challenges are applicable to both feedback loops:

- Which properties of the DAS and its operating context must be observed.
- Which sensors or aggregations of sensor values can measure observed properties.
- What frequency for sampling sensors or sensor values is appropriate.
- How does uncertainty in the system (i.e., imperfect or unreliable sensors) or environment (i.e., unexpected operating conditions) affect sampled sensor data.

The MAPE-T monitoring component must address each of these concerns to ensure that all processes within the MAPE-T feedback loop are provided with accurate and appropriate information. For example, an SVS is required to sample an array of sensors to execute a path planning algorithm appropriate to its environment. As such, an appropriate sampling frequency must be determined for each sensor, and the sampled values must be verified to ensure that faulty data as a result of system and/or environmental uncertainties are not introduced into the loop.

5.2.2.2 Enabling Technologies

As MAPE-T is based upon MAPE-K, we envision that the monitoring component will reuse the infrastructure already in place for monitoring a DAS in the MAPE-K architecture. To this end, MAPE-T and MAPE-K will depend upon the same monitoring infrastructure

that provides information regarding DAS components and processes as well as monitoring sensors that provide feedback on environmental conditions.

Additionally, the MAPE-T architecture must also observe the success and failure rates of test cases at run time. Moreover, the relevance of individual test cases to different operating conditions must also be observed to ensure that only necessary test cases are executed. As a result, enabling technologies must support the traditional MAPE-K monitoring requirements as well as the following:

- Test case instrumentation such that the DAS is continuously aware of each individual outcome.
- Correlation between test cases and particular operating contexts to determine test case relevance.
- Traceability between requirements and test cases to ensure that changing requirements also facilitate updates to test cases.

5.2.3 Motivating Example

We now provide a motivating example of the monitoring component of the MAPE-T feedback loop. For this example and all following motivating examples, the SVS is performing a mode change in which it previously executed the *RANDOM* path plan for 10 seconds and must now transition to a 20 second *SPIRAL* path plan.

Given that the SVS has been modeled as a DAS, a *monitoring infrastructure* already exists for monitoring system and environmental conditions. This feature of the DAS MAPE-K loop can be reused and extended to provide the necessary infrastructure required by MAPE-T. Upon notification from internal monitoring that the SVS has satisfied a 10 second path execution timer, the SVS begins its transition to a new path plan (i.e., the *SPIRAL* path plan). Before doing so, a run-time testing cycle must be performed due to the change in operating context.

5.2.4 Analyzing

The analyzing component focuses on generating an appropriate software test plan comprised of test cases that can be adapted at run time. As the DAS self-reconfigures or environmental conditions change, the test plan can contain test cases that are irrelevant to the current operating context, and therefore cannot be properly executed. By analyzing current conditions, MAPE-T can determine which test cases are relevant (i.e., valid for execution) and thereby generate an appropriate software test plan conducive to the operating context.

5.2.4.1 Key Challenges

Three key challenges must be addressed in order to implement this component. First, important changes within the DAS and its environment must be properly identified to determine if a contextual change has occurred. Second, the previously identified changes must then be analyzed to determine their impact on the current test plan, including relevance and precision of test cases and their parameters, respectively. Finally, the analyzing component must continually update traceability links between individual requirements, test cases, and the DAS implementation.

5.2.4.2 Enabling Technologies

The analyzing component will leverage and possibly extend existing techniques from formal methods [36], run-time requirements adaptation [8, 85, 94], and non-functional requirements satisfaction [39, 108]. For example, an approach for verifying component-based systems by instrumentation via a set of Labeled Transition System (LTS) properties has been proposed [36], where LTS is a formal method for specifying a set of states and transitions. Each transition is then labeled by a particular action. By formalizing a DAS's state space, it can be possible to monitor relevant artifacts within a system in an efficient manner.

Requirements are typically elicited at design time and are developed towards a set of target operating contexts that may become violated over time as the system or environment changes. However, requirements can be treated as objects to be satisfied at run time to counter environmental uncertainty that affects a DAS [94]. An approach to elevate requirements to run-time entities leverages goal and architecture-based reflection [8]. If requirements are considered as first-class artifacts in a system, then run-time reasoning [85] can be applied to consider trade-offs that may occur if the satisfaction of one requirement impedes the satisfaction of another. Similarly, we propose that test cases should also be converted to first-class entities in a system to enable run-time reasoning within testing activities.

Lastly, non-functional requirements must also be considered when designing a requirements-aware DAS [94], given that non-functional requirements impose important constraints on *how* a system delivers acceptable behavior. These constraints typically involve performance, efficiency, or usability concerns. Filieri [39] and Villegas [108] have both proposed methods for quantifying non-functional requirements. This quantification can then enable non-functional requirements to also be treated as first-class entities, enabling system functionality to evolve alongside system behavior during execution.

5.2.4.3 Motivating Example

We now provide a motivating example of the analyzing component of the MAPE-T feedback loop, following the example presented in Chapter 5.2.3.

Upon receiving control from the monitoring component, the analyzing component must determine which test cases are relevant to the current operating context. As the most recently completed path plan was *RANDOM*, test cases that validate the length of time for which it was executed can be selected, as well as any other test cases that can relate to the *RANDOM* path plan. This adaptation can be represented in LTS as a transition between DAS modes, where test cases associated with each state can be selected for execution. Moreover, all safety and failsafe test cases are also selected, as they must continually be re-validated.

5.2.5 Planning

The planning component accepts the set of test cases identified by the analyzing component and determines an appropriate schedule for executing the test plan. Specifically, planning must identify a period of time that is conducive to testing, and moreover ensure that DAS performance and behavior is not adversely impacted by the execution of the test plan. Furthermore, planning is also concerned with facilitating the adaptation of test case inputs and expected outputs, as adaptation points can be defined in the planning component to be used as a basis for later adaptation.

5.2.5.1 Key Challenges

We foresee four key challenges inherent in planning run-time test execution. First, the performance cost of test case adaptation and side effects must be rigorously analyzed, as the overall impact of running a test plan must be known. Specifically, executing and adapting test cases must not cause interference or delays in the overall satisfaction of DAS requirements. Second, automatically detecting low periods of activity within the DAS can provide information regarding proper timing for test plan execution. Third, defining the conditions that trigger input and output value adaptation must also be considered as a key challenge. Finally, extending standard testing approaches to consume run-time monitoring information for adaptation purposes is another challenge to be overcome.

5.2.5.2 Enabling Technologies

The planning component can benefit from techniques for continuous testing, search-based software engineering, using models as run-time entities, and test scheduling and planning. Continuous testing is an approach for performing testing activities on a system at run time. The setup and maintenance of test case parameters, test scheduling, and selection of test cases to be executed are concerns that must be addressed while performing continuous

testing. Along these lines, Saff and Ernst have studied the effects of continuous testing by performing online regression testing to aid software developers as they write code [91, 92]. In this case, spare processor cycles were used to execute regression test cases to verify that the code updates made by the software developers continually satisfied requirements while the system was executing. Similarly, Nguyen *et al.* [78] implemented run-time testing via distributed agents that execute tests at run time. Specifically, a dedicated, automated testing agent coordinates the testing of other agents within the distributed system. In this case, testing is performed by two types of agents: monitoring and testing. The monitoring agent detects and reports faults found within the system. The testing agent generates test suites to be executed by other system agents to provide continuous testing throughout execution.

Test cases derived for a system generally provide a suitable starting point for continuous testing, however the test cases are often static in nature and therefore can lose relevance as the system experiences unexpected operating conditions. As such, test cases must adapt along with the DAS. To address this concern, the planning component can supplement the initial suite of test case parameters with run-time monitored values [67] to accurately test the conditions that the DAS experiences throughout its lifetime. To this end, search-based techniques, such as online evolutionary computation [43], can be leveraged to explore optimal test suite parameter configurations. Furthermore, search-based techniques can also facilitate adaptive test case generation. In particular, machine learning techniques that apply probabilistic methods [4, 15, 38, 84], exploring points of failure [86], or extending current methods with evolutionary computation [69] can be considered for assisting in planning run-time testing.

Leveraging models at run time provides another dimension of enabling run-time requirements satisfaction. Baresi *et al.* [4] have proposed methods for validating software compositions at run time with an assertion language that describes both functional and non-functional system properties. Morin *et al.* [76] have proposed an approach that considers a DAS to be composed of software product lines (SPLs), where adaptations are represented as

different configurations of SPLs. To this end, the goal of SPL design is to share a common codebase between different projects to enable code reuse. Trade-offs in balancing object satisfaction must also be considered at run-time by reasoning over the models as they experience both system and environmental uncertainty [33].

Scheduling test case execution is another concern for the planning component of MAPE-T. Test case scheduling requires a careful balance between two competing concerns: maximizing the utilization of system resources while minimizing adverse side effects experienced by the DAS. As was previously described in the continuous testing approaches [91, 92], the planning component of the run-time testing framework can execute test cases as spare processor cycles become available. Likewise, the DAS can leverage selective testing strategies to filter extraneous tests that would otherwise consume needed resources. This concern is exacerbated within onboard embedded systems, where resources tend to be constrained to limited processing or memory capabilities.

Lastly, selecting an appropriate subset of test cases to be executed can be considered a multidimensional optimization problem, where competing concerns can be represented by system performance, test case execution scheduling, and adaptation penalties that may occur. Furthermore, there is still a large challenge in quantifying each of these concerns, as many comprise non-functional characteristics. As such, it may be possible to use multidimensional optimization algorithms, such as NSGA-II [27], to balance these competing concerns when selecting test cases. Moreover, extending NSGA-II to be performed either online or in a separate, evolutionary agent can further enable run-time test case selection.

5.2.5.3 Motivating Example

We now provide a motivating example of the planning component of the MAPE-T feedback loop, following the example presented in Chapter 5.2.3.

After the analyzing component has completed its tasks, the planning component of MAPE-T analyzes the current state of the SVS to determine if run-time testing can be phys-

ically performed, considering performance and memory restrictions that may be imposed by the limited processing capabilities of the on-board controller. Once the planning component determines that testing can be safely performed, a run-time test plan is generated comprising all test cases relevant to the current operating context.

5.2.6 Executing

The executing component performs the test plan according to the previously determined schedule, analyzes the results of each test case, and triggers any necessary adaptations in the testing framework or DAS. Following execution of the test plan, each individual test result is analyzed to determine if adaptation is required. A test case adaptation is necessary if the test case is no longer relevant to its environment (i.e., the input value and expected output value are no longer correct for current operating conditions). Note that we are not suggesting that test cases be adapted to ensure that they will always provide a passing value, instead, they are adapted to ensure that they remain applicable targets for execution. Finally, the test results may also be delivered to the DAS MAPE-K loop to provide extra information regarding DAS behavior and performance, the result of which may result in reconfiguration of the DAS.

5.2.6.1 Key Challenges

Three key challenges exist for the executing component. The first is promoting both requirements and test cases to be first-class entities in the DAS. Specifically, distilling requirements, especially non-functional requirements, and test cases into quantifiable entities is a non-trivial task. Moreover, providing adaptation capabilities for test cases is also non-trivial, as any flexibility must be guaranteed to not exceed any safety or failsafe tolerance. The second challenge is in determining an appropriate amount of adaptability for each test case, as excess plasticity can cause adverse effects on the DAS. The third challenge is in de-

termining and adapting the acceptance rate for test cases, as not all test cases can necessarily be executed at the same time.

5.2.6.2 Enabling Technologies

A DAS must be able to continually deliver acceptable behavior and satisfy requirements even as its operating context changes or requirements are updated or added. To this end, Souza *et al.* [99] have proposed eliciting *awareness requirements* that monitor requirements violations over time and are treated as run-time entities. Once a predetermined amount of violations occurs, the failing requirement can be adapted at run-time with previously identified parameter changes to improve its chances of future satisfaction. This approach can be modeled as a set of operations over a goal model, further enabling the use of models at run time. Other approaches also use requirements as run-time entities [8, 94] that provide capabilities for reasoning about trade-offs in requirements satisfaction at run time. These *requirements-aware* systems provide a step towards handling uncertainty in a systematic manner, using system requirements as a basis for quantifying run-time behavior. Additionally, including specification languages such as RELAX [21] or FLAGS [5] can reduce the rigidity of requirements, thereby improving the chances of run-time requirements satisfaction.

Run-time testing has yet to gain major traction in production software systems, however it has been successfully applied within hardware-based adaptive systems [75]. To this end, field-programmable gate arrays are used to provide adaptive capabilities. Run-time testing methods in hardware systems are typically concerned with component and/or system execution failures, and therefore must reconfigure the system as needed [29, 95]. Additionally, these systems are further constrained with embedded CPUs and must consider power optimization and minimization of memory usage in their design. A previous approach used monitoring, redundancy, and a well-defined recovery procedure to facilitate run-time testing of an adaptive hardware system [75]. This approach enables a system to continuously func-

tion while executing under adverse conditions without the need for external intervention, however, intervention can be required in the presence of severe or recurring faults. As such, hardware-domain techniques can be reused in software, such as using redundant controllers for monitoring and correcting faults at run time [78].

5.2.6.3 Motivating Example

We now provide a motivating example of the executing component of the MAPE-T feedback loop, following the example presented in Chapter 5.2.3.

Upon scheduling the test plan, the execution component runs the previously selected test cases and analyzes their results. Test cases results are correlated with the controlling SVS goal model (see Figure 2.2) to determine the validity of their results. Failed test cases are analyzed to determine if a run-time adaptation is necessary, as it is possible that the current operating context may no longer reflect the initial intent of test cases derived at design time [43]. Control is then passed back to the monitoring component, and the MAPE-T loop repeats.

5.3 Related Work

While testing is widely used throughout the software industry for the purposes of verification and validation, extending testing activities to the run-time domain has only recently been introduced as a viable means of providing assurance. Testing initiatives are often relegated to design-time assurance techniques [9, 52], with common approaches [14, 82] being structural, functional, and regression testing. However, test cases derived at design time are generally static in nature and cannot adequately provide assurance for a system as it experiences unexpected or unanticipated environmental conditions. As software grows in complexity, the need for run-time assurance becomes apparent, as software is required more and more to remain online while continually behaving as expected.

This section presents related work in providing run-time testing for complex, distributed systems. In particular, we overview how run-time monitoring of system behavior can be used to test systems at run time, as well as how multi-agent systems, a branch of distributed computing, can also be used as a basis for run-time testing.

5.3.1 Exploration of System Behavior

Distributed systems tend to comprise large networks of heterogeneous nodes that must communicate with each other to attain a common goal. As such, validating the behavior of each node and the system as a whole is a non-trivial task, given the size and complexity of the system, as well as the fact that source code and/or private data may not be available on nodes within the system. To this end, monitoring and exploring system behavior provides a black-box approach to testing a distributed system at run time.

Researchers have explored how system behavior can be analyzed to determine if potential faults exist within distributed systems [16, 17] and have introduced DiCE as a framework for providing run-time testing. DiCE is an approach that automatically explores how a distributed system behaves at run time and also checks to determine if there are any deviations from desired behavior. To accomplish this task, system nodes are subjected to a variety of inputs that systematically exercise their behaviors and actions. Furthermore, DiCE operates in parallel to a deployed system so as not to negatively or adversely impact the production system.

To systematically explore node behavior, DiCE begins execution from the current state of the distributed system, essentially creating a system checkpoint for its beginning state. DiCE clones this checkpoint to be used as a basis for testing, and then employs *concolic execution* to explore how different inputs affect the node behavior. Concolic execution, a combination of concrete and symbolic execution, uses concrete inputs while symbolically executing a program. In concolic execution, constraints are collected by following code paths and are then fed into a solver that attempts to find an input that negates each constraint.

Negating constraints provides DiCE with a set of concrete inputs that, in aggregate, provide full coverage of the conditions that the distributed system faces at run time.

DiCE was applied to a case study that explored how the border gateway protocol (BGP), the basis for the Internet's inter-domain routing, can experience routing leaks (i.e., hijacking of domain prefixes, such as a widely publicized issue in which traffic routed to YouTube.com was hijacked [11]). DiCE exercised each node's message processing code, resulting in a set of prefix ranges that can be leaked, and therefore filtered by Internet providers, to prevent such an attack.

While DiCE exhaustively explores different paths of execution in parallel to live systems, our run-time approaches to providing assurance focus on exploring the reactions of live systems in a production environment. This method provides the benefit that faults can be identified live within the production system, rather than relying on a secondary, parallel system. However, to enable this method, we stress that caution must be exercised when implementing run-time assurance methods to ensure that, by providing assurance, we do not adversely impact the normal execution behaviors of the running system.

5.3.2 Multi-Agent Systems

Multi-agent systems (MASs) are typically implemented as distributed systems that contain multiple agents that must interact to achieve key objectives for both the agents and the system as a whole. Agents, in reality, are separate software systems that must communicate with each other by passing messages. Furthermore, agents can be located within different host systems, where each host represents a different environment. Agents are *reactive* in that they can monitor and react to changes in operational context, and moreover are considered *proactive* in that they are autonomous and can decide which actions must be performed to achieve their goals. Due to these particular attributes of a MAS, testing is a non-trivial task requiring careful thought and extension of existing techniques.

Run-time testing of MASs requires that a framework be implemented that supports the autonomous, proactive, and reactive properties of agents within a system. In particular, Nguyen *et al.* [78] have proposed an approach that uses a set of manually derived test cases complemented with automatically generated test cases. These test cases are then executed continuously at run time by dedicated agents tasked with monitoring and performing testing activities. This framework, called *eCAT*, uses two types of agents to enable testing: an autonomous tester agent and a monitoring agent. Each of these agents is next described in detail.

5.3.2.1 Autonomous Tester Agent

The autonomous tester agent is tasked with automatically generating and executing test cases. In particular, this agent can use either random testing or evolutionary mutation (hereafter "evol-mutation") testing to generate test cases. Tester agents use methods for random test data generation [73, 101] to randomly populate test cases. These test cases, in turn, are then sent to the software systems (i.e., other agents in the MAS) that are under test to be executed. It is then the task of the monitoring agent to observe the results of test case execution for each system under test. Random testing may be a cheap and efficient technique for generating test data, however the resulting test cases often lack meaning and, as such, may not be able to uncover all faults within an agent.

Evol-mutation testing combines mutation testing with evolutionary testing. In particular, mutation testing [30] mutates existing programs (i.e., agents under test) in an attempt to artificially introduce known defects. Moreover, mutation testing has been shown to be a valid replacement for fault injection, as well as a better indicator of fault detection than code coverage [58]. For example, a mutant can be a copy of an existing agent, however an `if-statement` within the agent may be slightly modified with a different value being checked, the intent of which is a fault. Evolutionary testing [83] uses evolutionary methods to evolve test case parameters in order to achieve a better overall test suite using derived fit-

ness metrics. Combining these two approaches yields evol-mutation testing. Evol-mutation testing provides an advantage in that test cases can be evolved to catch the faults introduced into mutants, rather than relying on static testing methods. The steps required to perform evol-mutation are next presented:

1. *Preparation:* Initially, mutation testing is executed on the agent(s) under test to create a set of mutants. These mutant programs are provided as input to evolutionary testing.

2. *Test execution and adequacy measurement:* The test cases provided at design time are then executed on all mutants. A fitness value is then calculated for each test case based on a ratio of how many mutants were killed by each test case, where a killed mutant is an agent whose output deviates from what is expected, signifying a discovered fault.

3. *Test case evolution:* An evolutionary process is then applied to generate new test cases by applying crossover and mutation operations to existing test cases.

4. *Decision:* Lastly, this step determines if the evolutionary process should end (i.e., maximum number of generations has been attained) or if a new set of mutant agents should be created and tested, respectively.

5.3.2.2 Monitoring Agent

The monitoring agent is tasked with observing the results of test cases applied to agents under test. In particular, if monitored output deviates from an expected value, then the monitoring agent reports to a controlling entity that the particular agent exhibits a fault. If this agent is a mutant, then it is considered to have been killed. Furthermore, monitoring agents are also tasked with verifying that pre and post-conditions are met by agents under test to protect the MAS during execution. Lastly, monitoring agents observe agent behavior and provide an execution trace to the autonomous tester agents, including information regarding agent birth (i.e., new mutant created), agent death (i.e., mutant killed), and agent interactions.

eCat has been successfully applied to a case study in which a MAS, *BibFinder*, is tasked with finding and retrieving bibliographic information from a computer. Compared to random testing and goal-based testing, the continual testing approach of eCat was able to find more bugs in the system, particularly those that are either very hard or take a large amount of time to manifest.

Continuous testing of MASs is similar to the MAPE-T feedback loop that has been proposed within this dissertation. However, our approach differs in that MAPE-T defines a feedback-based approach for continuously executing and adapting testing activities. Moreover, MAPE-T is intended to test a production system that does not contain artificial defects or generated mutants, thereby providing assurance for the live system. Finally, the information provided by run-time testing can be used to actively reconfigure the DAS, thereby hardening it against uncertainty.

5.4 Discussion

This chapter has introduced the MAPE-T feedback loop, an approach for automatically providing run-time assurance for a DAS. MAPE-T comprises a set of components to facilitate run-time testing. A monitoring component measures system and environmental conditions, an analyzing component generates a test plan specific to current operating conditions, a planning component performs run-time test case adaptation and moreover schedules the execution of the test plan, and an execution component runs the generated test plan and analyzes its results. Each of these components are linked together by testing knowledge, a controlling aspect of MAPE-T that enables the sharing of test-related information within each component. Furthermore, this chapter described key challenges to overcome in implementing each component within the MAPE-T loop, and also proposed methods for leveraging existing design-time techniques within the run-time domain. The following chapters detail investigations into realizing the components of MAPE-T on a simulated SVS.

Chapter 6

Run-Time Test Adaptation

This chapter introduces our techniques for performing test adaptation at run time for a DAS. Specifically, we explore how both test suites and test cases require adaptation during DAS execution to remain relevant even as system parameters and environmental conditions change. We first motivate the need to perform both types of run-time test adaptation in the face of uncertainty. Next, we describe our techniques for supporting run-time test suites adaptation and test case adaptation, respectively. Proteus[1] is a technique for adapting test suites at run time to ensure that the appropriate test cases are selected for online execution. Veritas[2] is a technique for performing fine-grained adaptation of test case parameter values to ensure that individual test cases remain relevant to their current operating context. In addition, Proteus acts as a managing infrastructure to facilitate run-time test adaptation by performing coarse-grained test suite adaptation and invoking Veritas as necessary for finer-grained parameter value adaptations. Next, we present our approach for using both Proteus and Veritas at run time to enable adaptive testing. Then, we present results from two separate experiments. The first experiment details how Proteus was used to automatically adapt suites at run time, and the second experiment details how Veritas was used to automatically adapt test cases at run time. Next we explore whether RELAX operators can be applied to run-time

[1] In Greek mythology, Proteus was a deity that could change his form at will.

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

test cases to further enhance adaptive testing. Then, we compare and contrast both Proteus and Veritas with other testing initiatives. Finally, we summarize our findings and present future directions for our research into run-time test adaptation.

6.1 Motivation

DAS applications often must continually deliver critical functionality with assurance even as the environment changes, requiring the DAS to self-adapt to mitigate unexpected uncertainties. Moreover, the resulting state of the DAS following a reconfiguration may no longer satisfy its requirements. Performing run-time testing, as described in the previous chapter, adds a layer of assurance that the key objectives and requirements of the DAS are still being satisfied even as the system and environment changes. However, the test specification derived for the DAS at design time may lose relevance as the operating context changes. For instance, a test case that tests the sensing range of an on-board camera may become compromised as the SVS collides with a wall, causing the lens to be scratched and thereby effectively reducing its sensing range. As a result, the test case may require coarse-grained adaptation by Proteus (e.g., disabled as the camera is no longer functional) or fine-grained adaptation by Veritas (e.g., the expected sensing range is reduced, yet still valid, and can therefore be adapted to the new sensing range).

6.2 Terminology

This section introduces and defines key terminologies relevant to testing that are used throughout this chapter. These terms are defined as follows:

- *Test case*: Single test to assess all or a portion of a requirement. An IEEE standard test case comprises an input value, expected output value, and conditions necessary to perform the test case [54].

- *Test plan*: Describes the scope and schedule of testing activities.
- *Test suite*: Subset of test cases from the test specification. A test suite is typically derived to be executed under a particular operating context.
- *Test specification*: Set of all possible test cases derived for a system.

6.3 Introduction to Proteus

Proteus is a requirements-driven approach for run-time adaptation of test suites. Specifically, Proteus acts as a managing infrastructure for test adaptation activities performed at run time. To this end, Proteus provides two levels of adaptation capabilities. First, Proteus creates an *adaptive test plan* for each DAS configuration, as well as a default test suite for each adaptive test plan. An adaptive test plan describes how a set of test suites are used to test a particular operating context, where a test suite comprises a collection of test cases to be executed. From each default test suite, Proteus can selectively enable or disable test cases as necessary to maintain relevance of each individual test case as well as to optimize run-time testing performance. Second, Proteus determines when fine-grained test case adaptation is required by monitoring testing results. If necessary, Proteus invokes Veritas to search for an optimal configuration of test case parameter values.

Thesis statement. *Performing online adaptation of both test suites and test cases enhances the overall relevance of testing activities for a DAS at run time.*

6.3.1 Proteus Approach

Proteus is an approach for enabling test suite adaptation at run time. First, Proteus defines an *adaptive test plan* for each DAS configuration at design time. Each adaptive test plan contains a default test suite that is executed upon invocation of the DAS configuration at run time. The default test suite, in turn, contains all possible test cases that are relevant

to that particular DAS configuration. Run-time adaptation occurs by selectively activating and deactivating test cases to generate new test suites based upon test results, where test cases considered failing are re-executed and test cases considered passing are not. This behavior enables test cases determined to be irrelevant to be adapted at run time (i.e., by Veritas) and furthermore enables the conservation of system resources by not re-executing test cases that have already passed for the current operational context. For example, the DAS selects a new configuration C_i at run time. Proteus, in turn, activates the associated adaptive test plan ATP_i and then executes the default test suite $TS_{i.0}$. Following execution of $TS_{i.0}$, Proteus generates a new test suite $TS_{i.1}$ that reflects the configuration of passed and failed test cases. The following sections describe how Proteus defines test suites and adaptive test plans.

Figure 6.1 presents a graphical depiction of a DAS that has been instrumented with Proteus. Specifically, this figure presents the logical connections between DAS configurations, operating contexts, adaptive test plans, and test suites. For example, DAS configuration C_1 is triggered by operational context OC_1. When testing begins, Proteus selects the associated adaptive test plan ATP_1. ATP_1 comprises a dynamically-generated collection of test suites $TS_{i.j}$ that each defines a particular configuration of test cases to be executed. Moreover, $TS_{i.0}$ is considered to be the *default test suite* for ATP_1, and as such, is executed initially each time ATP_1 is selected for testing. The test case parameter values for all test cases within ATP_1 are stored in its associated data store $Params_1$. Finally, each test suite $TS_{i.j}$ comprises a collection of test cases that specify an activation state. Specifically, Proteus selectively activates or deactivates test cases based on test results, and therefore the configuration of test case states defines a unique test suite. The specific definition of a test suite with respect to Proteus is provided in Chapter 6.3.2.

Figure 6.1 also depicts how different testing cycles can generate different test suites within the existing collection of test suites for a particular adaptive test plan. A testing cycle is performed following a DAS reconfiguration, and the testing cycle continues until

either all test cases have passed or a new DAS configuration is selected. Specifically, the results from two testing cycles are pictured in Figure 6.1 by different patterns in each $TS_{i.j}$. The first testing cycle is represented by the darker, right-slanted shading, and the second cycle is represented by the lighter, left-slanted shading. $TS_{i.0}$, as the default test suite, was executed at the start of both testing cycles for regression purposes. Following execution of the default test suite, both testing cycles generated $TS_{i.1}$ based on the configuration of test cases. Following execution of $TS_{i.1}$, the first testing cycle then generated $TS_{1.2}$, $TS_{1.3}$, and $TS_{1.4}$, whereas the second testing cycle generated $TS_{1.5}$, $TS_{1.6}$, and $TS_{1.7}$.

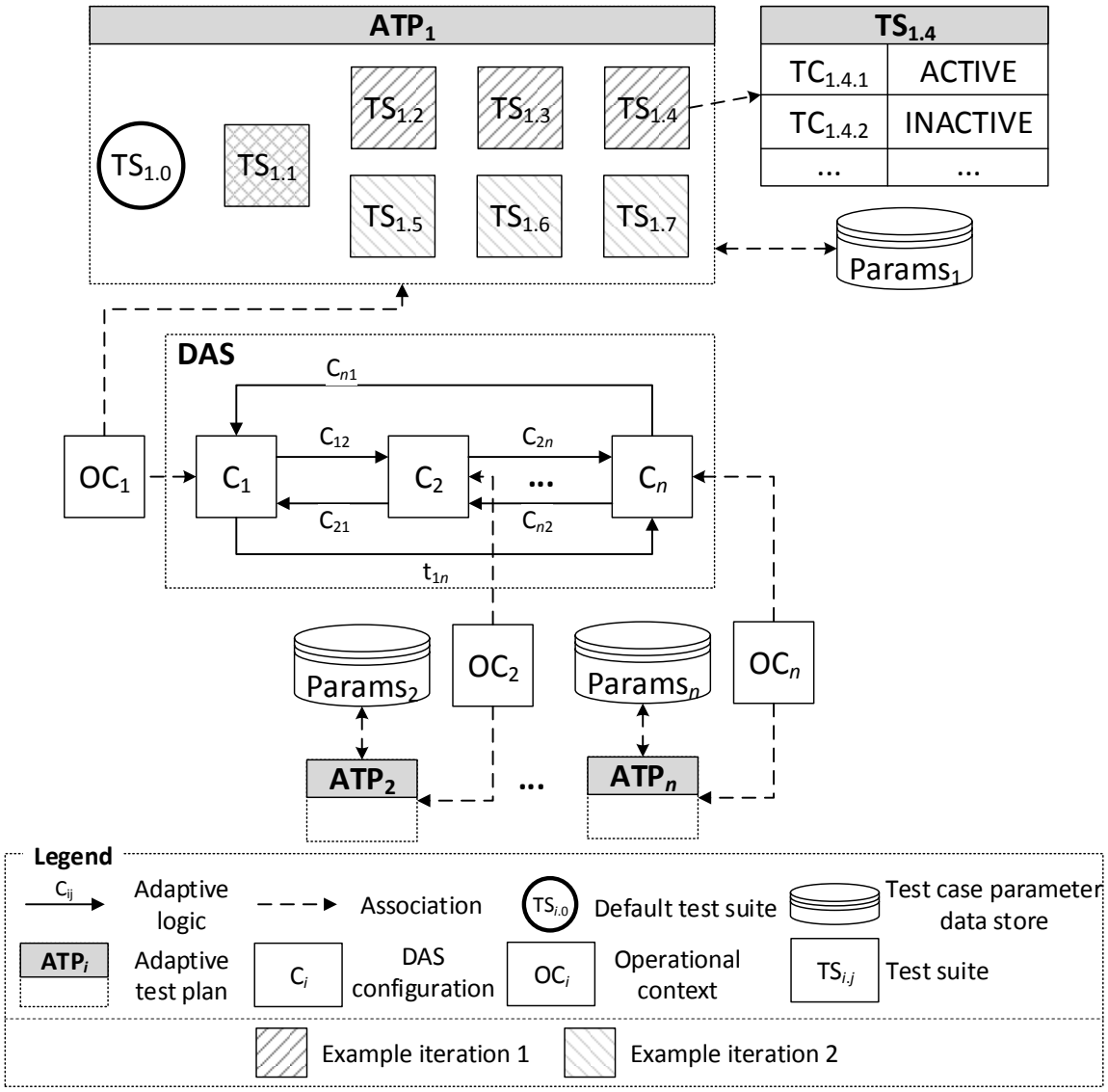


Figure 6.1: DAS configurations and associated adaptive test plans.

6.3.2 Test Suite

Test suites used by Proteus contain a set of test cases to be executed at run time, where each test case has an additional status attribute that can be either ACTIVE, INACTIVE, or N/A. Specifically, ACTIVE test cases are executed when the test suite is executed, INACTIVE test cases are not executed due to prior testing results, and N/A test cases are not executed as they are not relevant to the current operating context. Test case status is determined by two conditions. First, the operational context as specified by the DAS configuration determines which test cases are relevant to the adaptive test plan. Test cases that are not relevant are designated as N/A and cannot be executed while the current adaptive test plan is selected. For instance, the SVS may be following a *SPIRAL* path plan while attempting to navigate around obstacles in a room. In this case, test cases that are related to the *RANDOM* path plan would be designated as N/A, as they are not relevant to the current situation. Collectively, the set of test cases with their associated status form a unique test suite $TS_{i.j}$ within a particular adaptive test plan ATP_i.

The second condition for determining test case status is based on testing results. As run-time performance could be negatively impacted by performing testing activities, Proteus ensures that only *necessary* test cases are executed at run time. As such, only test cases that are designated as ACTIVE are executed. A test case is designated as ACTIVE if it has not yet been executed for a given DAS configuration C_i (assuming it is relevant), or if it has previously failed. Test cases that fail are subjected to further adaptation by Veritas in order to realign the test case with current operating conditions. Test cases that monitor safety or failsafe concerns are always designated as ACTIVE and moreover are precluded from adaptation. This approach enables Proteus to provide continuing assurance that the DAS is satisfying its safety objectives at run time.

Figure 6.2 provides a detailed view of two sample test suites: $TS_{1.0}$ and $TS_{1.1}$. Specifically, the default test suite $TS_{1.0}$ considers all test cases to be relevant except for TC_3,

where TC_3 will be defined as N/A for all other test suites within ATP_1 as it is not relevant to the current operating context. $TS_{1.1}$ defines TC_4 to be INACTIVE, as TC_4 has previously passed for this particular test suite and does not require further execution. As such, $TS_{1.1}$ is generated by Proteus at run time for this particular combination of test case statuses following execution of the default test suite $TS_{1.0}$.

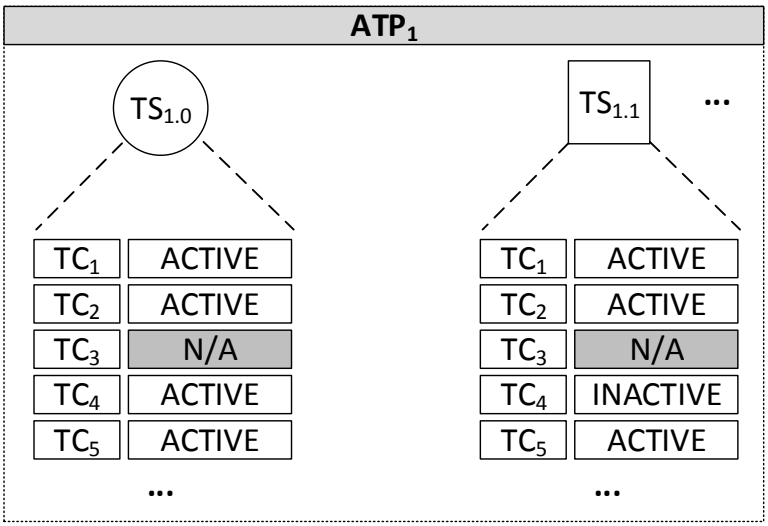


Figure 6.2: Example of test case configuration for $TS_{1.0}$ and $TS_{1.1}$.

6.3.3 Adaptive Test Plan

An adaptive test plan comprises a default test suite and a set of dynamically-generated test suites that are relevant to a particular DAS configuration and operating context. At design time, an adaptive test plan contains only a default test suite $TS_{i.0}$ that specifies each relevant test case to be ACTIVE and each irrelevant test case to be N/A. Following execution of $TS_{i.0}$, Proteus analyzes the test results to generate a new test suite to be executed next. Test cases that are considered to be invalid (e.g., false negatives or false positives) are labeled as ACTIVE and test cases that have passed and are valid are labeled as INACTIVE. Invalid test cases are then provided to Veritas for further adaptation.

Testing within a particular adaptive test plan continues until either all test cases have passed (i.e., are labeled as either INACTIVE or N/A) or the DAS has selected a new con-

figuration. When the DAS selects a new configuration, its associated adaptive test plan is activated and Proteus then executes the default test suite for that particular adaptive test plan.

To illustrate Proteus adaptive test plans, consider the execution of the SVS following startup. At the outset, the SVS randomly selects a *STRAIGHT* path plan to follow while operating under a normal power consumption mode. This configuration of SVS features, defined for a particular set of system and environmental conditions OC_1, is considered to represent DAS configuration C_1. As such, ATP_1 is selected by Proteus for run-time testing and the default test suite $TS_{1.0}$ is executed. Proteus analyzes the results to determine which test cases should remain active. In Figure 6.2, TC_1 and TC_2 monitor safety requirements and are therefore required to remain ACTIVE, regardless of the result of test case execution. TC_3 was not executed, as it is labeled N/A and therefore not relevant to the current operating context (i.e., OC_1). TC_4 passed and was therefore designated as INACTIVE until the operating context changes. TC_5 failed and therefore remains ACTIVE. Proteus analyzes these results and generates a new test suite $TS_{1.1}$ that specifies the current status of each test case. For each test case failure that *disagrees* with its correlated utility function(s) (i.e., test case is determined to be invalid), Proteus invokes Veritas to further adapt the test case parameter values. Following parameter value adaptation, Proteus stores the parameter values for whichever individual test case (i.e., parent or child) exhibited the largest fitness value to $Params_1$ for use in the next testing cycle.

6.3.3.1 Regression Testing

To continually ensure that the DAS is satisfying its requirements even as the environment changes, Proteus performs three different levels of regression testing. First, test cases that monitor safety and failsafe conditions (i.e., Figure 2.2, Goals (D), (J), (R), and (S)) must always remain active to ensure that they are re-validated when each test suite is executed. Second, test cases that monitor system invariants (i.e., Figure 2.2, Goal (G)) are

always considered relevant to all operating contexts and are therefore re-validated within each test suite. Third, test cases that monitor non-invariant goals must be re-validated in the event of a contextual change. Specifically, following reconfiguration of the DAS from C_1 to C_2, the default test suite $TS_{i.0}$ is executed with all relevant test cases set to ACTIVE. Assuming that a particular test case has been previously validated in a prior DAS configuration, re-executing that test case in the current DAS configuration provides functional regression assurance.

6.4 Introduction to Veritas

Veritas is an evolutionary computation-based approach for fine-tuning test case parameter values to ensure that each test case remains relevant to its operational context. Specifically, Veritas uses an online EA (c.f., Chapter 2.5.4) to search for an optimal combination of parameter values for each test case within a pre-defined safety tolerance defined within a test specification. For example, a test case that tests the distance between the SVS and a liquid spill must also provide a safety tolerance. Specifically, the test case may specify that the SVS must detect a spill within 0.5 m, however any value larger than 0.3 m is still considered safe, and any value below 0.3 m causes a safety violation. Therefore, Veritas cannot adapt the expected value of this test case to be any smaller than 0.3 m.

Veritas uses DAS utility functions to assess test case relevance. While utility functions typically quantify high-level system intent (e.g., maintaining a safe distance between vehicles for an intelligent vehicle system), test cases provide a finer-grained assessment for the behavior of individual features and functions. By correlating utility functions with particular test cases, Veritas can determine if test cases are valid or invalid at run time, and if invalid, provide an approach for realigning test case parameters with the operational context.

Thesis statement. *Adapting test case parameter values aids in providing run-time assurance.*

6.4.1 Assumptions, Inputs, and Outputs

In this section, we describe the assumptions that must hold when using Veritas. We then state the required inputs as well as the expected outputs for Veritas when adapting test cases.

6.4.1.1 Assumptions

We assume that a system goal model provides an accurate representation of key objectives and requirements for a system under test. Moreover, the goal model and its derived utility functions are assumed to have been derived correctly to provide a representative quantification of system behavior. Lastly, we assume that a test specification derived from a goal model provides full coverage of all key system objectives and requirements.

6.4.1.2 Inputs and Outputs

Veritas requires five inputs to properly adapt test case parameters at run time: a goal model of the DAS, a set of utility functions to be used for high-level requirements monitoring [109], an executable specification or prototype of the DAS, a set of monitoring elements, and a test specification that comprises all possible test cases. First, Veritas requires a goal model of the DAS under test to capture the requirements and key objectives of the DAS. For this chapter, we use the SVS goal model presented in Figure 2.2.

Second, Veritas needs a set of utility functions for high-level requirements monitoring. The utility functions must quantify each goal's performance during DAS execution. An example utility function that measures the satisfaction of power conservation measures for SVS movement (Figure 2.2, Goal (F)) is presented as follows in Equations (6.1) and (6.2):

$$utility_{Goal\_F} = BatteryDecay, \tag{6.1}$$

where

$$BatteryDecay = \begin{cases} 1.00 & \text{if } BatteryCharge >= 75\%, \\ 0.75 & \text{if } BatteryCharge >= 50\%, \\ 0.50 & \text{if } BatteryCharge >= 25\%, \\ 0.25 & \text{else.} \end{cases} \tag{6.2}$$

The returned utility values are then used to determine if a DAS reconfiguration is necessary. For example, Equations (6.1) and (6.2) quantify the current state of the battery. If the remaining charge falls below a predefined threshold, for instance 50%, then the SVS can self-reconfigure to provide reduced power to its wheel motors, thereby conserving battery power while still achieving motion and satisfying Goals (A) and (B). Veritas, in turn, uses utility values to validate test results. Following the prior example, a test case that monitors the SVS's power mode is considered to be valid if the utility value ($utility_{goal\_F}$) is violated *and* the test case determines that the SVS has not successfully entered a reduced power mode, implying that there is an issue in the DAS reconfiguration mechanism.

Third, an executable specification of the DAS that captures the system and environmental contexts is required. The executable specification is responsible for executing DAS behavior in a simulation environment that can generate and apply all possible operating contexts in which the DAS may operate.

Fourth, a set of monitoring elements is required to monitor and quantify DAS operating conditions, particularly environmental elements (ENV), variables that monitor those ENV elements (MON), and a relationship between each ENV and MON element (REL). For example, while in a power conservation mode, the SVS can reduce power to its wheel motors (ENV). To do so, the SVS queries the wheel motor sensors (MON) to determine how much torque is currently being applied to the wheels. Upon entering the power conservation mode, the

SVS commands the wheel motors to reduce the applied torque (REL) to effectively reduce the amount of battery power consumed, thereby extending the amount of time that the SVS can continue to operate.

Fifth, a test specification that provides full coverage of system requirements must be provided to Veritas. Each test case must also be specified as *invariant* or *non-invariant*. Invariant test cases are precluded from adaptation and are typically associated with safety or failsafe concerns (e.g., SVS shutting down power in case of a safety violation). Non-invariant test cases can be adapted at run time and are typically associated with functional or behavioral concerns. For run-time validation, each test case must also be associated with one or more utility functions that provide oversight on test case validity. Table 6.1 provides a subset of example SVS test cases. For instance, Test Case 3 ensures that the SVS object sensor (MON) can detect large dirt particles in the environment (ENV) within a radius of 0.5 m (REL). As such, the expected value of an SVS internal variable (i.e., `ObjectSensor.DetectRadius`) must equal 0.5 m. If this test case fails, then Veritas checks to determine if an adaptation is necessary, as operating conditions may have changed such that the detection radius of the object sensor is reduced.

Veritas outputs a set of tuples comprising the *environmental configuration*, *system configuration*, and *test specification configuration*, as well as an *adaptation trace*. The tuple is used to determine the state of the DAS at each point of test execution and adaptation and can be used offline to determine the types of system and environmental configurations that were experienced by the DAS during execution. Moreover, the adaptation trace provides information regarding the contextual changes that triggered testing activities, the results of testing, and the adaptation path of each test case. An example of an output tuple is provided in Listing 6.3. This listing provides a snapshot of the data found within each configuration in the tuple `([Environmental Configuration], [System Configuration], [Test Specification], [Adaptation Trace])`.

```
[Environmental Configuration],
  % Environment configuration information
  Number of liquid spills              : 3
  Room width                           : 10m
  Room depth                           : 12m
  Downward step                        : True
  ...
[System Configuration],
  % SVS configuration information
  Starting battery life                : 98.0%
  Starting capacity                    : 0.0%
  Probability of wheel sensor failure  : 5.0%
  ...
[Test Specification],
  ...
 %TC5 - Network actuator has never partitioned network.
  TC5 expected value    : NetworkActuator.NumPartitions = 0
  TC5 safety tolerance  : NetworkActuator.NumPartitions ∈ [0,2]
  TC5 type              : non-invariant
  ...
[Adaptation Trace]
  ...
  % TC5 expected value has been adapted to 1
  Timestep 5: TC5 expected value : 0 -> 1
  ...
```

Figure 6.3: Veritas example output tuple.

Table 6.1: Examples of SVS test cases.

	Test Case 1	*Test Case 2*	*Test Case 3*
Test Case (ENV)	Test suction tank for large objects	If SVS falls off cliff, ensure that failsafe mode was enabled and all power disabled	Verify large dirt particle detection within $0.5m$
Agent (MON)	`InternalSensor (IS)`	`InternalSensor (IS)`, `Controller (C)`	`ObjectSensor (OS)`
Expected Value (REL)	`IS.NoLargeObjects` $== TRUE$	`C.FailSafeActive` $== TRUE$	`OS.DetectRadius` $== 0.5m$
Type	Invariant	Invariant	Non-invariant
Accepted Value	$TRUE$	$TRUE$	$[0.25m,\ 0.75m]$
Goal Constraint	$utility_{Goal\_D}$, $utility_{Goal\_J}$	$utility_{Goal\_D}$, $utility_{Goal\_J}$	$utility_{Goal\_D}$, $utility_{Goal\_R}$

6.4.2 Veritas Fitness Functions

This section describes the fitness functions and validation metrics used by Veritas. While test cases typically return a Boolean *pass* or *fail*, Veritas returns a *measured fitness sub-function value* and *test result*. A fitness value provides an extra metric for comparison of test case relevance, and a particular fitness sub-function is defined for each test case based on its type (i.e., either an exact value or range of values) as well as designation as invariant or non-invariant. Table 6.2 provides the fitness sub-functions for each combination of test type and designation.

The different types of fitness sub-functions are derived based upon the type of variable measured by the particular test case. For those test cases that monitor an *exact*

Table 6.2: Individual test case fitness sub-functions.

Test type	**REL**	**Fitness**

Invariant	Exact Value	if ($value_{measured} == value_{expected}$) then
		$fitness_{measured} = 1.0$
		else
		$fitness_{measured} = 0.0$

Invariant	Range of Values	if ($value_{measured} \in [value_{low\_boundary}, value_{high\_boundary}]$) then
		$fitness_{measured} = 1.0$
		else
		$fitness_{measured} = 0.0$

Non-invariant	Exact Value	$fitness_{measured} = 1.0 - \frac{\lvert value_{measured} - value_{expected} \rvert}{\lvert value_{expected} \rvert}$

Non-invariant	Range of Values	if ($value_{measured} \in [value_{low\_boundary}, value_{high\_boundary}]$) then
		$fitness_{measured} = 1.0$
		else
		$fitness_{measured} = 1.0 - \lvert \frac{value_{measured} - value_{optimal}}{value_{optimal}} \rvert$

value, the test's measured value ($value_{measured}$) is compared to the test's expected value ($value_{expected}$). Variables defined as a *range* expect that the measured value falls within pre-determined boundary values (i.e., $[value_{low\_boundary},\ value_{high\_boundary}]$). Moreover, if a *range* variable falls outside of those boundaries, then the optimal value (i.e., $value_{optimal}$) is defined as the nearest boundary to the measured value, as defined in Equations (6.3) and (6.4) that respectively depict the distance calculation to the nearest boundary and the optimal value:

$$
\begin{aligned}
d_{low} &= |value_{measured} - value_{low\_boundary}| \\
d_{high} &= |value_{high\_boundary} - value_{measured}|,
\end{aligned} \tag{6.3}
$$

where

$$
value_{optimal} = \begin{cases} value_{low\_boundary} & \text{if } (d_{low} < d_{high}), \\ value_{high\_boundary} & \text{else.} \end{cases} \tag{6.4}
$$

As such, overall fitness for each test case considers inclusion within the low and high boundaries as an impetus for providing a fitness boost during the evolutionary process. Particularly, a measured value that is not equal to the expected value, yet is within the specified range of acceptable values, is rewarded for being a valid result. To this end, aggregate fitness for each test case is calculated as a linear weighted sum that comprises the measured fitness sub-function and fitness boost for validity, where weights $\alpha_{measured}$ and α_{valid} define the relative importance of each sub-function. Aggregate test case fitness is next presented in Equations (6.5) and (6.6):

$$
\begin{aligned}
fitness_{test\_case} = \; & \alpha_{measured} * fitness_{measured} + \\
& \alpha_{valid} * ValidResult,
\end{aligned} \tag{6.5}
$$

where

$$
ValidResult = \begin{cases} 1.0 & \text{if } value_{measured} \text{ is valid,} \\ 0.0 & \text{else.} \end{cases} \tag{6.6}
$$

Given that invariant test cases cannot be adapted, they can only pass or fail. Test cases are also considered as passing or failing based upon a *fitness threshold value*. As such, this dissertation considers a test case with a fitness value of 0.75 or higher to have passed and otherwise to have failed. This threshold value was selected to provide a higher probability

that the adaptation process would be triggered as test case fitness must be relatively high to be considered passing. Given that invariant test cases cannot be adapted, they can only pass or fail, and therefore can only use the threshold value to determine the test result. Non-invariant test cases that are considered failed are then adapted by an online evolutionary algorithm (EA).

6.4.2.1 Test Case Validation

Veritas analyzes the results of non-invariant test case execution to determine if adaptation is warranted based on the validity of the test cases. Test case validity is determined by the relationship between the test case and its corresponding utility function, with validity comprising four possible cases:

- **True positive**: Test case fitness is within $[Threshold, 1.0]$ and its correlated utility value is within $(0.0, 1.0]$, indicating that the test is valid and has passed. No extra action is required by the DAS reconfiguration engine or the Proteus test adaptation framework.

- **True negative**: Test case fitness is within $[0.0, Threshold)$ and its correlated utility value equals 0.0, indicating that an error has occurred and the test has failed. Presence of a true negative implies that the DAS requires reconfiguration.

- **False positive**: Test case fitness is within $[Threshold, 1.0]$ and its correlated utility value equals 0.0, requiring both DAS reconfiguration and test adaptation. This result indicates that the test case has passed, however its correlated goal is not satisfied. In this case, the test case and utility value are in *disagreement*, indicating that the test case and/or the utility function are incorrect. A DAS test engineer can analyze the false positive and associated execution trace to determine the source of the error, particularly if the test case or utility function is in error.[3]

[3]For this work, we assume that all utility functions have been derived correctly, and therefore a false positive indicates an error in a test case parameter value.

- **False negative**: Test case fitness is within $[0.0, Threshold)$ and its correlated utility value is within $(0.0, 1.0]$, indicating that the test case requires adaptation. In this case, the test case and utility value are again in *disagreement*. Given that the utility function is satisfied, the test case requires adaptation to become relevant again. Moreover, a DAS test engineer can analyze the state of the DAS and its environment at this point in time to determine the reason(s) that the test case became irrelevant.[3]

For example, a test case that monitors the health of the SVS's cliff sensors is considered to be valid if its corresponding utility value(s) indicate that the SVS has not enabled a failsafe mode (Goal (J)) and has not fallen down a set of stairs (Goal (D)). However, if this test case is passing (i.e., implying that the cliff sensors are fully functional) and the SVS has fallen down a set of stairs (violating Goals (J) and (D)), then the test case *disagrees* with its utility functions and is therefore invalid. In this case, either a DAS reconfiguration or sensor replacement is necessary to rectify the problem. Test cases that are invalid are marked for adaptation.

6.4.2.2 Online Evolutionary Algorithm

Veritas runs the (1+1)-ONLINE EA (see Chapter 2.5.4) on each individual test case that was selected for adaptation, where the (1+1)-ONLINE EA comprises a population of two individuals: a parent and a child, where the child is created by mutating the parent. Mutation of test case parameters is enabled by updating either the test case expected value or range of acceptable values, depending on the type of test case (i.e., exact or range). Following execution and analysis of each individual test case, the individual with the higher fitness value is retained for the next iteration. Veritas considers test cases to be individuals, and evaluation is considered to be a test case execution.

Compared to offline evolutionary approaches, the (1+1)-ONLINE EA gains a significant advantage in its ability to provide evolutionary capabilities at run time by limiting the amount of individuals in the population and only performing mutation. However, an ex-

haustive search of the solution space is precluded due to the limited population size of the (1+1)-ONLINE EA. As a result, the (1+1)-ONLINE EA may not be able to successfully converge to a globally-optimal solution.

6.5 Run-Time Testing Framework

Together, Proteus and Veritas provide a unified approach for DAS run-time test adaptation to mitigate uncertainty at different granularities. Proteus enables coarse-grained test suite adaptation to ensure that only relevant test cases are executed as necessary even as environmental conditions change. Veritas enables fine-grained test case adaptation to update individual test case parameters to ensure that their values continually reflect changing environmental conditions. This section describes how Proteus and Veritas can be used in tandem to ensure that both test suites and test cases can continually provide run-time assurance while testing a DAS. Furthermore, we also describe how Veritas calculates a fitness value for individual test cases, describe how test cases are validated, and lastly describe the process by which Veritas uses the (1+1)-ONLINE EA.

To this end, Figures 6.4 and 6.5 present a combined workflow diagram that illustrates how Proteus and Veritas, respectively, are used to enable run-time test adaptation. Particularly, Figure 6.4 presents the Proteus technique for coarse-grained test suite adaptation, and Figure 6.5 presents the Veritas technique for fine-grained test case adaptation. As is illustrated by the shading in Figure 6.4, (10) and (17), Veritas is a component within the Proteus framework.

We now describe each step of Figures 6.4 and 6.5 in detail, where bolded items indicate Boolean conditions checked before the step is executed.

(1) The DAS identifies a change within its operating context and selects a new DAS configuration. Proteus, in turn, consumes this information and activates the associated adaptive test plan for that particular DAS configuration.

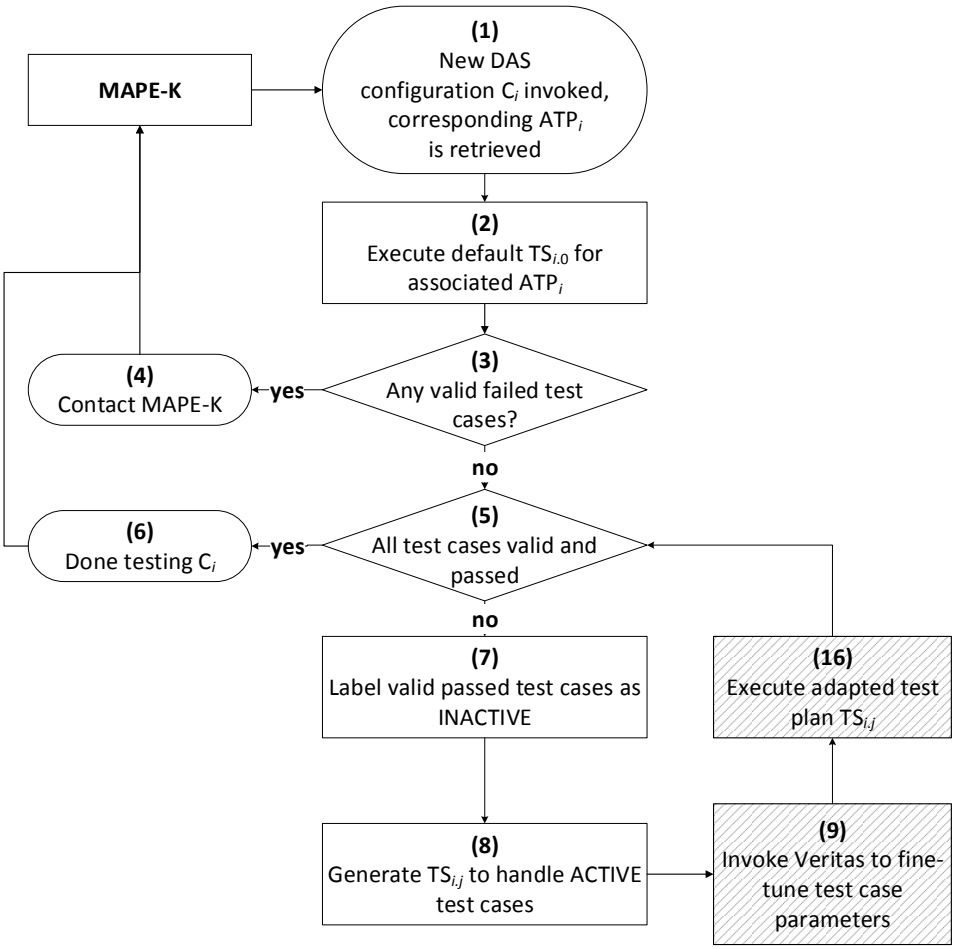


Figure 6.4: Proteus workflow diagram.

(2) Upon activation of run-time testing, Proteus first executes the default test suite, effectively executing all relevant test cases for the particular operational context.

(3) Proteus analyzes the test results to determine if any *true negative* results exist.

(4) **If a *true negative* exists**, then the DAS is notified to perform a self-reconfiguration as an issue has been identified that requires self-reconfiguration.

(5) **If a *true negative* does not exist**, then Proteus checks to see if execution of all test cases resulted in *true positives*.

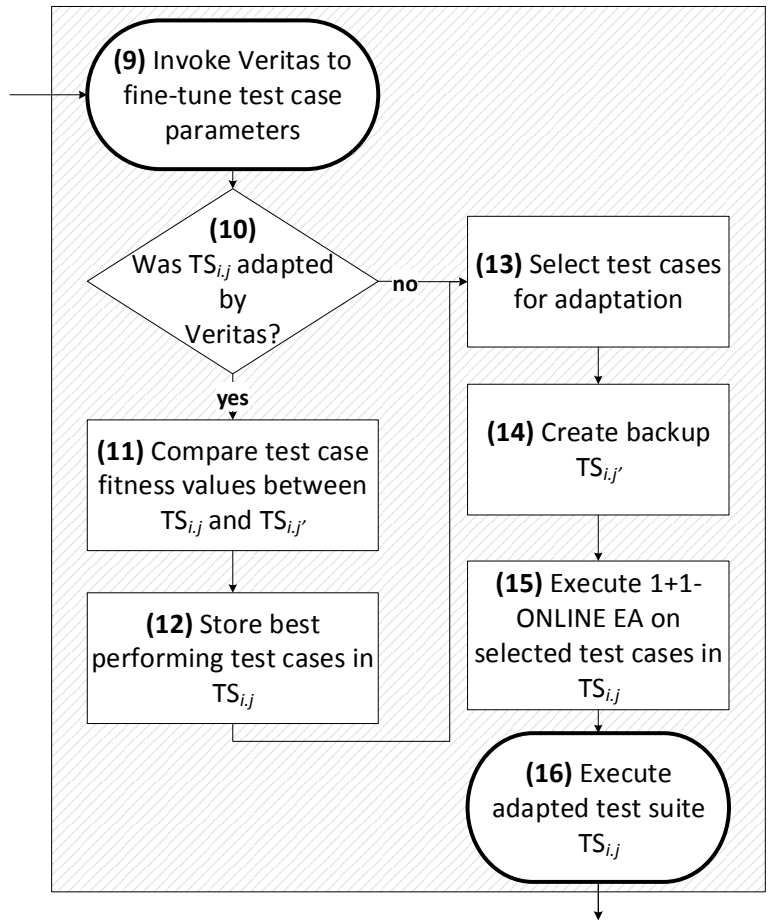


Figure 6.5: Veritas workflow diagram.

(6) **If all test results are *true positives***, then testing is considered complete for the current DAS configuration and testing is halted until a change in operating context occurs.

(7) **If at least one test result is not a *true positive***, then Proteus updates the current test suite. All *true positive* test cases are labeled *INACTIVE*, as they do not need to be re-executed until a change in operating context occurs. All remaining test cases are labeled *ACTIVE*, or valid for execution.

(8) Proteus dynamically-generates a new test suite based upon the default test suite $TS_{i.0}$ defined for ATP_i to reflect the configuration of *ACTIVE* and *INACTIVE* test cases with respect to the current operating context.[4]

(9) Veritas is invoked to determine if any *ACTIVE* test cases require fine-grained adaptation, the details of which are presented in the following section.

(10) Veritas checks to determine if test cases in the current test suite $TS_{i.j}$ were previously adapted.

(11) **If a test case in the current test suite $TS_{i.j}$ was previously adapted**, then Veritas determines which test cases perform better based upon calculated fitness values between the current test suite $TS_{i.j}$ and the backup test suite $TSi.j'$ (see Step 15).

(12) Veritas then stores the better performing test cases within the current test suite $TS_{i.j}$.

(13) **If no test case in the current test suite $TS_{i.j}$ was previously adapted**, then Veritas first determines which test cases require adaptation (i.e., test cases that have failed).

(14) Veritas creates a backup of the current test suite $TS_{i.j'}$ to provide a basis for comparison between adapted and non-adapted test cases in the following testing cycle, where the backup test suite is an exact copy of $TS_{i.j}$ prior to adaptation.

(15) Veritas executes the (1+1)-ONLINE EA on test cases selected for adaptation within the current test suite $TS_{i.j}$.

(16) Proteus executes the current test suite and then the procedure iterates back to Step 5.

[4]Test cases considered irrelevant to the current operational context are labeled *N/A*.

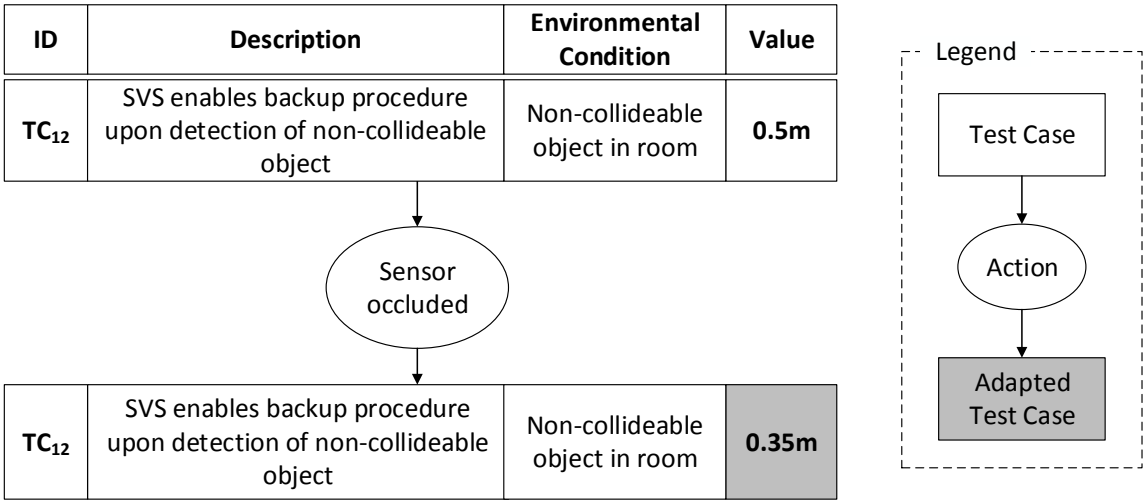
6.5.1 Test Case Adaptation.

We now discuss how test cases can be adapted at run time in the face of uncertainty. Particularly, we present two test cases that were derived according to the SVS goal model (Chapter 2, Figure 2.2), and specifically demonstrate how these particular test cases can be adapted at run time. To this end, Figure 6.6 presents two representative test case adaptations. For each example, we present the test case identifier (TC_i), a description of the test case, the conditions necessary for the test case to be executed, and the expected value of the parameter being tested.[5] We then present an unexpected occurrence that would otherwise cause the test case to fail as well as an adaptation provided by Veritas that ensures the particular test case remains relevant. For each of the presented test cases, the adaptation is assumed to be within a tolerated safety value (i.e., adaptation of the test case does not violate requirements or safety constraints).

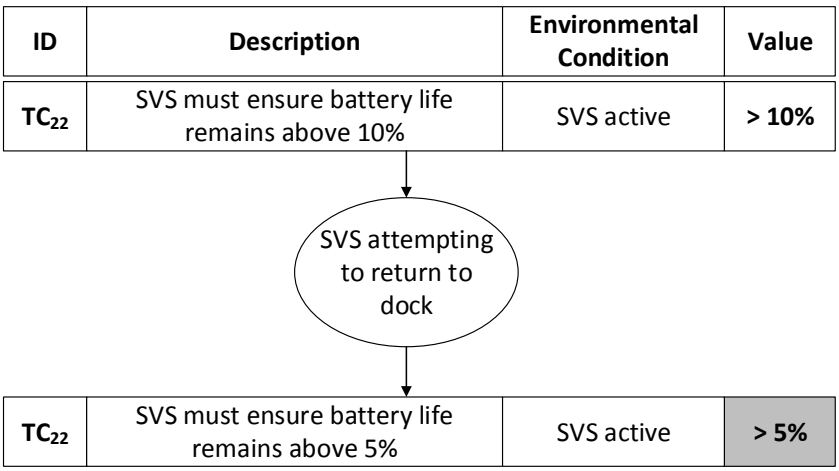
First, Figure 6.6(A) presents an example test case that is impacted by a sensor occlusion. TC_{12} tests that the SVS can successfully follow a backup procedure in the event that a non-collideable (e.g., baby or pet) object is detected within 0.5 m of the SVS. However, due to an unforeseen sensor occlusion, the effective sensing range of the SVS is lowered. As such, Veritas can adapt the expected value that TC_{12} measures to be 0.35 m. The new value ensures that TC_{12} will pass due to the new system condition being experienced. However, this adaptation is only valid if the adaptation falls within a pre-defined safety tolerance, as well oversight from a corresponding DAS utility function.

Figure 6.6(B) presents an example test case that measures the remaining battery life of the SVS, and moreover ensures that power does not fall below 10%. During the simulation, the SVS focused its efforts on cleaning the room and may not have conserved enough power to ensure that it can return to a docking station while still maintaining more than 10% remaining battery power. As such, Veritas adapts the test case parameter for TC_{22} to allow

[5]The full definition of the each presented test case is specified in Appendix .



(A) Sensor occlusion example.



(B) Power conservation example.

Figure 6.6: Examples of test case adaptation.

extra flexibility in the amount of power remaining while the SVS attempts to reach its docking station. For reference, if power fell below 5%, then a safety violation would occur and this test case would irrevocably fail. Depending on the correlated utility functions (i.e., $utility_{Goal\_K}, utility_{Goal\_M}$, c.f., Appendix), this test case failure will either be a *true negative* (i.e., utility functions are also violated) or a *false negative* (i.e., utility functions are satisficed to a degree).

6.6 Case Study

This section presents three case studies that illustrate how Proteus and Veritas can be used to provide adaptive testing at run time. The first case study presents experimental results in which Proteus was enabled to adapt test suites at run time, the second case study presents experimental results in which Veritas was used to adapt test case parameter values at run time, and the third case study presents experimental results in which we enabled both Proteus test suite adaptation and Veritas test case adaptation together. The following sections also present the simulation parameters controlling the SVS application and experimental results for both Proteus and Veritas.

6.6.1 Simulation Parameters

Each experiment was conducted using the SVS application. Particularly, the SVS was configured as an autonomous robot that must behave in accordance with a controlling goal model (c.f., Chapter 2, Figure 2.2), with specific focus on effectiveness, efficiency, and safety. The physical simulation was created within the Open Dynamics Engine,[6] and the SVS and its environment were configured as described in Section 2.2.2.

We subjected the SVS to uncertainty by using Loki [86], an approach to automatically generate unique combinations of system and environmental conditions. System uncertainty manifested in random occlusions and/or failures to randomly selected sensors. Environmental uncertainty comprised the amount, location, and distribution of dirt particles; instantiation of objects that can damage the SVS (e.g., liquid spills), instantiation of objects that the SVS must navigate around (e.g., stationary pets); floor friction (e.g., carpeting); downward steps; and room dimensions.

The input test specification comprised 72 test cases, where 17 test cases covered safety concerns and the remaining 55 test cases covered system functionality. The test cases that

[6] See `http://www.ode.org`.

monitored safety and failsafe concerns were defined to be *invariant*, and therefore precluded from adaptation. The test cases that validated system functionality were defined to be *non-invariant*, thereby allowing run-time adaptation. The fitness function weights $\alpha_{measured}$ and α_{valid} (c.f., Equation 6.5) were defined as 0.4 and 0.6, respectively, to maximize the amount of test results considered valid.

The SVS simulation was configured to run for 120 timesteps and was required to vacuum at least 50% of the small dirt particles within the room. Conversely, the SVS was required to avoid large dirt particles, liquid spills, and downward steps to avoid both internal and external damage. The SVS simulation was also instrumented to enable run-time monitoring of system and environmental conditions that are not typically available to SVS sensors. Examples include monitored variables that store data regarding the SVS decision-making logic and data structures that maintain the state of all objects within the simulation environment over time.

For the first experiment, we compared and evaluated adaptive test plans generated by Proteus for each DAS configuration with a manually-derived test plan (hereafter the "Control") that did not provide run-time adaptation capabilities. The Control test plan comprises all test cases from the input test specification and it only executes the test cases that satisfy their execution requirements (i.e., conditions necessary for execution were met). According to the IEEE standard [54], a test case must define the test criteria for the test to be successfully executed. The intent of the Control test plan is to provide coverage of all possible mode changes that may be experienced at run time by the SVS. For the second experiment, we compared test cases adapted by Veritas to test cases that were not adapted (i.e., the Control). As such, Proteus was disabled to focus on the impact of Veritas. Lastly, we combined our experiments to determine how enabling both Proteus and Veritas can enhance testing assurance. For statistical purposes, we conducted 50 trials of this experiment, and, where applicable, plotted mean values with corresponding error bars or deviations.

6.6.2 Proteus Experimental Results

For this experiment, we define the null hypothesis H_0 to state that "there is no difference between Proteus adaptive test plans and a manually-derived test plan." Moreover, we define the alternate hypothesis H_1 to state that "there is a difference between a Proteus adaptive test plan and a manually-derived test plan."

Figure 6.7 presents boxplots of the average number of test cases that should not have been executed (hereafter termed "irrelevant") between a Proteus adaptive test plan and a manually-derived test plan. A test case is considered to be irrelevant if its fitness value (c.f., Equation 6.5) equals 0.0. For those test cases designated as irrelevant, the difference between the measured and expected values is large, indicating that the test case is no longer relevant to its operating context. As Figure 6.7 demonstrates, Proteus significantly reduces the number of irrelevant test cases executed in comparison to those executed under a manually-derived test plan (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).

Testing activities were further analyzed to monitor the amount of *false positive* test cases, or instances where test case fitness falls within $[Threshold, 1.0]$ and its correlated utility value equals 0.0 (c.f., Chapter 6.4.2). As Figure 6.8 demonstrates, Proteus significantly reduces the amount of false positive test results as compared to testing with the manually-derived test plan (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). These results indicate that testing with adaptive test plans can reduce the amount of false positive test results, thus reducing the need for spurious test adaptation and, more importantly, reducing the burden of unnecessary analysis by the DAS test engineer.

As Figure 6.9 demonstrates, Proteus also significantly reduces the amount of *false negative* results, or instances when test case fitness is calculated to be within $[0.0, Threshold)$ while its correlated utility value is greater than 0.0 (c.f., Chapter 6.4.2), that were encountered during testing (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). This result indicates that Proteus adaptive test plans assist in reducing the amount of run-time testing adaptations

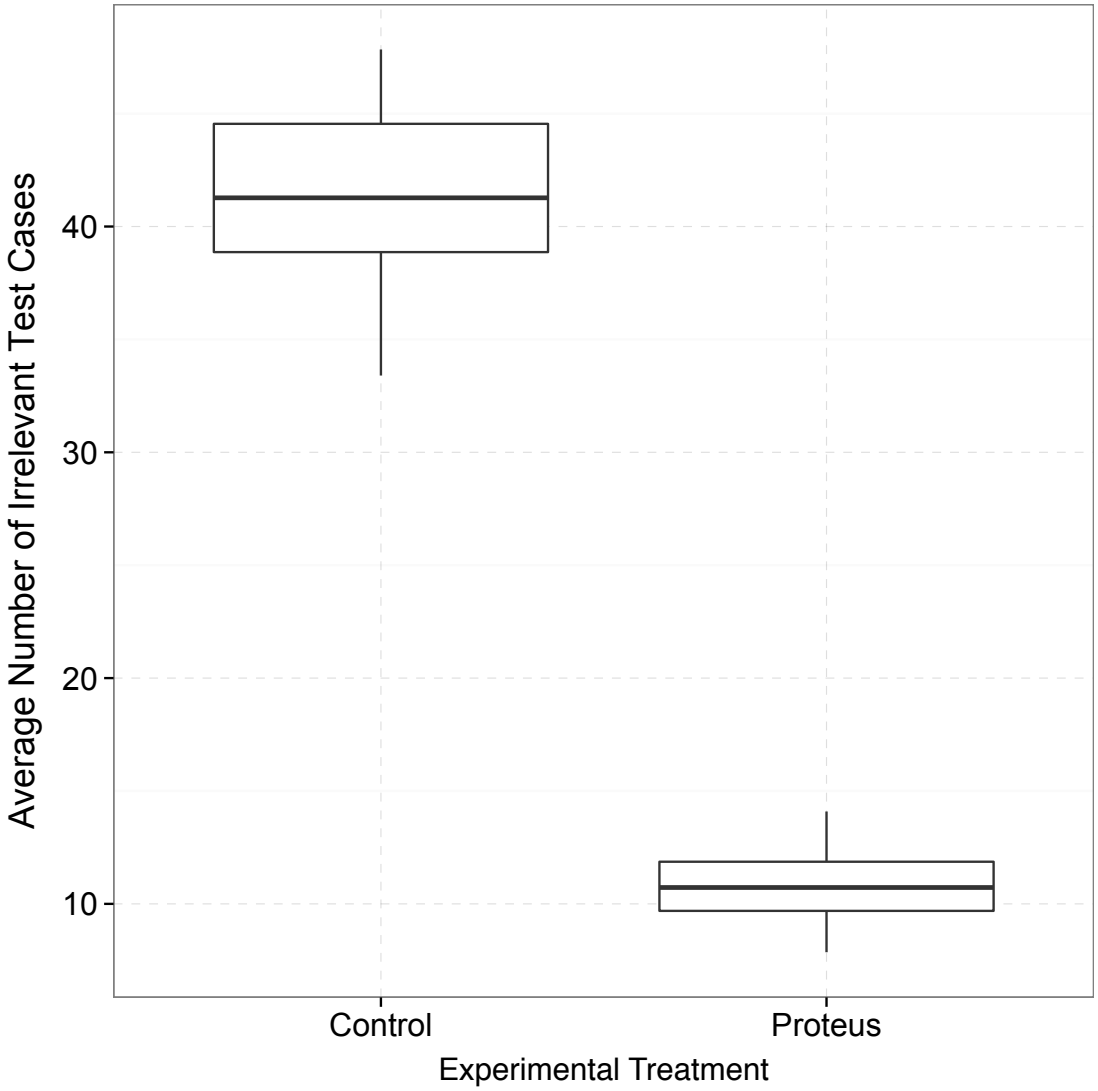


Figure 6.7: Average number of irrelevant test cases executed for each experiment.

required for the testing framework, thereby reducing the overall cost of testing the DAS and the amount of analysis effort required by a DAS test engineer.

Lastly, the total number of executed test cases were recorded to provide a measure of the overall impact of run-time testing to a DAS. Particularly, we demonstrate that Proteus significantly reduces the amount of executed test cases per testing cycle by ensuring that only relevant test cases are executed, as is shown in Figure 6.10. This figure illustrates that Proteus can reduce the amount of required effort by a testing framework at run time

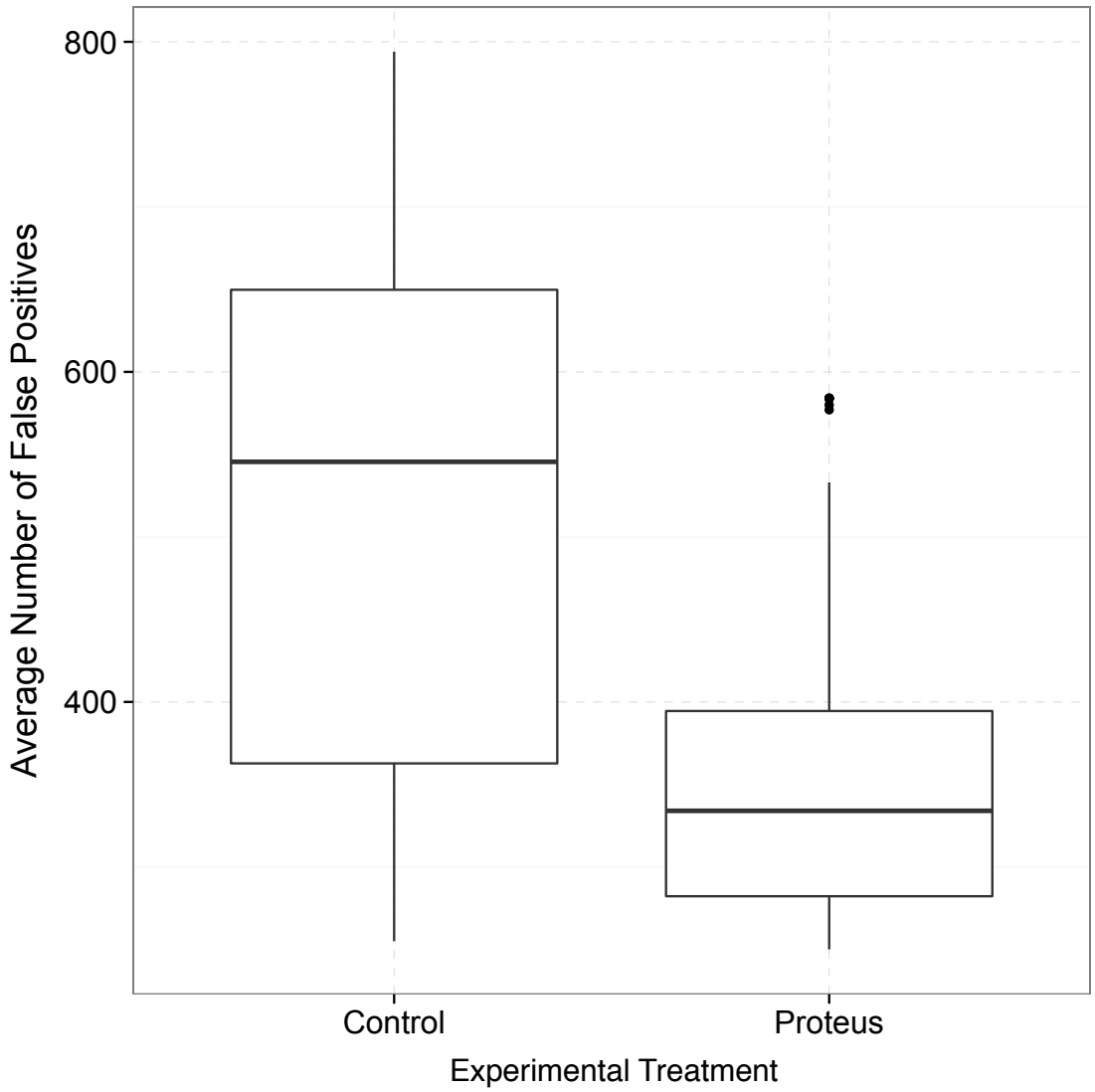


Figure 6.8: Average number of false positive test cases for each experiment.

(Wilcoxon-Mann-Whitney U-test, $p < 0.05$). The performance and behavioral impacts of run-time testing on a DAS are explored in Chapter 7.

The combined results presented in Figures 6.7 – 6.10 enable us to reject the null hypothesis H_0 and determine that a clear difference exists between Proteus adaptive test plans and a manually-derived test plan. Moreover, these results enable us to accept the alternate hypothesis H_1, based on the assumption that the manually-derived test plan was defined in relation to normal operating conditions.

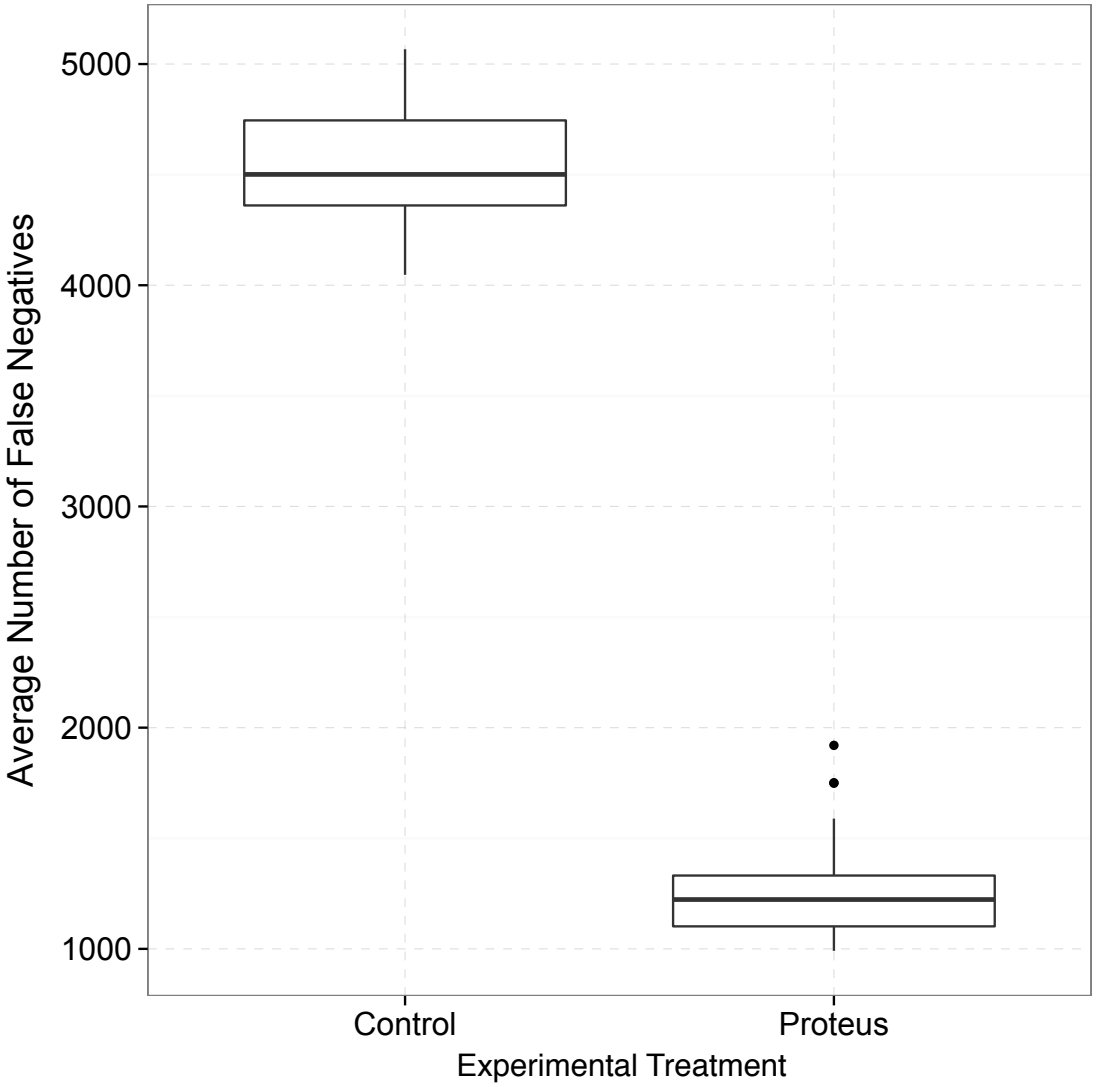


Figure 6.9: Average number of false negative test cases for each experiment.

6.6.2.1 Threats to Validity.

The research presented within this section is intended as a proof of concept to determine the feasibility of using Proteus for managing and adapting run-time testing activities. One threat to validity is if Proteus will achieve similar results within adaptive systems that do not implement mode changes. Another threat to validity occurs in the validity of the input test specification, as it must be fully comprehensive to enable complete coverage of the system requirements for testing assurance to be provided.

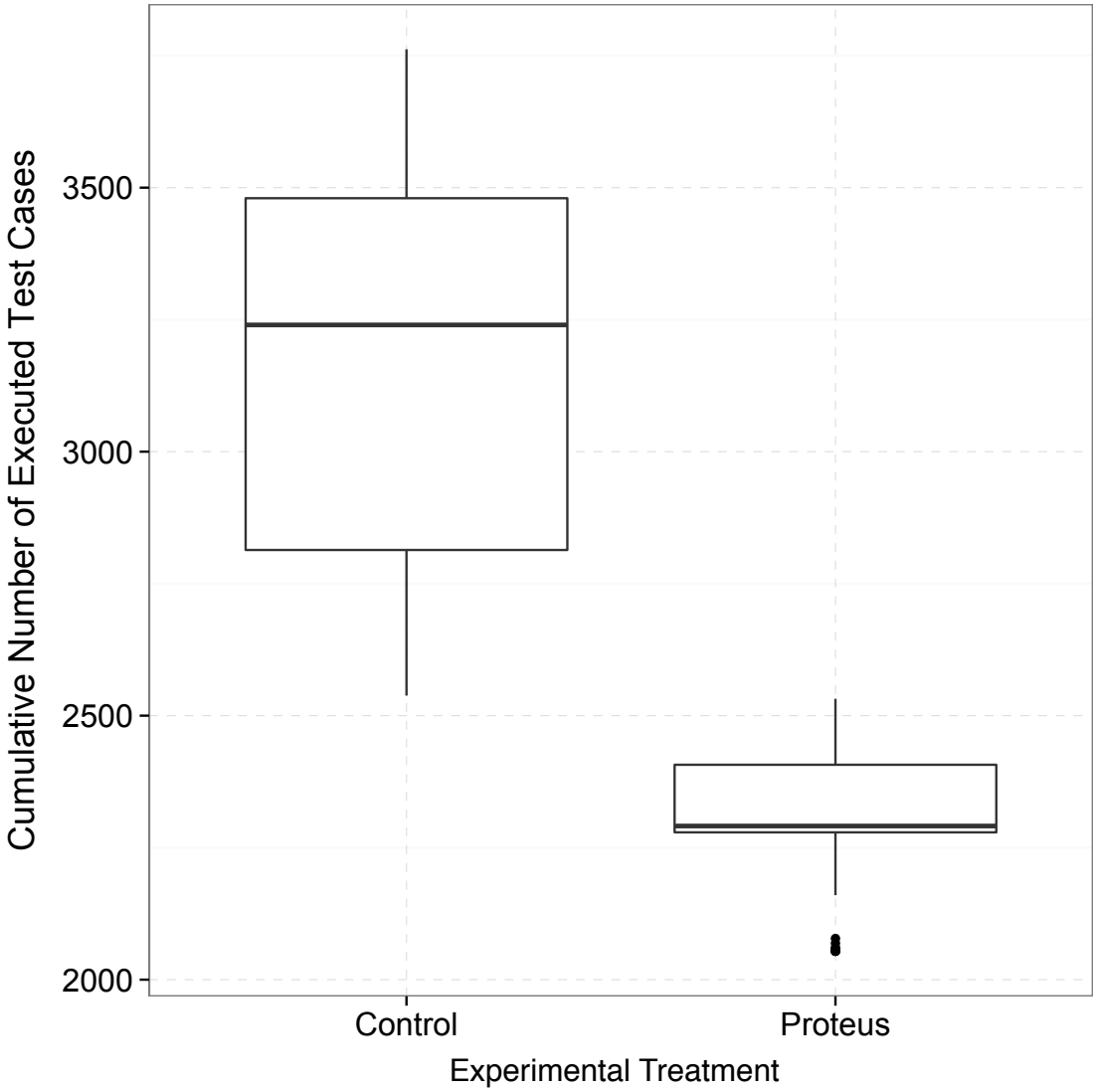


Figure 6.10: Cumulative number of executed test cases for each experiment.

6.6.3 Veritas Experimental Results

This section presents the experimental results from enabling Veritas test case adaptation on the SVS. For this experiment, we define the null hypothesis H_0 to state that "there is no difference between Veritas test cases and unoptimized test cases." We also define the alternate hypothesis H_1 to state that "there is a difference between Veritas test cases and unoptimized test cases." Figure 6.11 presents boxplots of the mean fitness values of all executed test cases across different environments for both Veritas and the Control. As the

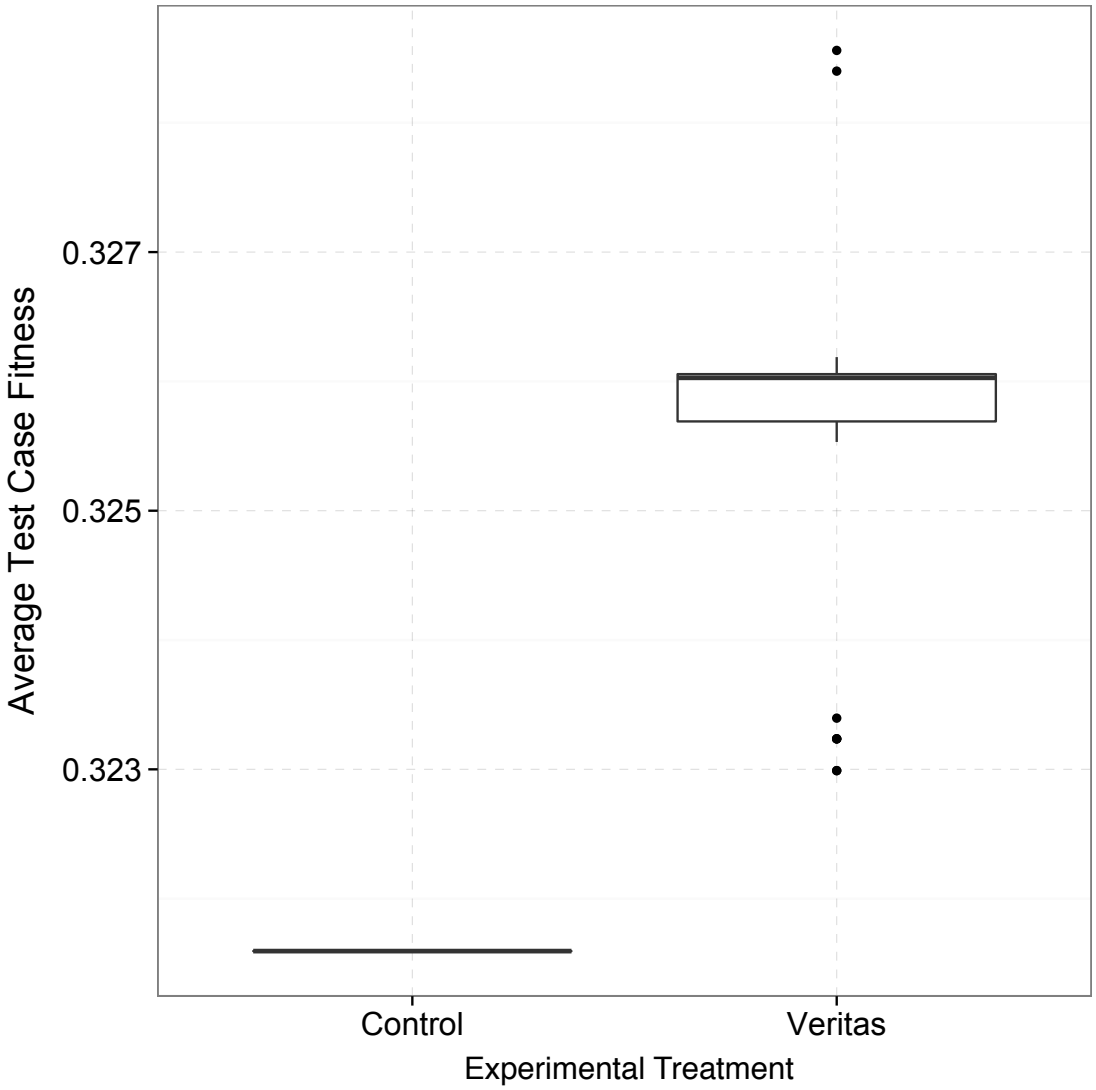


Figure 6.11: Comparison of fitness between Veritas and Control.

figure demonstrates, Veritas test cases achieve significantly higher fitness values than those that were not adapted at run time (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). Moreover, these results demonstrate that test cases adapted by Veritas consistently remain relevant to their changing environment, whereas the non-adapted test cases quickly lost relevance. These results enable us to reject H_0 that no difference exists between adapted and non-adapted test cases.

Next, Figure 6.12 presents the mean number of test case failures experienced throughout the simulation for both Veritas and the Control while the corresponding utility values for

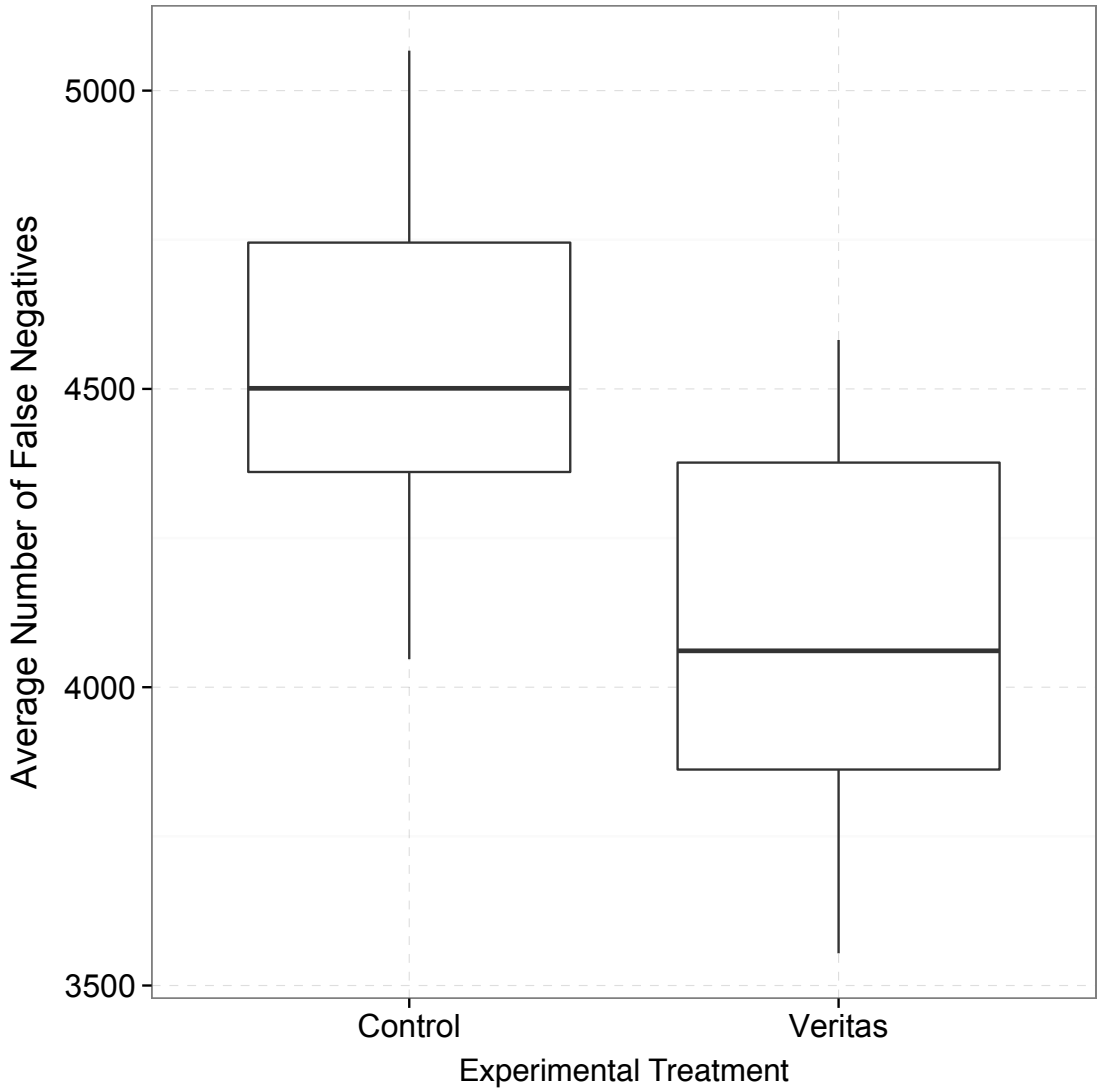


Figure 6.12: Comparison of false negatives between Veritas and Control.

each test case was satisfied. This type of failure is considered to be a *false negative*. As is demonstrated by the figure, adapting test case parameter values with Veritas can minimize the amount of false negatives experienced at run time in the face of changing environmental conditions, whereas the non-adapted test cases could not handle the unanticipated environmental conditions and therefore became irrelevant.

As Veritas can successfully maximize fitness (Figure 6.11) and minimize the amount of false negatives incurred during testing (Figure 6.12), we can safely reject H_0 and accept H_1. Moreover, we can state that Veritas can optimize run-time test cases to fail significantly less

often under changing environmental conditions while still maintaining a valid representation of the operational context.

6.6.3.1 Threats to Validity.

This experiment was intended as a proof of concept to determine the feasibility of adapting test case parameter values at run time to support the MAPE-T feedback loop. Veritas was applied to an adaptive system that used mode changes to facilitate adaptation. One threat to validity is if Veritas can achieve similar results in adaptive systems that do not use mode changes as a basis for adaptation. Another threat to validity is the validity of the derived utility functions, as they play a major role in determining the validity of test results. Furthermore, the performance impact of Veritas on a live system was not explored. Lastly, the overall representativeness of the test cases with respect to the system and encountered environmental conditions is another threat to validity.

6.6.4 Combined Experimental Results

Lastly, we enabled both Proteus test suite adaptation and Veritas test case parameter adaptation together to demonstrate the impact of using both techniques simultaneously, reusing the experimental setups defined in the previous sections. To this end, Figure 6.13 presents the average number of executed irrelevant test cases for each experiment. In comparison to the Control in which no test adaptation was performed, both Proteus and Veritas significantly reduce the number of irrelevant test cases (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). As Figure 6.13 shows, Proteus incurs far fewer irrelevant test cases than Veritas incurs alone. Moreover, combining Proteus and Veritas does not significantly reduce irrelevant test cases further than Proteus alone (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). This result indicates that, while Veritas does reduce irrelevant test cases, Proteus provides a much greater overall impact.

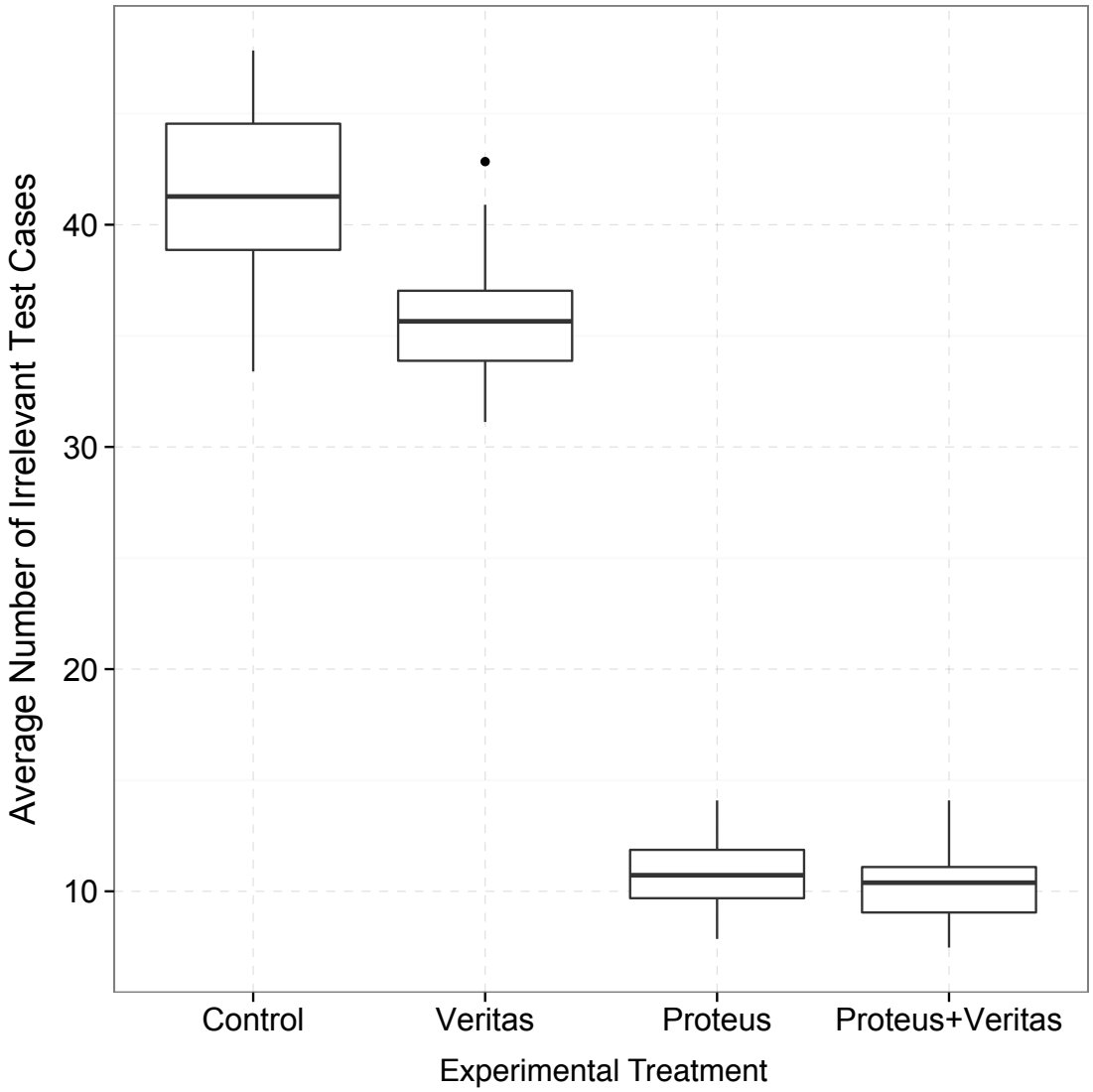


Figure 6.13: Average number of irrelevant test cases for combined experiments.

Next, Figure 6.14 presents the amount of false positive test cases executed for each experiment. In this case, only Proteus reduces the amount of false positives significantly in comparison to the Control experiment (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), indicating that test suite adaptation provides a larger benefit than test case adaptation in reducing false positives.

Following, Figure 6.15 presents the amount of false negative test cases executed for each experiment. Here, both Proteus and Veritas significantly reduce the amount of false

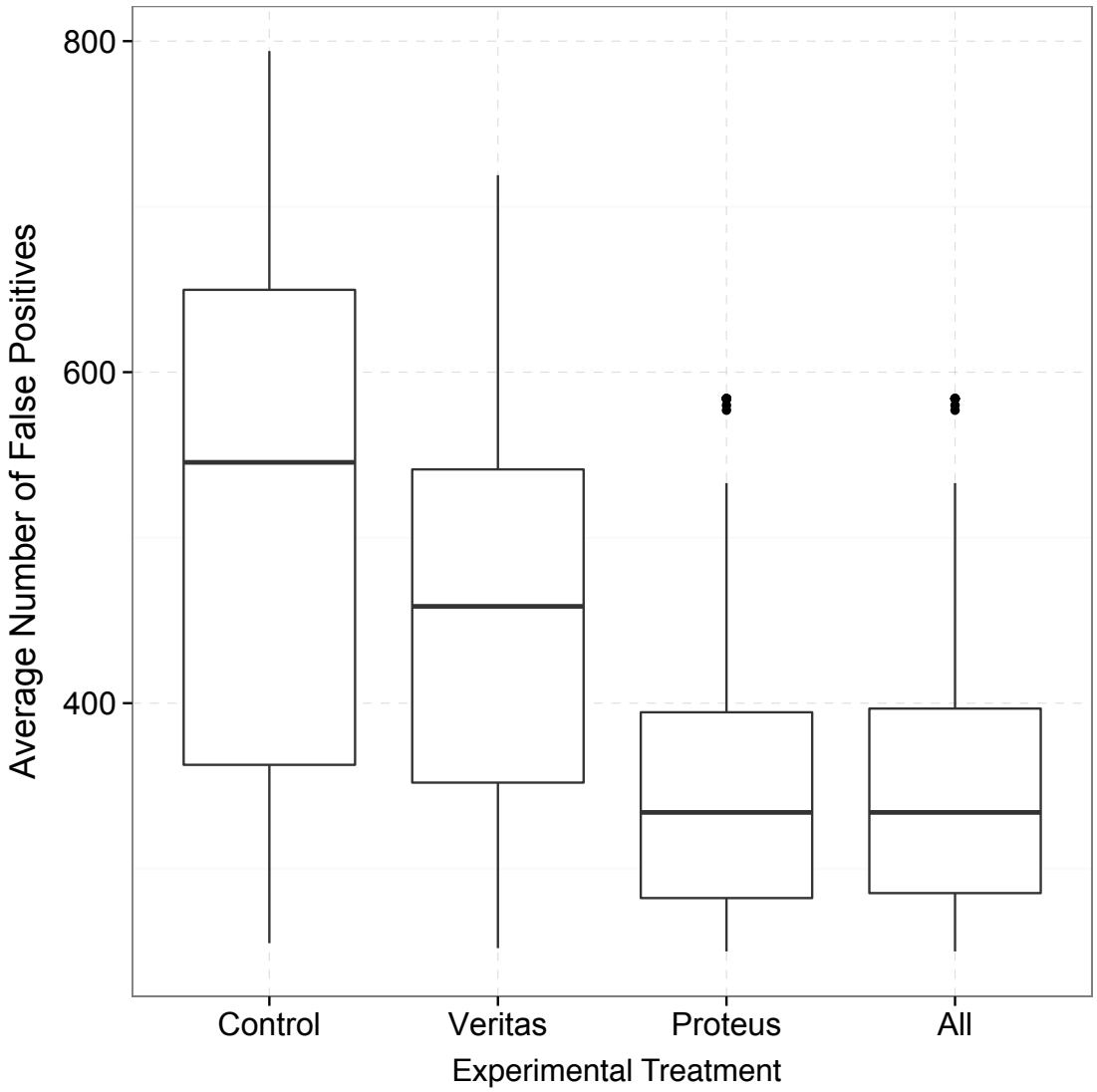


Figure 6.14: Average number of false positive test cases for combined experiments.

negatives (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), however Proteus once more provides a far greater overall impact than does Veritas.

Lastly, Figure 6.16 presents the average test case fitness of SVS execution over several consecutive environments. While Veritas does provide a fitness boost for executed test cases, employing Proteus adaptive test plans introduces a further overall increase in fitness, as only relevant test cases are executed. This result occurs as Veritas may not have converged to a fully-optimal solution, while Proteus gains an advantage by executing test cases specified by adaptive test plans as relevant to the current operating context, and therefore better-

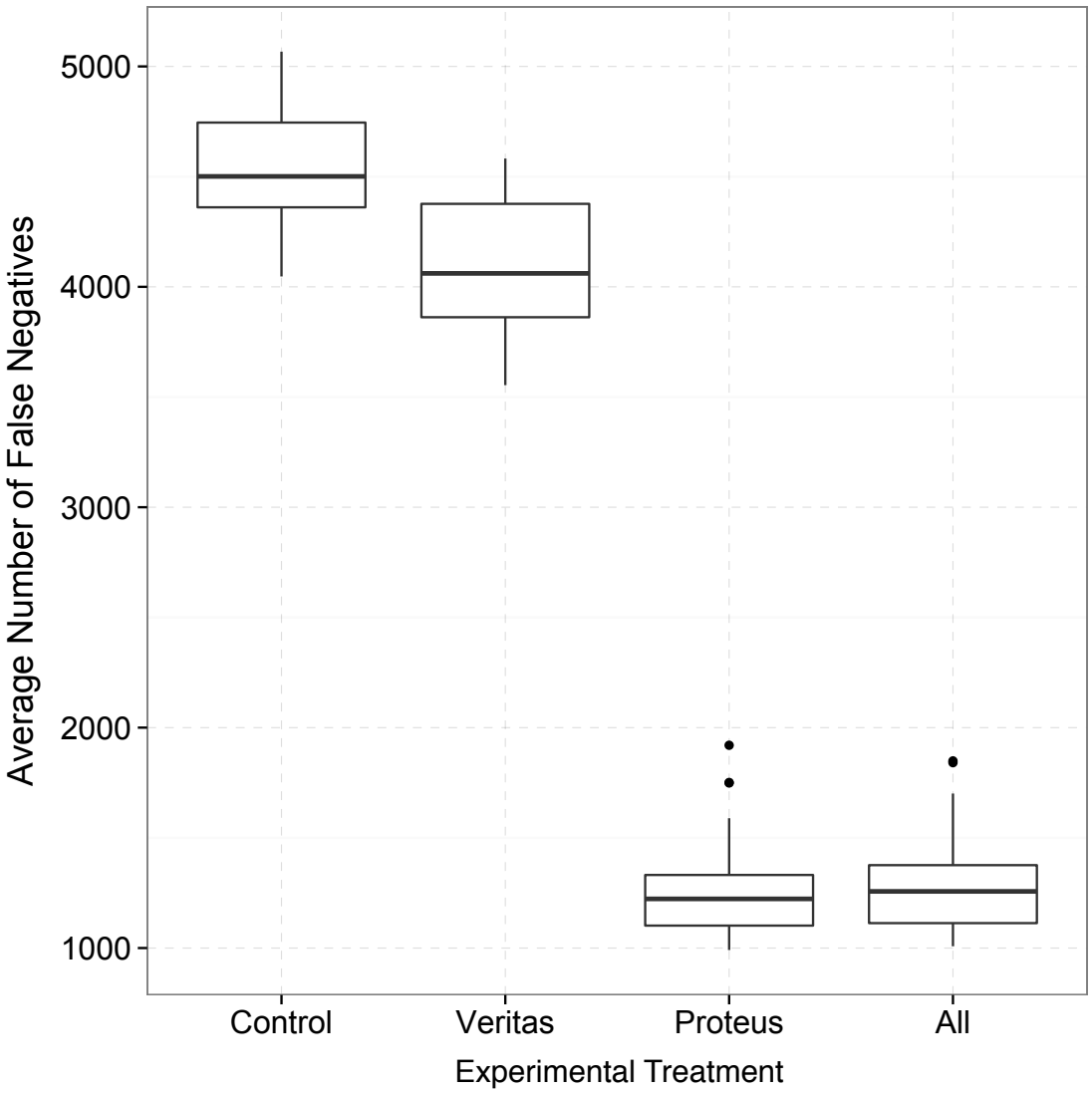


Figure 6.15: Average number of false negative test cases for combined experiments.

performing in comparison to Veritas test cases. Together, Proteus and Veritas perform significantly better than the Control that performs no test adaptation (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), indicating that the combination of Proteus and Veritas provides a clear advantage for run-time testing.

In summary, Proteus and Veritas enable test adaptation to improve the relevance of run-time test cases. Proteus ensures that test suites remain relevant to their operational context by requiring that only relevant test cases are executed, thereby reducing the amount of executed irrelevant test cases. Furthermore, Veritas ensures that test case parameter values remain

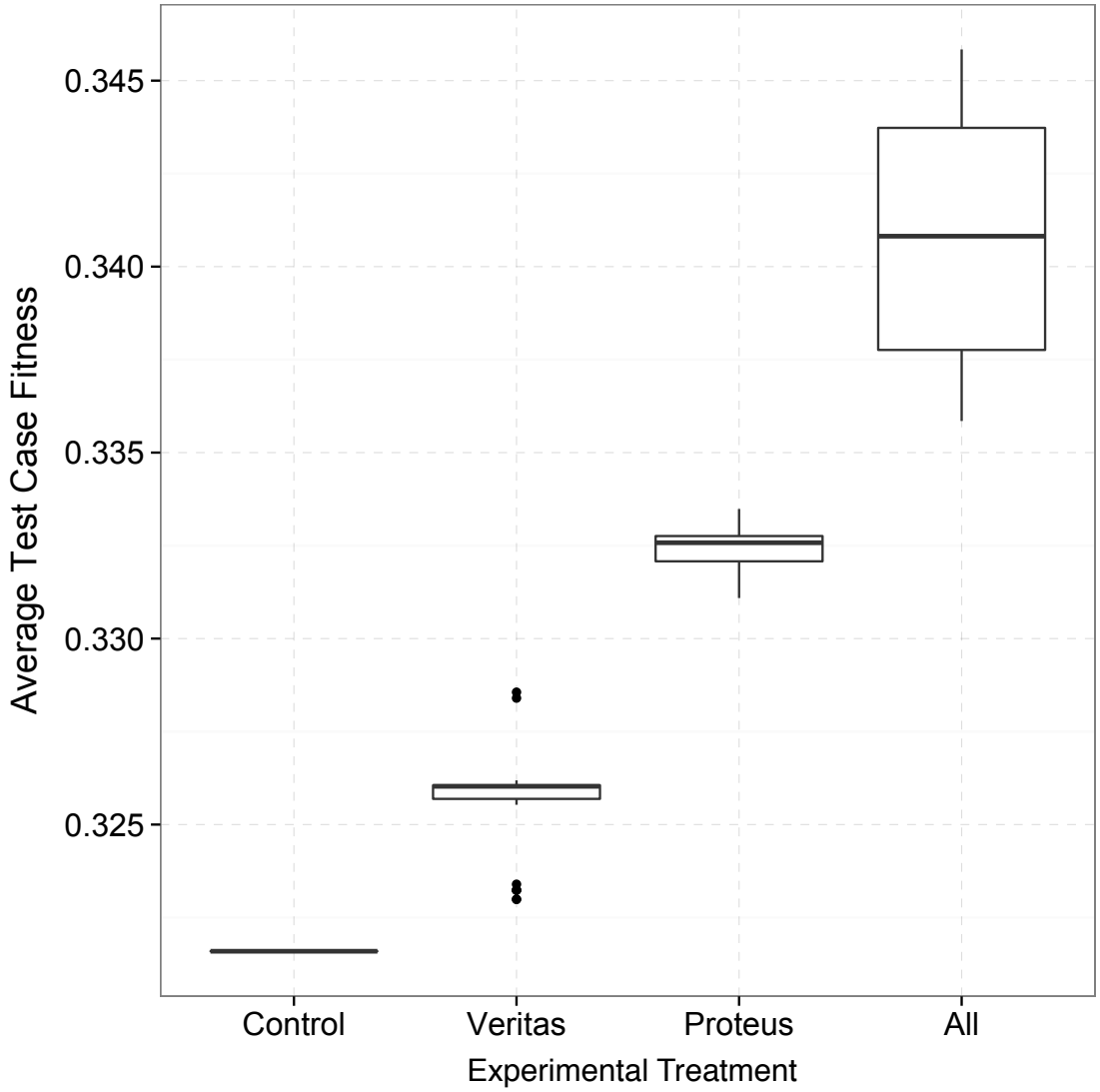


Figure 6.16: Average test case fitness for combined experiments.

relevant to their operational context by ensuring that parameter values adapt alongside the environment. Together, Proteus and Veritas can increase overall test case relevance in the face of uncertainty.

6.7 RELAXation of Test Cases

This section presents our investigation into the feasibility of using RELAX operators to facilitate adaptive testing. Particularly, we examined how RELAX, a fuzzy logic-based

language for specifying flexibility in requirements, can be extended to the testing domain to provide flexibility in executing test cases under uncertain conditions. In this section, we introduce our technique for applying RELAX operators to test cases, describe and present our experiment in which RELAX operators were both manually and automatically applied to a test specification, and discuss the implications of this study.

RELAX [21, 113] was initially developed for the requirements domain to introduce flexibility to requirements and provide a means for tolerating uncertainty. As such, we extended the RELAX approach to the testing domain to investigate if RELAX operators can provide flexibility to test cases that may be no longer be indicative of their operating context as a result of uncertainty. Similar to Veritas, RELAX operators can only be applied to *non-invariant* test cases to ensure that both safety and failsafe concerns are continually satisfied. Moreover, our approach follows the existing RELAX approach [113] in that uncertainty factors, monitoring agents, and mathematical relationships between each must be defined for each use of a RELAX operator. Our approach extends the existing RELAX approach by requiring that each test case be linked to at least one requirement or goal to provide both traceability and coverage.

In the case of requirements or goals, a fuzzy logic function is applied to each associated utility function to provide flexibility in the level of satisfaction, based upon the calculated utility value. Ideally, a test case can be RELAXed in a similar fashion. Specifically, test cases derived for Proteus and Veritas each specify a fitness function (c.f., Table 6.2) that captures the difference between the measured and expected value for each test case. Given that the fitness function is normalized to a value between $[0.0, 1.0]$, a fuzzy logic function can be applied to the fitness function to provide extra flexibility in satisfaction with the resulting distance between the measured and expected value. For the purposes of this dissertation, we reused the set of RELAX operators presented in Table 2.1.

To investigate test case RELAXation, we both manually and automatically introduced RELAX operators to non-invariant test cases. For the manual approach, a DAS test engineer

identified a set of target test cases most conducive to RELAXation. The automatic approach applied a genetic algorithm in a similar fashion to the AutoRELAX technique introduced in Chapter 3.2.2, where test cases replaced goals as targets for automated RELAXation.

We compared both automated and manually RELAXed test specifications to a Control that did not provide any RELAXation or test adaptation over 50 trials to establish statistical significance. Figure 6.17 presents boxplots comparing the average test case fitness between experimental treatments where test cases were automatically RELAXed, manually RELAXed, and not RELAXed (i.e., the Control). Furthermore, we compared these results to the average test case fitnesses generated by run-time test adaptation provided by both Proteus and Veritas. These results indicate that introducing RELAX operators, either manually or automatically, does not significantly improve test case fitness (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), especially in comparison to run-time test case adaptation.

Next, Figure 6.18 presents boxplots of the average number of test case failures recorded throughout execution, where a test case failure occurs when the measured test case fitness value falls below the test case failure threshold (c.f., Chapter 6.4.2). These results indicate that RELAXation does not improve run-time testing by reducing the amount of test case failures (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). Using Proteus and Veritas test adaptation does significantly reduce the amount of experienced failures, indicating again that Proteus and Veritas test adaptation continually improves the relevance of test cases to their environment.

6.7.1 Discussion

This section has presented our investigation into the feasibility of using RELAX operators to provide extra flexibility for run-time test cases within a test specification. Specifically, the fuzzy logic functions derived for each RELAX operator were intended to enable RELAXed test cases to accept a wider range of possible measured values as valid, thereby extending the relevance of RELAXed test cases to different operational contexts. Experimental results

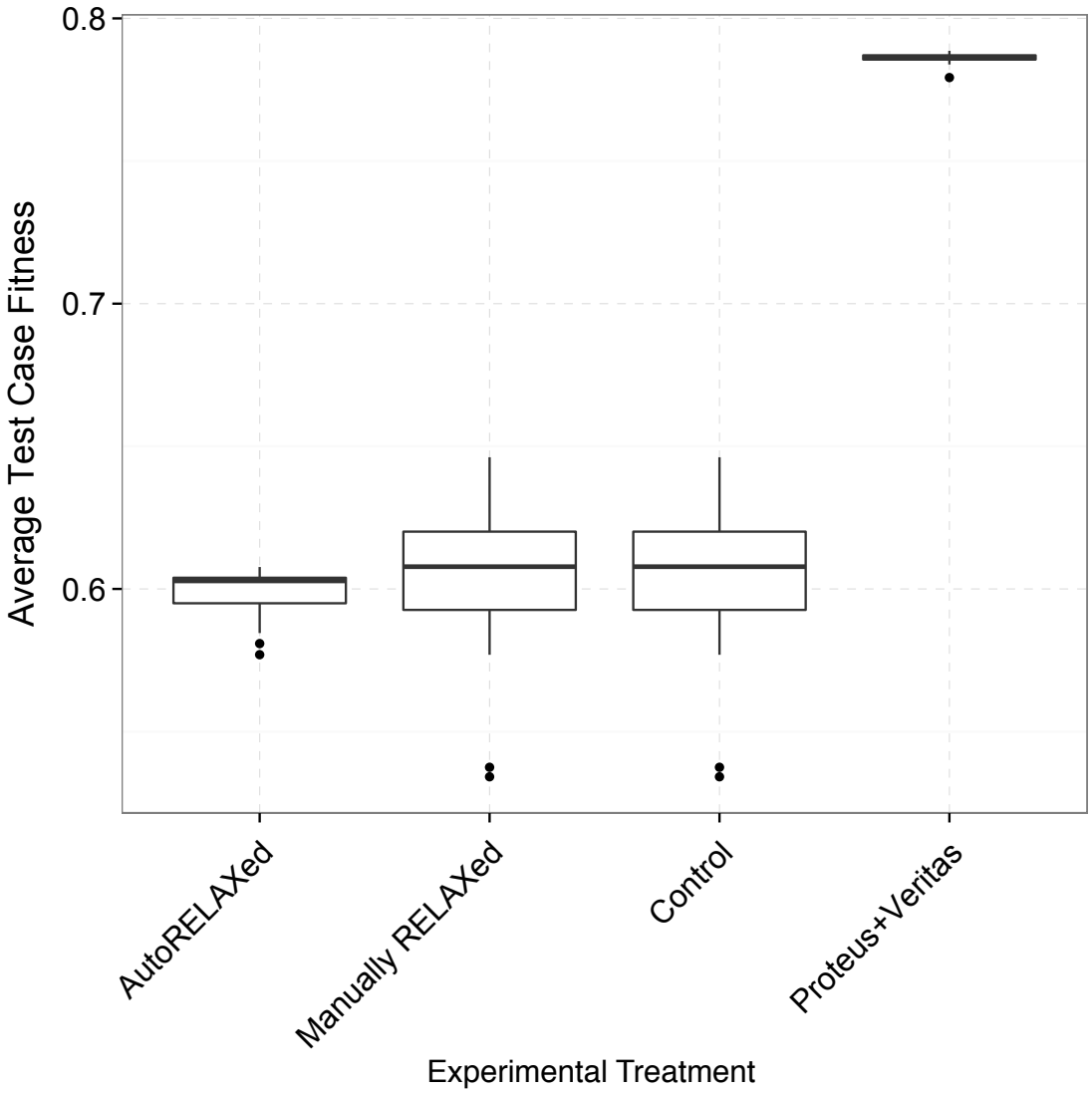


Figure 6.17: Comparison of average test case fitness values for RELAXed test cases.

indicate that a test specification with RELAXed test cases does not perform significantly better than a non-RELAXed test specification. Moreover, run-time test adaptation provided by Proteus and Veritas performs significantly better than a RELAXed test specification. As such, Proteus and Veritas adapt test suites and test case parameters at run time to optimize testing activities with respect to changing environmental conditions, whereas RELAX operators introduce flexibility into test cases by using fuzzy logic functions to enable a test case to tolerate a wider range of values than it could without RELAX. The proactive adaptation provided by Proteus and Veritas to ensure that test cases are relevant to their operating

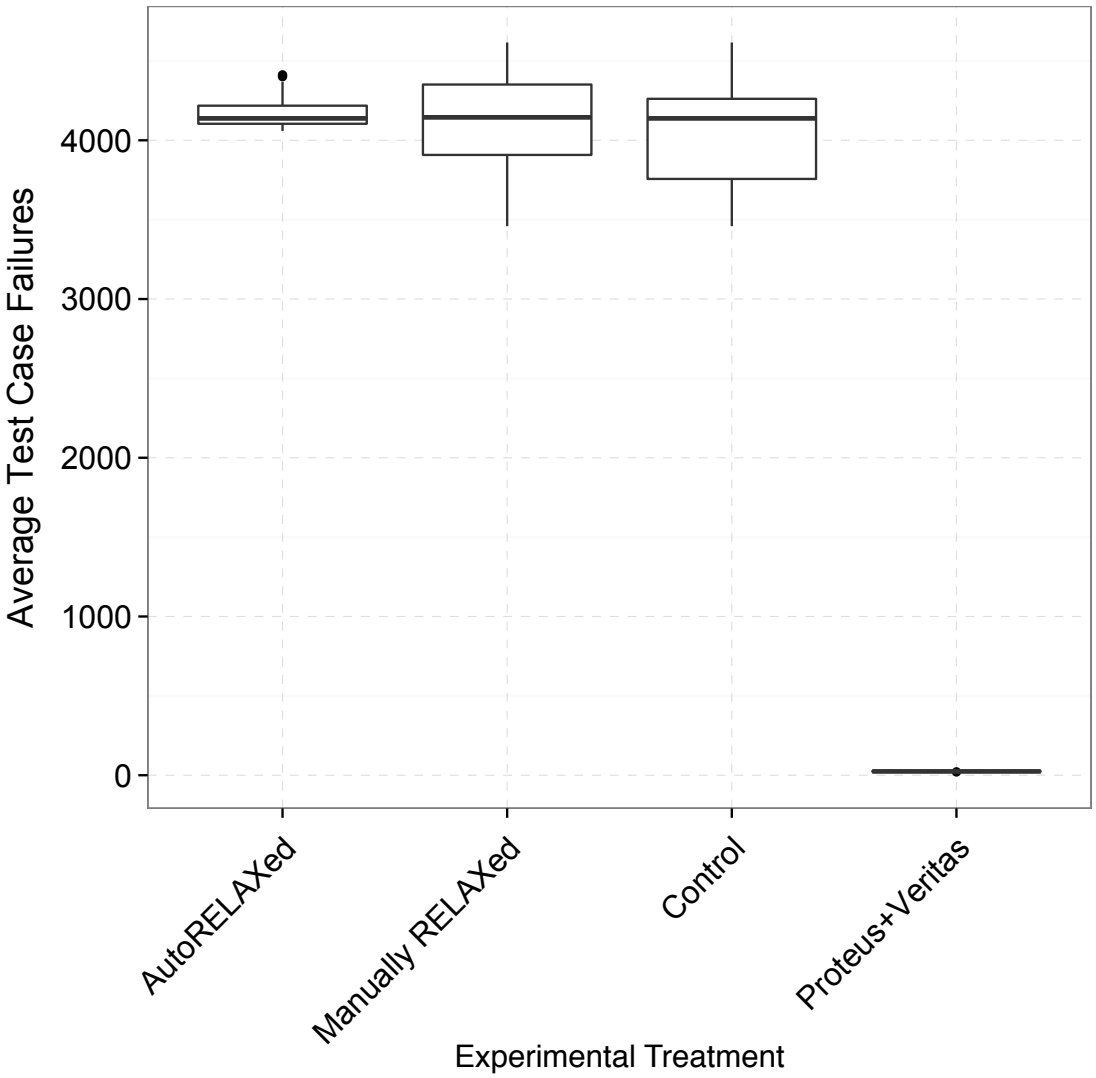


Figure 6.18: Comparison of average test case failures for RELAXed test cases.

context demonstrates a significant advantage in the average test case fitness and number of failures encountered, whereas the RELAXed test specification may still have been too rigid to provide any significant benefit. These results indicate that RELAXation of test cases may not be a feasible strategy for extending the relevance of test cases to changing environmental conditions.

6.7.2 Threats to Validity

The research presented within this section is intended to be a proof of concept to determine the feasibility of applying RELAX operators to test cases to tolerate uncertainty at run time. As such, one threat to validity is the validity of the derived test specification, as the safety ranges must be correctly defined to ensure that safety or failsafe concerns are not violated. Another threat to validity is if similar results can be achieved in a DAS application that does not implement mode changes.

6.8 Related Work

This section presents existing research that is related to run-time testing. Particularly, we overview prior work in search-based software testing, run-time testing, test plan generation, and test case selection.

6.8.1 Search-Based Software Testing

Search-based software testing leverages search-based techniques, such as evolutionary computation, simulated annealing, and hill climbing, to the field of software testing. Search-based techniques have been successfully applied to automated test case generation, including applications in model-based testing, regression testing, and structural testing [51, 52, 69]. EvoSuite [40] and Nighthawk [3] are testing frameworks that use evolutionary computation to generate test suites and instantiate unit tests, respectively. Proteus does not directly leverage search-based techniques, however Veritas applies search-based techniques at run-time. In contrast, other search-based applications tend to focus at design-time optimization. Using evolutionary techniques at run time enables a DAS to react to and adapt towards live conditions, rather than requiring the system be taken offline for bug fixes or code optimizations.

6.8.2 Run-Time Testing

Testing software at run time provides insights into the behavior and correctness of a system while executing in different operating conditions. Previously, run-time testing has been implemented using models [50] and as a variation of reinforcement learning [107]. By extending an assertion language, unit testing has been incorporated into a run-time verification framework [71]. Run-time software testing has previously been accomplished by recording execution traces and then replaying them on a secondary computer in parallel while examining those traces for faults [103]. Markov models have been used to mitigate uncertainty in a run-time monitoring framework [38], and agent-based approaches have also been used to facilitate run-time testing in distributed systems [78]. While each of these approaches facilitate run-time testing, Veritas combines evolutionary search with goal-based validation. Furthermore, Proteus focuses on maintaining test plan and test case relevance as system and environmental conditions change.

6.8.3 Test Plan Generation

Automatic generation of software test plans is a field that has been extensively studied. One approach to automated test plan generation uses a requirements specification and formal grammars to automatically define a test plan [6]. In this approach, requirements are converted to a finite state automata (FSA) and a grammar is then derived from that FSA. Test plans are then generated from the grammar and can be used during testing activities. Conversely, Proteus generates new test plans based upon a predefined default test plan and then executes test plans at run time based upon monitored system and environmental conditions.

Automated planning approaches have also been applied to testing graphical user interfaces (GUIs) [70]. Here, artificial intelligence techniques were used to anticipate actions taken by users who are actively using the GUI, resulting in a set of testing goals. The algo-

rithm then generates a partially-ordered plan that realizes the testing goals and generates a set of related test cases. Conversely, Proteus automatically generates a test plan by analyzing run-time monitoring information to determine which test cases are appropriate based upon the operational context.

6.8.4 Test Case Selection

Automatic selection and prioritization of test cases has also been extensively studied, with a particularly excellent survey covering the search-based software engineering domain [51]. In this survey, test case selection is concerned with selecting a representative set of test cases, and test case prioritization is concerned with optimizing the order of test case execution. However, the surveyed techniques tend to focus on design-time approaches, whereas Proteus selects test cases at run time. Tropos [78] is an agent-based implementation that uses a testing agent to continuously validate the behavior of other agents in the system by performing randomized testing. Conversely, Proteus generates test plans that are targeted towards each DAS configuration based on each set of operating conditions.

6.9 Conclusion

This chapter has introduced our approach for adaptive run-time testing. We first introduced Proteus, a technique for automatically generating adaptive test plans to ensure that only relevant test cases are executed at run time. Next, we introduced Veritas, a technique for adapting test case parameter values at run time to ensure that test cases remain relevant to changing environmental conditions. Together, these approaches support the MAPE-T feedback loop by enabling the creation and execution of test suites and test cases at run time. We have demonstrated both Proteus and Veritas on the SVS application that was required to clean a room effectively, efficiently, and safely while mitigating uncertainty in the system and environment. These uncertainties included random sensor occlusions and/or

failures, randomly placed dirt particles, downward steps to be avoided, and randomly placed obstacles that may cause damage to the SVS or to the object itself. Experimental results indicate that providing run-time adaptation capabilities for testing activities significantly increases the *relevance* of executed test cases. Possible future directions for this work include investigating other search-based techniques, such as simulated annealing or multi-objective optimization, for run-time use to automatically derive a representative set of test suites and optimize test case parameters. Lastly, another direction includes providing feedback to the MAPE-K loop to assist in performing self-reconfigurations.

Chapter 7

Impact of Run-Time Testing

This chapter provides an analysis of the impact that run-time testing has on a DAS from both a performance and a behavioral standpoint. We first motivate the need to analyze the impact of testing on a DAS. Next, we describe the metrics we use to quantify each type of impact on the DAS. Following, we discuss optimizations to the run-time testing framework to reduce the overall impact of run-time testing on the DAS. Lastly, we summarize our findings and propose future optimizations to our work.

7.1 Motivation

Run-time testing provides a valuable layer of assurance for a DAS. However, the relative impact that a testing framework imposes upon the DAS must also be considered when performing run-time testing, as the addition of testing activities can require additional processing time, extra memory overhead, and/or unexpected changes to DAS behavior. To this end, we analyze how our run-time testing framework impacts a DAS at run time in terms of extra performance overhead or unexpected behavioral changes. We define these metrics in the following section.

7.2 Analysis Metrics

This section presents the metrics used for analyzing the performance and behavioral impact that run-time testing imposes on a DAS. To study the effect of these metrics on a DAS, we analyze how run-time testing affects the RDM application at run time.

7.2.1 DAS Performance

We quantify DAS performance based upon two key metrics: *total execution time* and *memory footprint*. The method for which we measure each metric is defined as follows.

7.2.1.1 Total execution time

In simulation, the DAS executes for a set number of timesteps and therefore the total execution time can be measured. To this end, we measure the total execution time of the function that is responsible for executing the complete DAS simulation. In particular, we use the `cProfile` Python package[1] to measure the *cumulative* time that the simulation execution function requires. Measuring the execution times of a deterministic DAS instrumented with run-time testing and with testing disabled can then provide a point of comparison for any extra time used to perform run-time testing.

7.2.1.2 Memory footprint

Extra memory may be consumed when instrumenting a DAS with a run-time testing framework. Depending on the hardware used to support the DAS, the extra memory cost may be prohibitive, particularly in embedded systems where memory is limited. Given that the RDM application has been implemented in Python, we use the `resource` package to examine the total amount of memory consumed throughout execution.

[1] See `https://docs.python.org/2/library/profile.html`.

7.2.2 DAS Behavior

While execution time and incurred memory costs are relatively straightforward to quantify, examining differences in DAS behavior is less obvious. To this end, we define two key metrics for quantifying behavior: *requirements satisficement* and *behavioral function calls*. Each of these metrics are next described in turn.

7.2.2.1 Requirements satisficement

We first examine the extent to which software requirements are satisficed during execution to examine the impact that testing may impose upon a DAS. In particular, we monitor utility values calculated based upon a provided DAS goal model to determine if any difference occurs when run-time testing is performed on a DAS. Given that a utility value is required to be within a range of $[0.0, 1.0]$, any difference in behavior can be identified based on a comparison of the utility values that quantify goal satisficement.

7.2.2.2 Behavioral function calls

We also quantify behavioral performance based upon the number of behavioral function calls invoked. We define a behavioral function call as a function identified by the DAS engineer as having an integral impact on DAS behavior. For the purposes of this chapter, we monitor the number of self-reconfigurations performed by the DAS, as a self-reconfiguration can create a major divergence in system behavior.

7.3 Baseline Results

This section presents the results found by executing the RDM in a baseline setting to determine the impact that run-time testing has on a DAS, where the baseline introduces no

optimizations.[2] In particular, we instrument the RDM application to provide the metrics that we have previously defined in this chapter. Moreover, we configure run-time testing activities on the RDM as follows:

- **(S1)**: All run-time testing activities enabled (i.e., Proteus and Veritas enabled)
- **(S2)**: Run-time testing disabled (i.e., Proteus and Veritas disabled)
- **(S3)**: Run-time testing removed (i.e., Proteus and Veritas data structures and functions removed from DAS codebase)

(S1) executes the RDM with all run-time testing activities enabled, including test case execution, Proteus test plan adaptation, and Veritas test case parameter value adaptation. (S2) does not perform run-time testing, however the data structures and functions required by the framework are still instantiated by the DAS. Lastly, (S3) completely removes the run-time testing framework from the DAS.

Figure 7.1 presents boxplots of the number of seconds required to execute the RDM simulation for each testing state across 50 trials. As these results demonstrate, performing run-time testing requires significantly more time for the simulation to complete than either disabling or removing run-time testing from the RDM application (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).

Next, Figure 7.2 presents boxplots of the total amount of memory consumed by the RDM application for each testing state. In particular, we examine the total number of kilobytes required by the RDM simulation across 50 trials. As these boxplots demonstrate, no significant difference exists in memory overhead incurred by each testing state (Wilcoxon-Mann-Whitney U-test, $p > 0.05$).

Figure 7.3 presents boxplots that display an average of all utility value calculations across 50 trials of RDM execution. Here, we average the calculated utility values across

[2]For the purposes of this experiment, Proteus is not considered to be an optimization, but a managing infrastructure and an approach for continuously ensuring test case relevance.

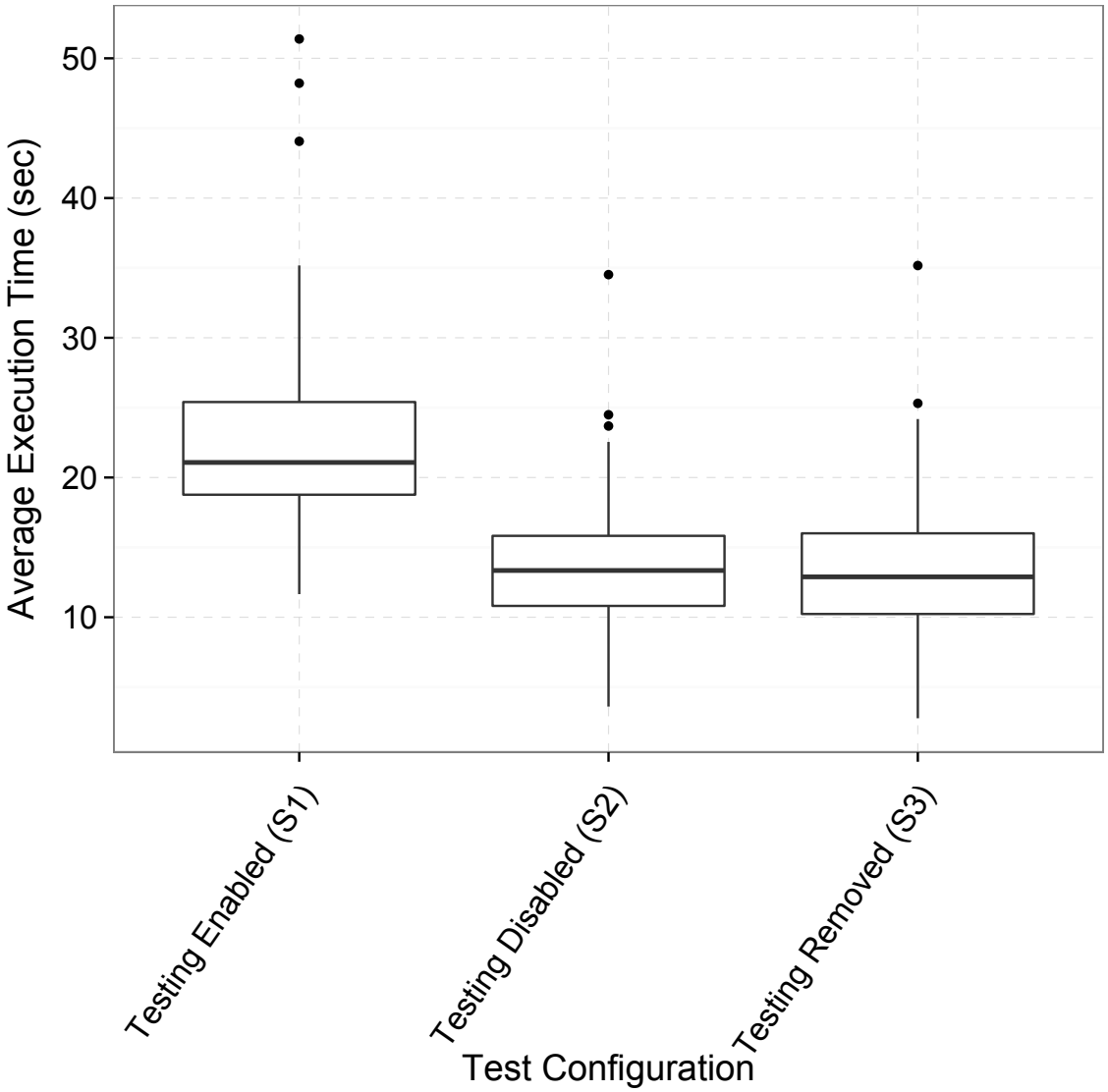


Figure 7.1: Amount of time (in seconds) to execute RDM application in different testing configurations.

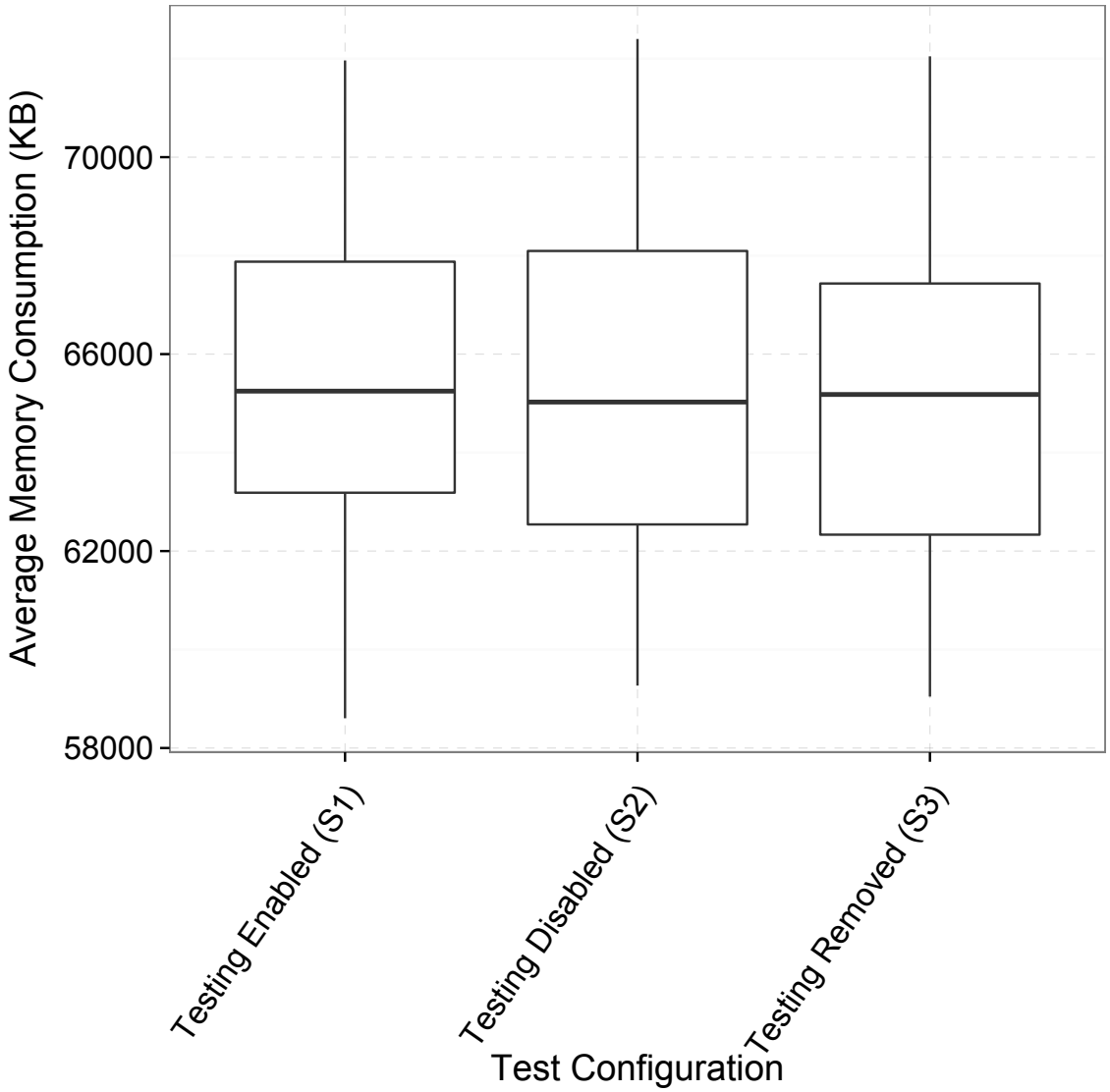


Figure 7.2: Amount of memory (in kilobytes) consumed by the RDM application in different testing configurations.

all timesteps of RDM execution into a single value to represent overall goal satisficement. These results, while exhibiting a difference in their respective means, are not statistically significantly different from each other overall (Wilcoxon-Mann-Whitney U-test, $p > 0.05$), resulting in the conclusion that our testing framework does not significantly impact DAS behavior.

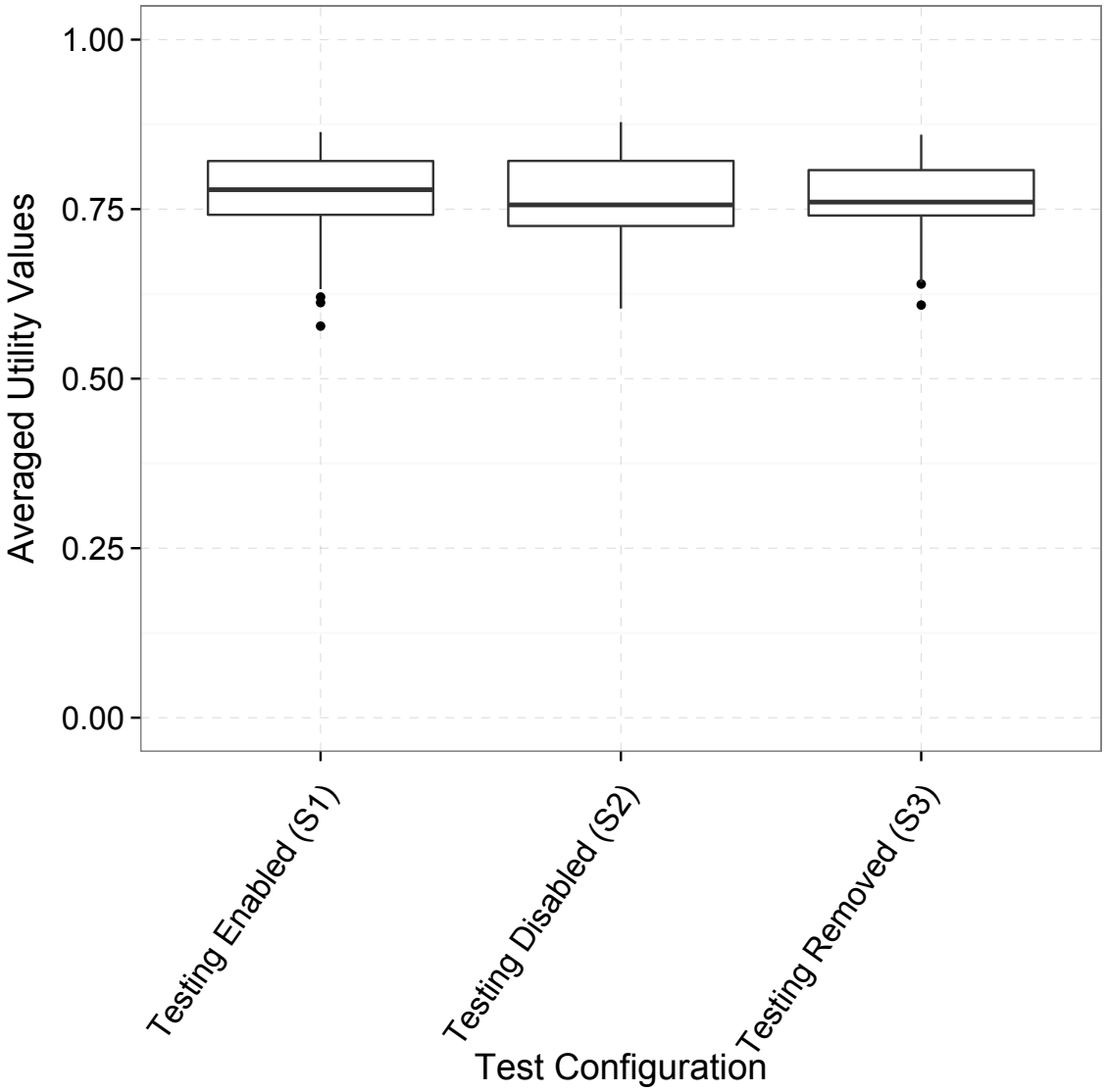


Figure 7.3: Average of calculated utility values throughout RDM execution in different testing configurations.

Next, Figure 7.4 presents boxplots that display the average number of utility violations throughout RDM execution. Statistically, performing run-time testing does not incur sig-

nificantly more utility violations as compared to disabling or removing run-time testing, as no significant difference exists between each set of data (Wilcoxon-Mann-Whitney U-test, $p > 0.05$).

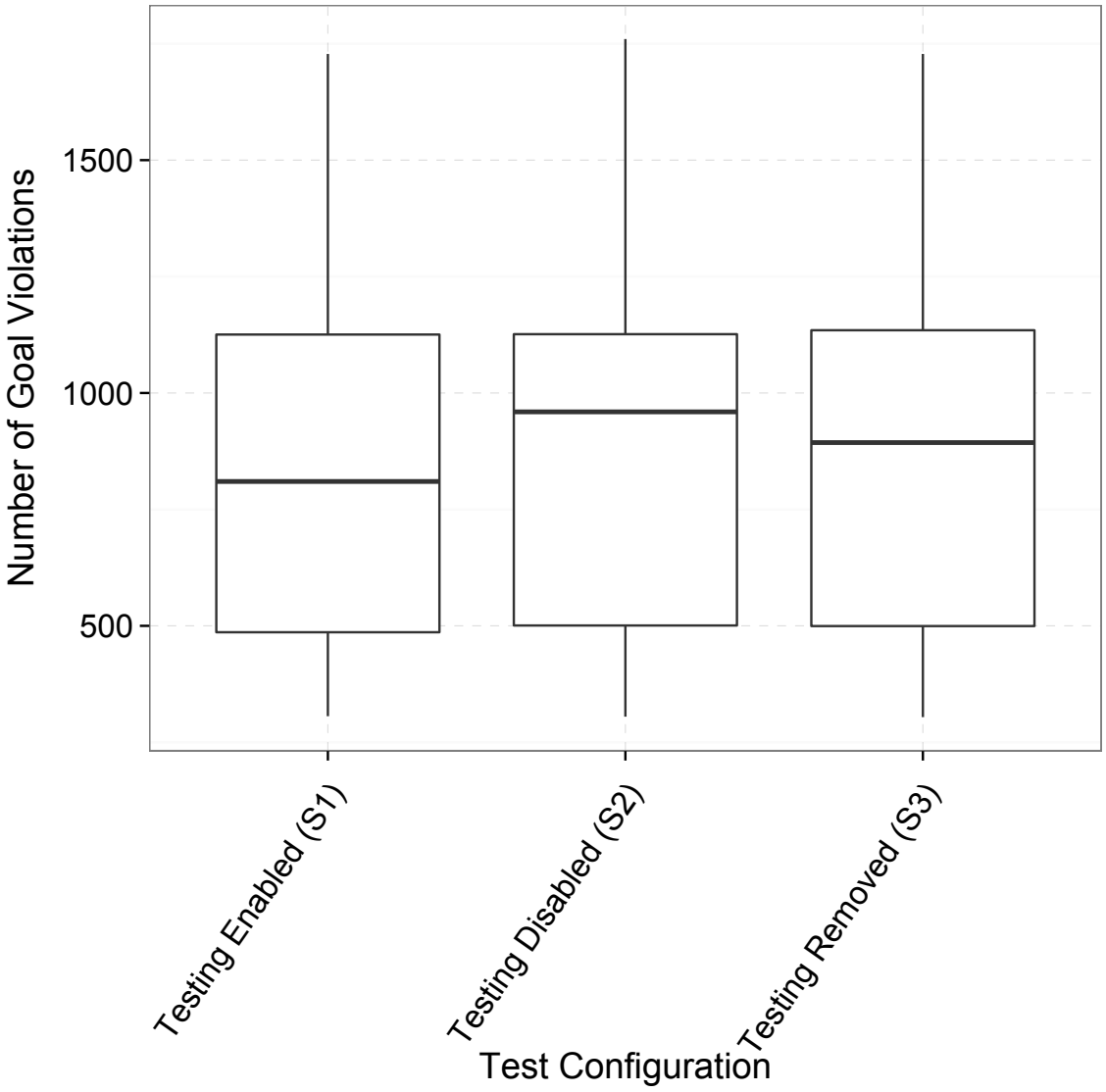


Figure 7.4: Average amount of utility violations throughout RDM execution in different testing configurations.

While the presented results in Figure 7.3 and 7.4 are not statistically significant, a clear difference exists in the presented mean values. As such, we examine RDM execution traces to determine why this difference occurs. To this end, we found that, while no difference exists in the operating context between all three testing states, a minor difference in execution

behavior does occur. Specifically, utility functions sample data provided by the RDM each timestep. When testing is enabled or available, the real time at which utility values are calculated will be slightly delayed by the extra time required to perform testing activities, resulting in a sampling of different RDM environmental states. For example, at a certain point during RDM execution, the number of passive data mirrors increased when testing was enabled, decreasing the number of active data mirrors and resulting in a partitioned network. As such, this behavior induces a utility violation in Goal (F) (c.f., Figure 2.4).

We next compare the total amount of self-reconfigurations performed by the RDM throughout its simulation and present these results in Figure 7.5. This figure corroborates the information presented in Figures 7.3 and 7.4 in that run-time testing did not incur significantly more adaptations than were found by disabling or removing testing (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). These results, coupled with the results presented in Figure 7.3 and 7.4, indicate that our testing framework does not introduce significant behavioral change to the RDM application.

The results presented in Figures 7.1 – 7.5 indicate that run-time testing only significantly impacts a DAS in the amount of execution time required, whereas memory overhead and behavior are not significantly impacted by run-time testing. The next section details our approaches for optimizing run-time testing from a performance perspective.

7.4 Optimization Approach

We now detail our optimization strategy to lessen the overall impact of run-time testing on a DAS. As such, this section details our implemented optimizations for run-time testing in terms of execution time. Specifically, we discuss how we introduce parallelization into the RDM framework. In particular, we create a separate worker subprocess using Python's `multiprocessing` package[3] to perform testing in parallel to RDM execution. The worker

[3]See `https://docs.python.org/2/library/multiprocessing.html`.

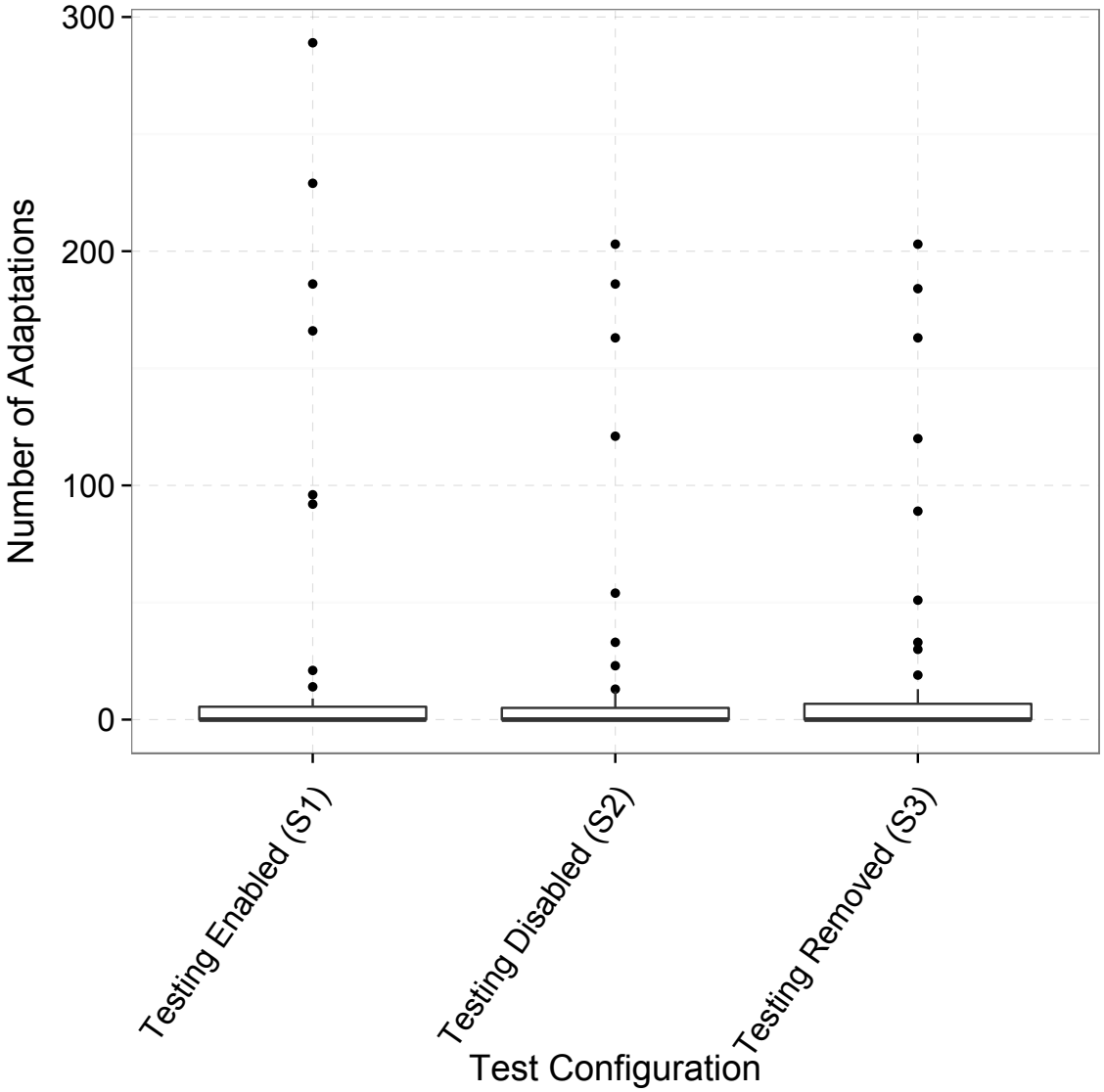


Figure 7.5: Number of adaptations performed throughout RDM execution in different testing configurations.

subprocess comprises instantiation of testing-related data structures, sequential execution of test cases, and sequential adaptation of test suites and test cases. Information particular to RDM execution, such as the number of active network links and number of quiescent data mirrors, are passed to the worker subprocess via a pipe.

We now examine the impact that parallel processing has on the performance of our run-time testing approach. To this end, we extend the execution states previously defined in Chapter 7.3 (i.e., (S1), (S2), and (S3)) to understand the impact that each technique has on RDM performance:

- **(S1)**: Proteus test plan adaptation enabled, Veritas test case adaptation enabled
- **(S1.1)**: Proteus test plan adaptation disabled, Veritas test case adaptation enabled
- **(S1.2)**: Proteus test plan adaptation enabled, Veritas test case adaptation disabled
- **(S2)**: Proteus test plan adaptation disabled, Veritas test case adaptation disabled
- **(S3)**: Testing framework removed (i.e., Control)

We compare the overall execution time for States (S1), (S1.1), and (S1.2) with their parallelized counterparts (i.e., (S1'), (S1.1'), and (S1.2')) in Figure 7.6. No comparison to a parallelized implementation is made for States (S2) and (S3) as no test execution is performed while these particular states are active. Figure 7.6 presents boxplots of the number of seconds required to execute the RDM simulation for each testing state across 50 trials.

The results presented in Figure 7.6 demonstrate the differences in execution time between each State and its parallelized counterpart. As such, parallelizing States (S1), (S1.1), and (S1.2) significantly decreases the amount of execution time required (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). Moreover, States (S1'), (S1.1'), and (S1.2') do not demonstrate a significant difference between States (S2) and (S3), indicating that parallelizing testing activities does not significantly impact run-time performance of the RDM (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).

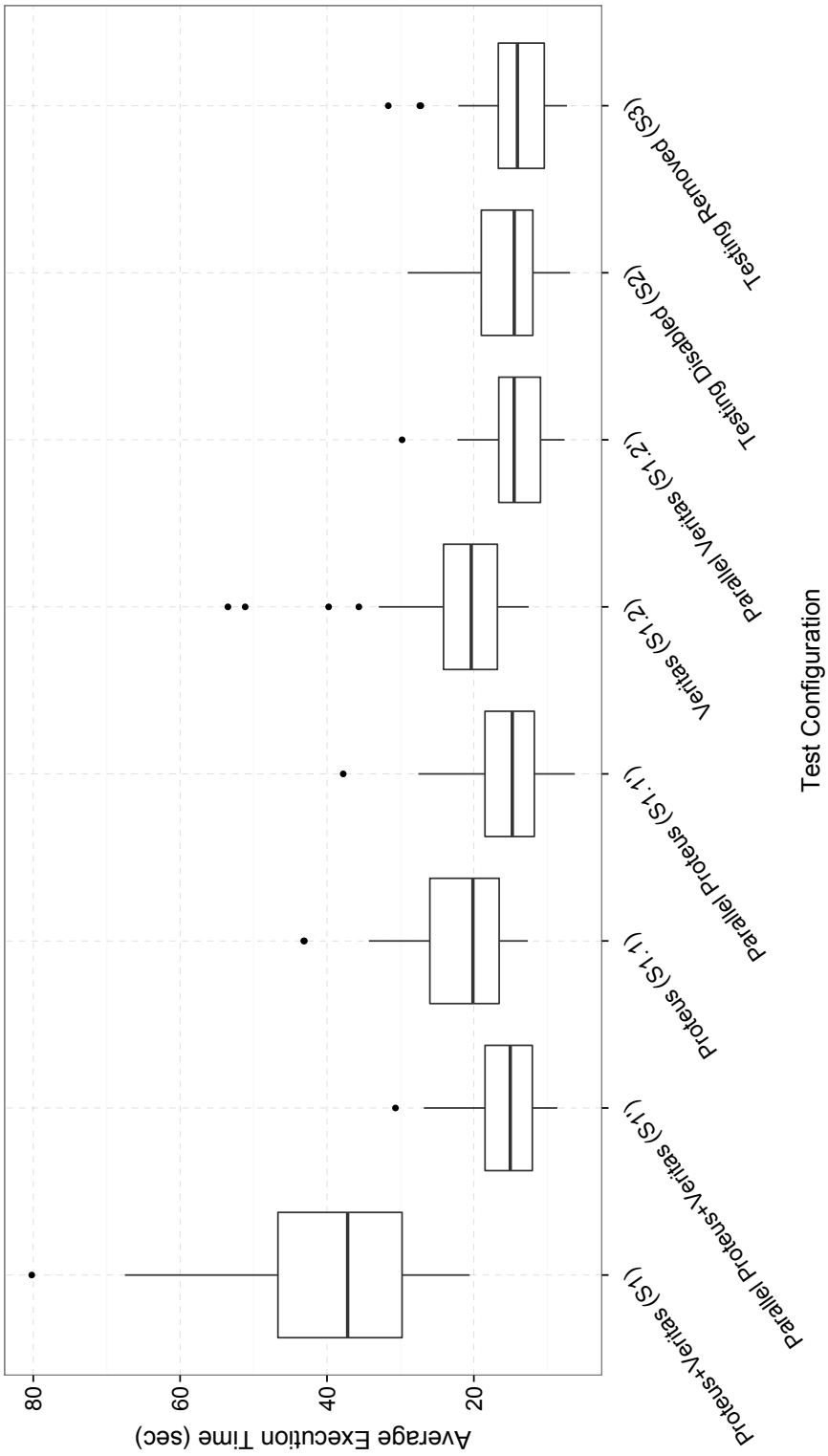


Figure 7.6: Amount of time (in seconds) to execute optimized RDM application in different parallel and non-parallel testing configurations.

To confirm our results, we now examine the RDM framework in greater detail. In particular, we examine the amount of time required to execute a controlling function that executes both network and testing activities, whereas we previously examined the execution time of the RDM simulation as a whole, including instantiation of all requisite data structures as well as execution of the controlling function. Figure 7.7 next presents boxplots of the execution time required for each testing state in terms of the controlling function. Again, parallelizing testing activities significantly reduces the execution time required.

Next, we analyze the speedup that our optimization approach provided using Amdahl's Law [2]. In particular, we use Equation 7.1 to calculate the speedup ratio:

$$S(N) = \frac{T(1)}{T(N)} \tag{7.1}$$

where $S(N)$ is the speedup ratio, $T(1)$ is the time required for the program to execute sequentially, and $T(N)$ is the time required for the program to execute with N worker processes. For our experiment, we used 1 worker process. Over 50 experimental treatments, we calculate an average $S(N)$ of 2.742, or a 274.2% speedup, in processing time when parallelization is introduced.

Following, we re-examine the behavioral impact of testing activities on the RDM in our parallelized and non-parallelized testing states. Specifically, Figure 7.8 presents the average utility value calculated during the RDM simulation, Figure 7.9 presents the average number of utility violations encountered during the simulation, and Figure 7.10 presents the number of triggered adaptations. For each figure, there again is no significant behavioral difference, suggesting that our parallelization approach did not induce significant changes to RDM behavior (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).

Lastly, we examine two other approaches to reduce the performance impact of testing. For the first approach, we create two subprocesses to execute in parallel to RDM execution, the first of which performs testing adaptation and the second of which executes the test cases sequentially. For the second approach, each test case is executed in parallel in

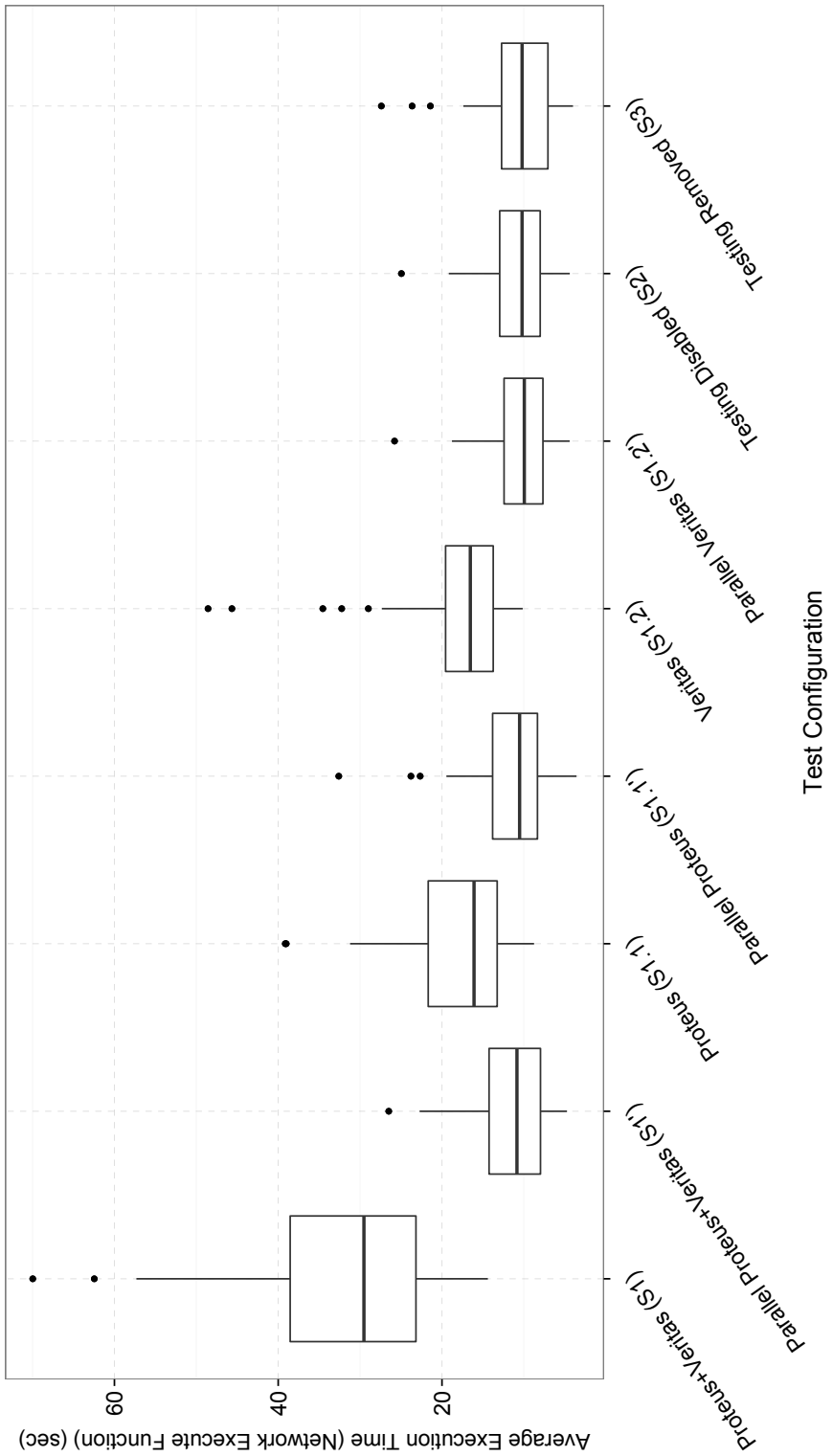


Figure 7.7: Amount of time (in seconds) to execute optimized network controller function in different parallel and non-parallel testing configurations.

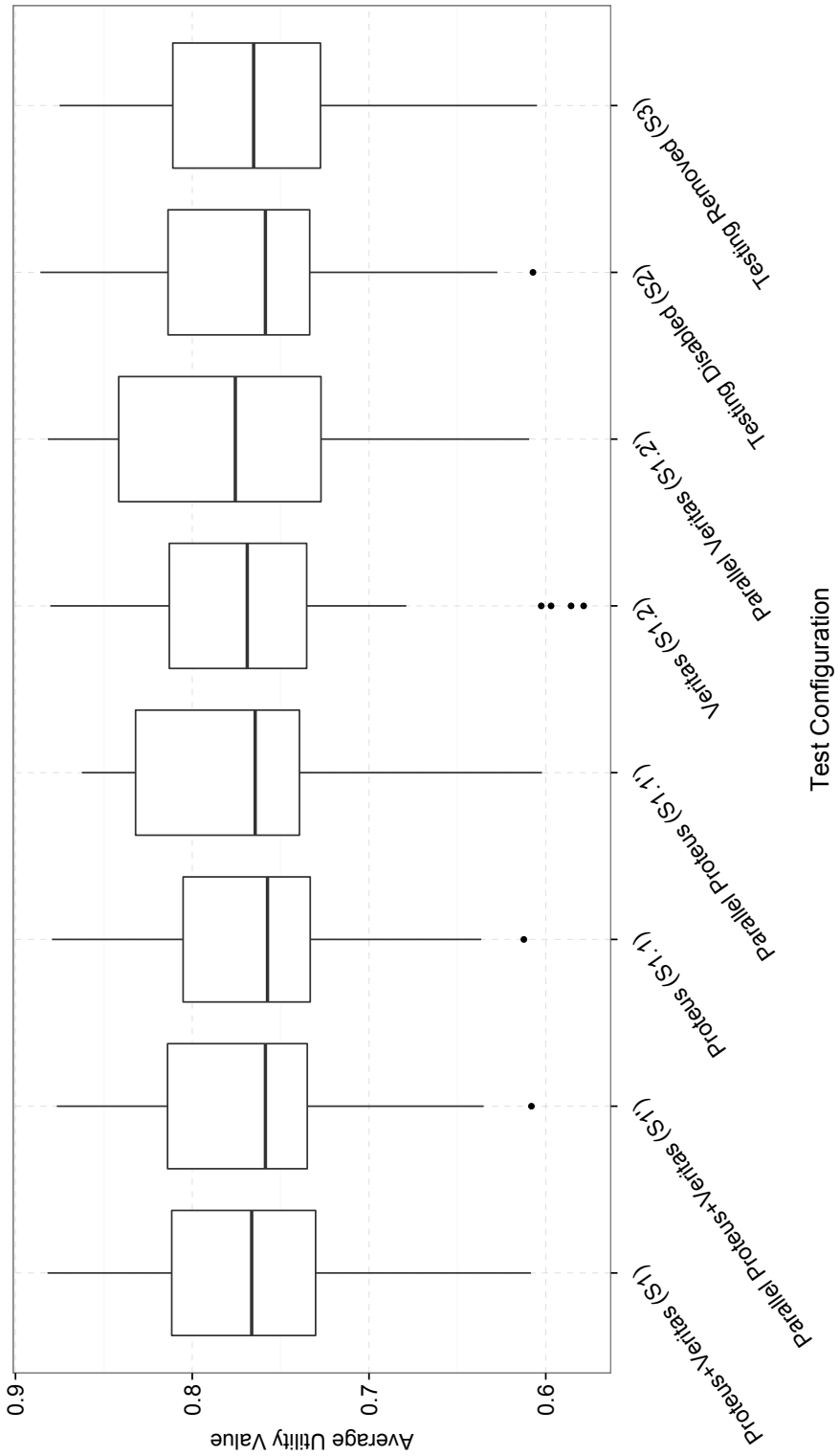


Figure 7.8: Average of calculated utility values throughout RDM execution in different parallel and non-parallel testing configurations.

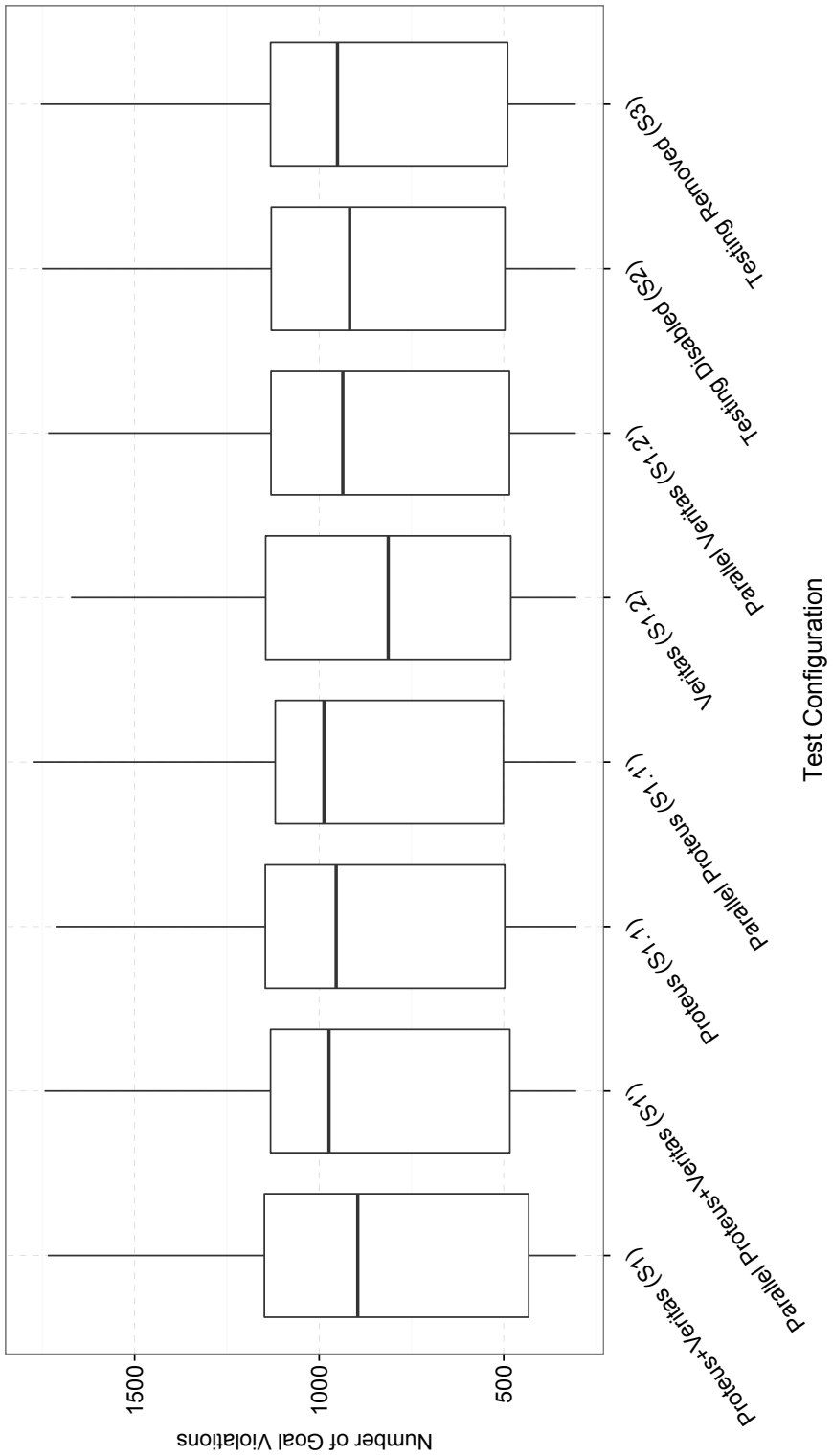


Figure 7.9: Average number of utility violations encountered throughout RDM execution in different parallel and non-parallel testing configurations.

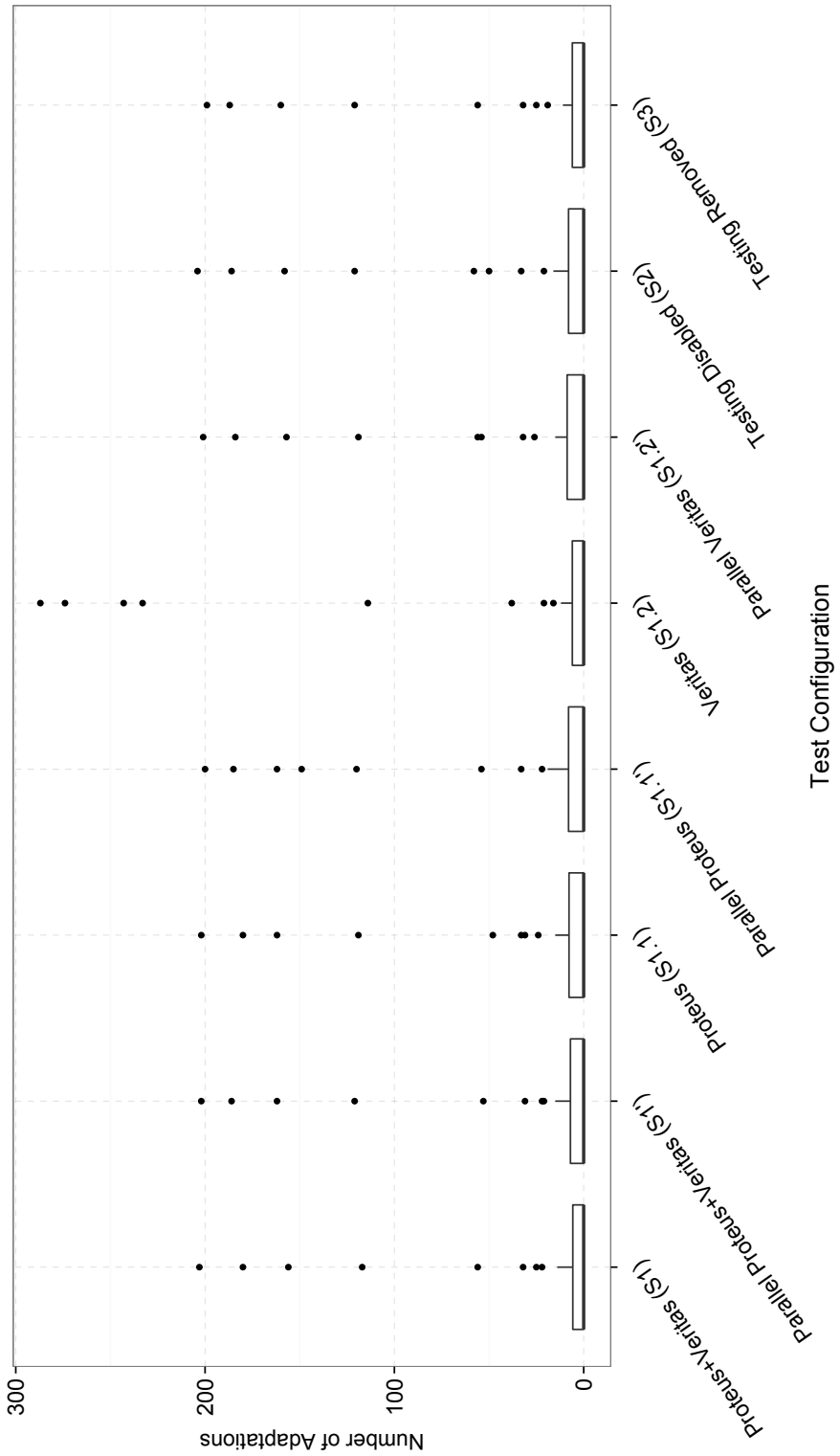


Figure 7.10: Number of adaptations performed throughout RDM execution in different parallel and non-parallel testing configurations.

separate subprocesses. In particular, no dependencies exist between our test cases and, for this case study, test case execution can be considered an embarrassingly-parallel problem. Performance slightly improves for the first approach, however is not significant enough to be considered successful. Interestingly, performance degrades for the second approach. Specifically, executing test cases in parallel significantly increases the execution time required for the RDM simulation. After further study, we determine that the increase in execution time is a result of using Python for the RDM application, as the overhead necessary to continually instantiate subprocesses far outweighs the performance benefit from executing test cases in parallel.

7.5 Related Work

This section overviews related work in optimizing software testing. In particular, we examine how physical processors can be monitored to determine when testing should be performed and how testing can be divested onto agents to remove the burden on the main processing unit.

7.5.1 Processor Cycles

Previously, Saff and Ernst [91, 92] introduced a continuous testing approach for assisting a software developer in catching bugs while the developer writes code. The continuous testing framework was integrated into a development environment to run regression tests in parallel to development, thereby increasing both development and testing efficiency. In particular, their approach monitored intervals between developer activity (i.e., writing code, compiling, etc.) and executed a set of prioritized test cases while the developer was inactive. As such, continuous testing was shown to dramatically improve both developer and testing efficiency, specifically by localizing software bugs in advance of a formal testing cycle. In comparison, our run-time testing approach parallelizes testing activities at run time, reducing

the performance impact to the DAS. However, our approach does not fully minimize the impact that run-time testing incurs on DAS performance. Therefore, regularly monitoring system activity to determine appropriate intervals for executing test cases and performing test adaptations may be an optimization to complement our current testing strategy.

7.5.2 Agent-Based Testing

We have previously discussed agent-based testing in detail in Chapter 5.3, however we now examine the physical use of agents to perform testing. Nguyen *et al.* [78] examined the use of multi-agent systems (MAS) to provide a continuous and evolutionary testing approach. In particular, the agent-based approach requires that a particular agent, or separate software system, be used solely to perform testing. As such, the performance impact incurred when performing run-time testing does not affect agents tasked with critical system functionalities, as they are not required to perform testing. Instead, the testing agent is required to perform testing and is fully dedicated to that task. Our approach to provide run-time testing uses subprocesses that execute in parallel to the DAS. While we do offload testing activities to subprocesses, the DAS and portions of the testing framework share a central processor, thereby introducing the possibility for contention of processor usage between testing activities and DAS execution. Introduction of a separate testing agent into the DAS environment, effectively creating a union of a DAS and a MAS, may be an effective approach to reducing the performance impact of run-time testing.

7.6 Conclusion

In this chapter, we have described the impact that run-time testing can have on a DAS. In particular, we examined impact based on two metrics: performance and behavior. Performance was measured based on execution time and memory overhead. Behavior was measured based on overall requirements satisfaction and the number of performed self-reconfigurations.

Our experiments indicate that our run-time testing framework has an impact on performance in terms of required execution time, however, it does not significantly impact the DAS in terms of memory overhead, requirements satisfaction, or DAS reconfigurations.

We next introduced an optimization strategy to reduce the performance impact in terms of execution time. To this end, we introduced parallelization into the RDM simulation, moving instantiation and execution of testing activities into a worker subprocess that executes in parallel to the RDM. As such, we found that parallelization can significantly reduce overall execution time of the simulation. In terms of the RDM application, introducing parallelization effectively reduced overall execution time to be comparable with disabling or removing run-time testing activities. Future investigations include studying how memory overhead can be optimized in onboard systems. Specifically, examining how test trace data can be scaled back while still providing useful information to the test engineer, offloading data storage to separate devices, and introducing extra processors or modules for additional processing capabilities are examples of future optimizations.

Chapter 8

End-to-End RDM Example

This dissertation has presented techniques for providing assurance for a DAS at varying levels of abstraction and it has demonstrated those techniques in different domains. This chapter presents an end-to-end example of how these techniques can provide assurance for the RDM application at its different levels of abstraction. In particular, we present how each technique can be used sequentially to increase overall assurance. First, we present the configuration of the RDM application for each portion of the end-to-end investigation. Next, we discuss how a goal model describing RDM requirements can be automatically refined with RELAX operators to provide requirements-based assurance, as well as the overall scalability of this technique. Following, we present our technique for analyzing the RDM codebase to discover unexpected paths of execution. Lastly, we instrument the RDM application with an adaptive, run-time testing framework to provide run-time testing-based assurance. Where applicable, this chapter reuses previously presented experimental results.

8.1 RDM Configuration

This section describes the configuration of the RDM application. In particular, we define the configuration of the RDM network and sources of uncertainty.

The RDM configuration comprises parameters that define the RDM network as well as parameters that define the sources of uncertainty that can affect the RDM during execution. First, Table 8.1 presents a subset of the parameters that configure the RDM network. Specifically, we provide the number of timesteps the simulation is specified to run for all experiments, the possible network topologies that may be selected, the number of data mirrors available to the network, the number of messages that must be distributed by the end of the simulation, and the available budget and the cost of each network link in terms of U.S. dollars.

Table 8.1: Subset of RDM network parameters.

Configuration Parameter	**Value**
Timesteps	300
Network topology	$\{Complete, Grid, Tree, Random, Social\}$
Number of data mirrors	$[15, 30]$
Number of messages	$[100, 200]$
Budget	$[\$350,000.0, \$550,000.0]$
Network link cost	$[\$10,000.0, \$15,000.0]$
...	...

Next, Table 8.2 describes a subset of the parameters that define the sources of uncertainty. Here, the function that generates random values for the RDM application is provided with a different seed and distribution for each replicate. Furthermore, the probabilities describing the failure rates, fuzz rates, and rates of possible message errors that can occur during simulation are also presented.

Table 8.2: RDM sources of uncertainty.

Configuration Parameter	**Value**
Random seed	$[1, 50]$
Random number distribution	$\{Binomial, Exponential, Normal, Poisson, Uniform\}$
Probability of data mirror failure	$[0.0\%, 5.0\%]$
Probability of data mirror sensor failure	$[0.0\%, 5.0\%]$
Probability of data mirror sensor fuzz	$[0.0\%, 20.0\%]$
Probability of network link failure	$[0.0\%, 10.0\%]$
Probability of dropped message	$[0.0\%, 20.0\%]$
Probability of delayed message	$[0.0\%, 10.0\%]$
Probability of corrupted message	$[0.0\%, 10.0\%]$
...	...

8.2 Requirements-Based Assurance

This section first presents a case study examining how AutoRELAX enables requirements-based assurance and then investigates the scalability of AutoRELAX with respect to the RDM application.

8.2.1 AutoRELAX Case Study

We now discuss how assurance can be provided for a DAS at its requirements level. First, we examine how RELAX can be used to augment the RDM goal model to provide requirements-based assurance. In particular, we study the impact of AutoRELAX as applied to the RDM application. Next, we examine how the SAW can be used to enhance requirements-based assurance by tailoring fitness sub-function weights to different operational contexts. Lastly, we provide an investigation into the scalability of AutoRELAX.

8.2.1.1 RDM Goal RELAXation

We applied AutoRELAX to the RDM application to explore how AutoRELAX enables requirements-based assurance. To facilitate readability, we now recreate the RDM goal model in Figure 8.1 and AutoRELAX fitness functions in Equations (8.1) – (8.3). As such, we examined the resulting fitness values generated by AutoRELAXed goal models, manually RELAXed goal models, and unRELAXed goal models. For the manually RELAXed goal model, we introduced RELAX operators to Goals (C), (F), (G), and (H) (c.f., Figure 8.1).

$$FF_{nrg} = 1.0 - \left(\frac{|relaxed|}{|Goals_{non-invariant}|}\right) \tag{8.1}$$

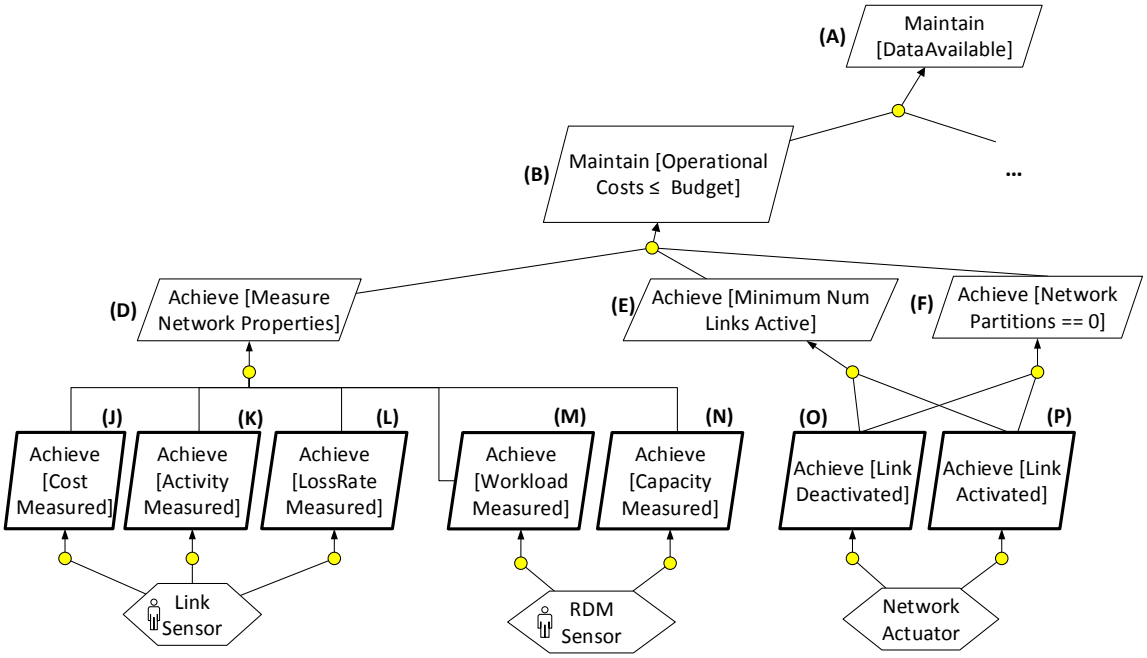
$$FF_{na} = 1.0 - \left(\frac{|adaptations|}{|faults|}\right), \tag{8.2}$$

$$Fitness\ Value = \begin{cases} \alpha_{nrg} * FF_{nrg} + \alpha_{na} * FF_{na} & \text{iff invariants true} \\ 0.0 & \text{otherwise} \end{cases} \tag{8.3}$$

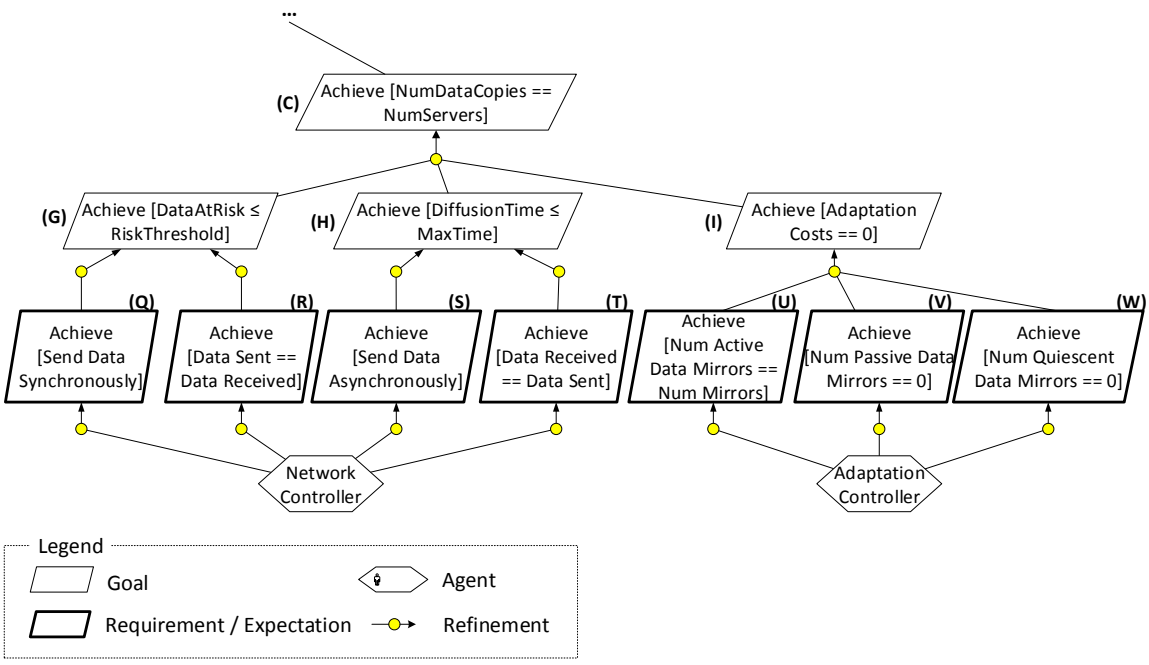
Table 8.3 presents the configuration of both the genetic algorithm and hyper-heuristic used for AutoRELAX and AutoRELAX-SAW, respectively.

Table 8.3: End-to-end AutoRELAX-SAW configuration.

Configuration Parameter	**Value**
Number of generations	50
Population size	20
Crossover rate	50.0%
Mutation rate	40.0%
SAW Sampling Rate	Every 5^{th} generation



(A) Left half of remote data mirroring goal model.



(B) Right half of remote data mirroring goal model.

Figure 8.1: KAOS goal model of the remote data mirroring application.

Figure 8.2[1] presents boxplots of the average fitness values calculated for an RDM application using AutoRELAX to configure its goal models, a manually-RELAXed goal model configured by a requirements engineer, and a Control in which no goal RELAXations were applied. As is demonstrated by these results, AutoRELAX can attain significantly higher fitness values than can be found using either the manually-RELAXed goal model or the unRELAXed goal model.

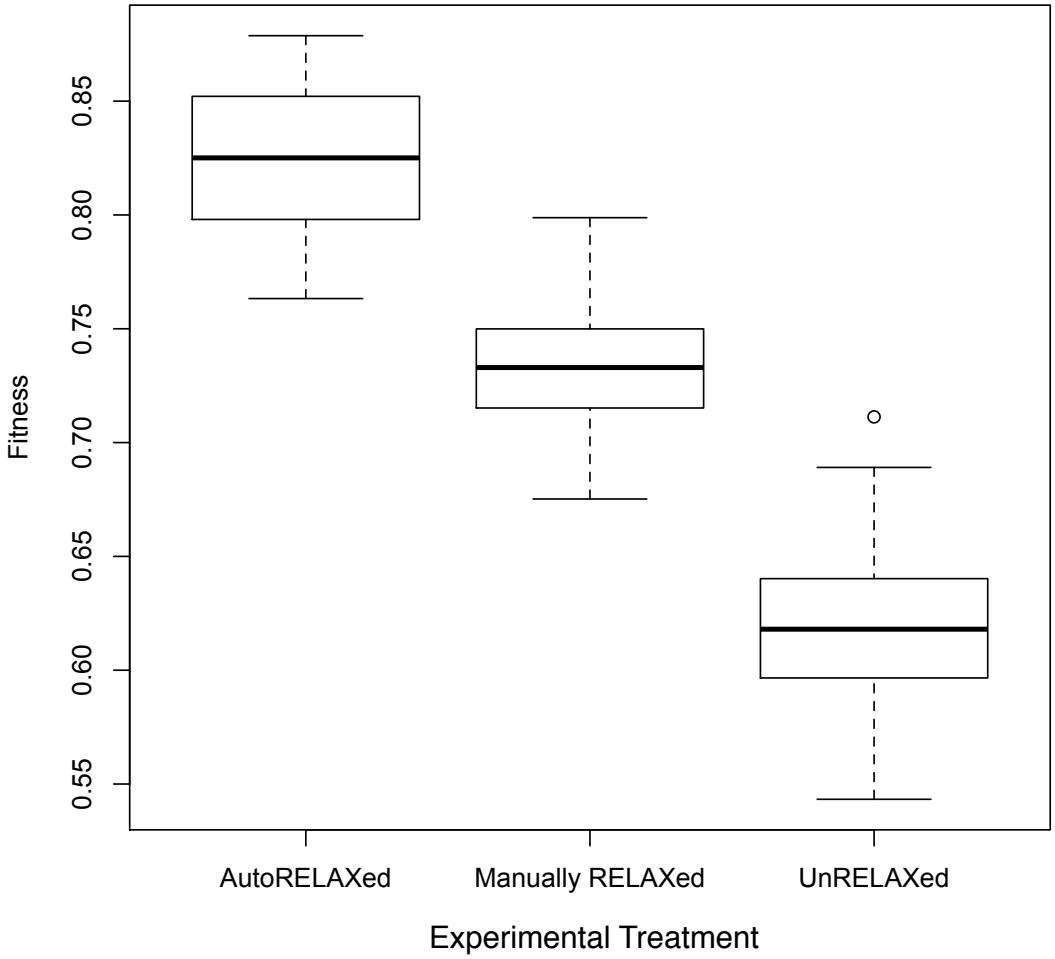


Figure 8.2: Fitness values comparison between RELAXed and unRELAXed goal models for the RDM.

Furthermore, we examined how SAW can optimize fitness by evolving the fitness subfunction weights. Figure 8.3[2] presents boxplots of the average fitness values calculated for

[1]Figure 8.2 was previously presented in Chapter 3.3.1.1.
[2]Figure 8.3 was previously presented in Chapter 3.3.1.2.

AutoRELAXed goal models and goal models evolved with AutoRELAX-SAW. As these results demonstrate, SAW can significantly increase fitness by evolving fitness sub-function weights.

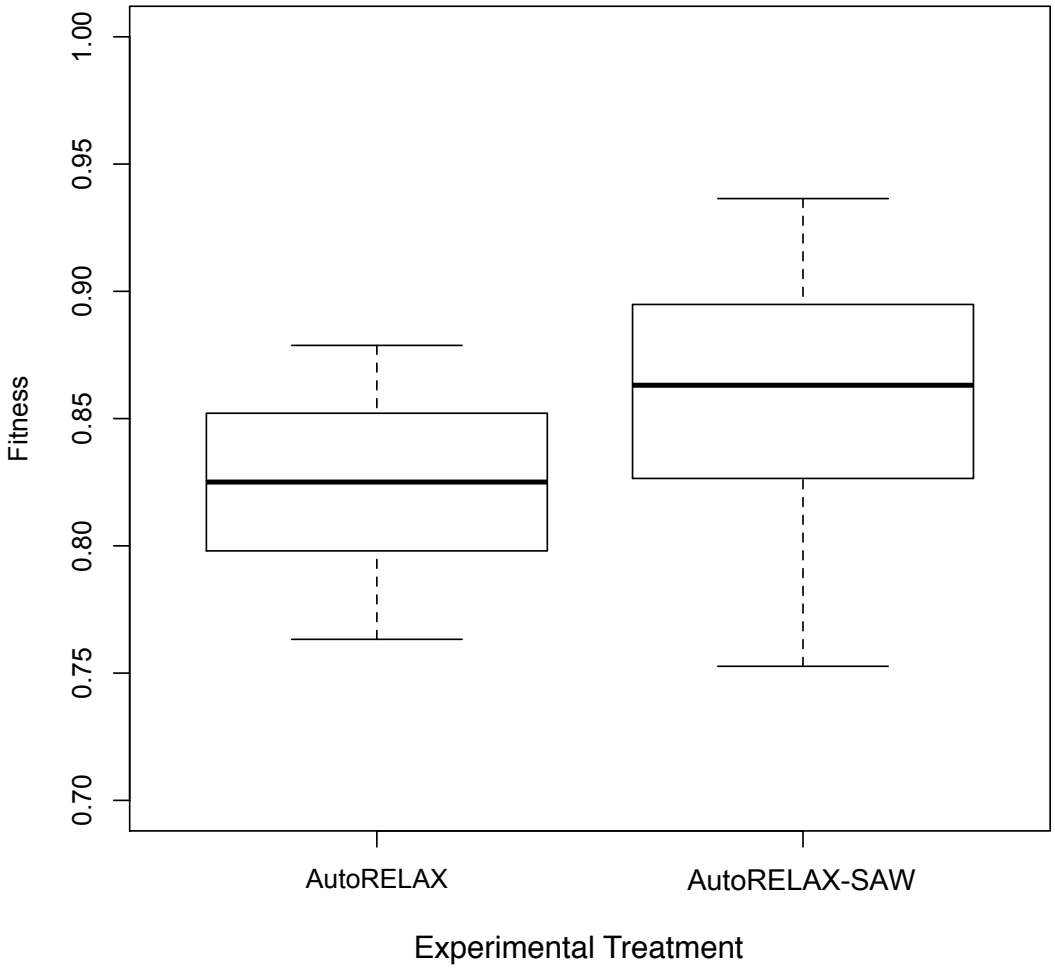


Figure 8.3: Fitness values comparison between AutoRELAXed and SAW-optimized AutoRELAXed goal models for the RDM.

Based on these results, we conclude that AutoRELAX can generate better configurations of RELAX operators than can be found by a requirements engineer. We analyzed the resulting set of data and determined that Goals (C), (F), and (T) provided the largest impact to RDM fitness. To this end, we manually introduce RELAX operators to Goals (C), (F), and (T) in all instances of the RDM application for the remainder of this end-to-end study, as RELAXation of these goals provides extra flexibility necessary for the RDM to satisfy its key objectives.

Figure 8.4 presents the RDM goal model updated to use the RELAX operators found by AutoRELAX.

8.2.2 Scalability of RDM Application

In this section, we explore the scalability of the RDM application. First, we introduce the parameters used to configure the RDM in both the normal and scaled experiments. Next, we describe our approach for comparing the normal and scaled versions of the RDM. Finally, we explore the effectiveness of AutoRELAX in the scaled RDM application.

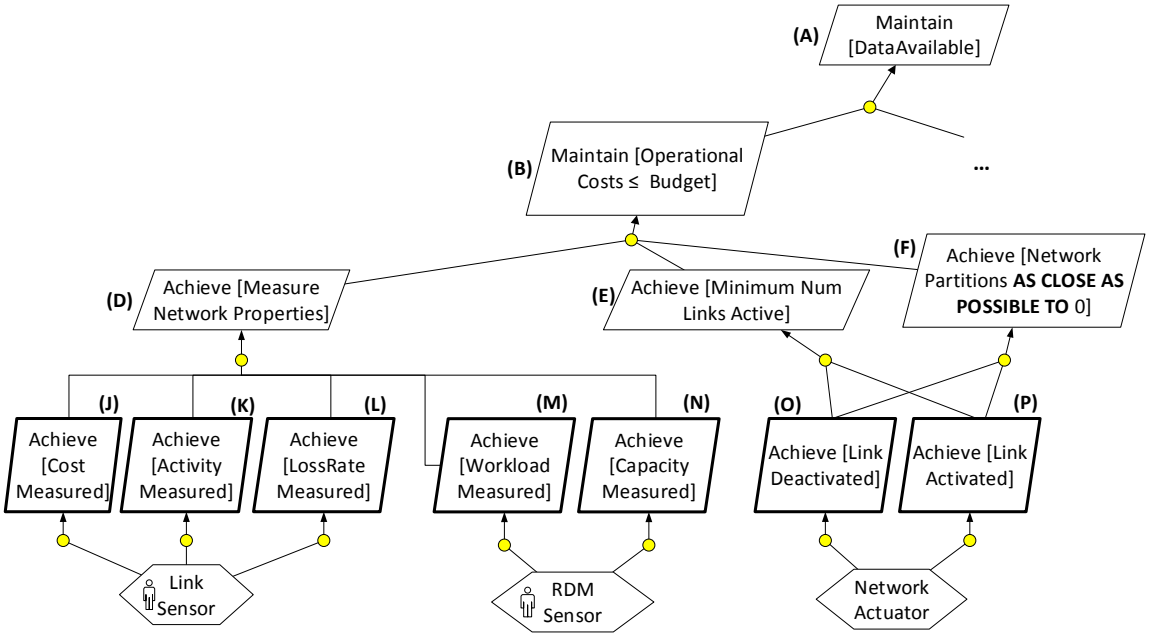
8.2.3 Scaled RDM Configuration

The RDM application accepts a configuration file that defines parameters such as the number of data mirrors, number of messages to disperse, and an operating budget. Thus far, the RDM has been configured on a relatively small scale for simulation purposes. Table 8.4 presents the relevant configuration parameters that have been increased to effectively scale the RDM application.

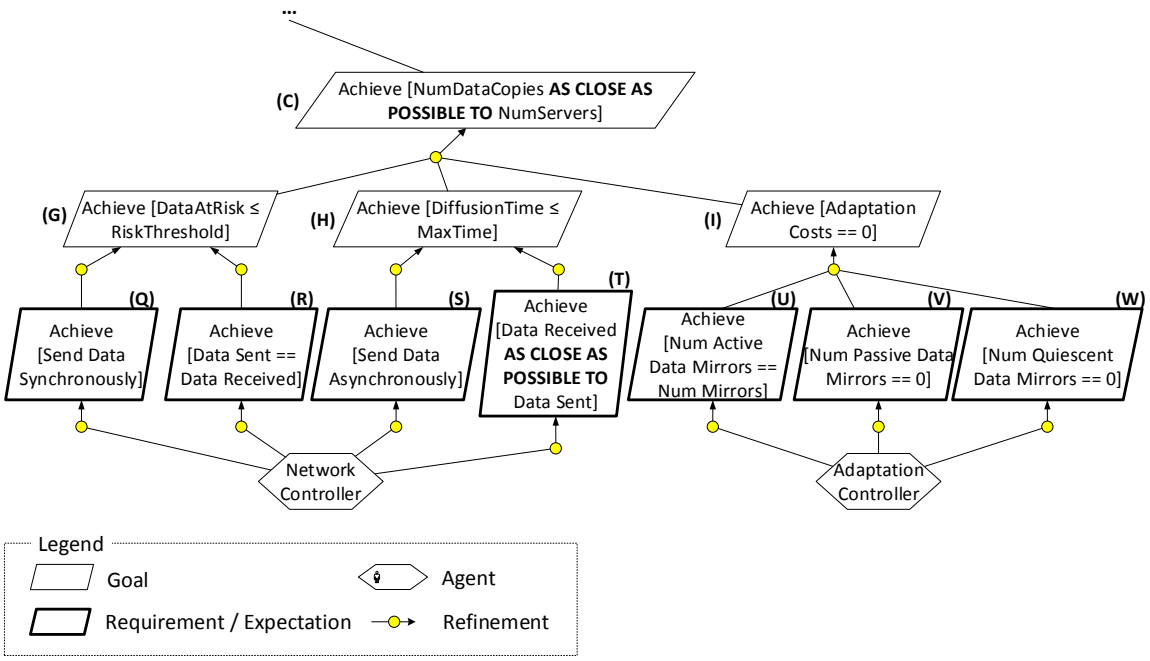
Table 8.4: Comparison of RDM Configuration Parameters.

Configuration Parameter	**Normal RDM Value**	**Scaled RDM Value**
Number of Messages	$[100, 200]$	$[800, 1000]$
Timesteps	300	2000

As such, only two parameter updates were necessary to scale the RDM application, as the existing number of data mirrors and operating budget can successfully handle the larger number of messages inserted into the network. In particular, we increased the number of messages to be distributed to effectively increase the amount of traffic handled by the RDM application. To this end, we also increased the number of timesteps available for the RDM to disseminate those messages, as extra time was required to fully replicate each message throughout the network.



(A) Left half of RELAXed remote data mirroring goal model.



(B) Right half of RELAXed remote data mirroring goal model.

Figure 8.4: RELAXed KAOS goal model of the remote data mirroring application.

8.2.4 Approach

For this study, we analyzed the scalability of the RDM application as applied to AutoRELAX. In particular, we were interested in the viability of evolving a RELAXed goal model for a network that has been scaled upward. To this end, we reuse the AutoRELAX approach introduced in Chapter 3.2.2, however the RDM was configured to use the scaled configuration parameters defined in Table 8.4. Moreover, we also reuse the RDM goal model presented in Figure 8.4. In the interest of both execution time and available storage space, AutoRELAX-SAW was not applied to this study.

8.2.5 Experimental Results

We now present experimental results from applying AutoRELAX to a scaled RDM application. We performed three experimental treatments to determine the scalability of both the RDM and AutoRELAX. For the first treatment, we applied AutoRELAX to the RDM goal model. For the second treatment, we manually applied RELAX operators to the RDM goal model. In particular, RELAX operators were manually applied to Goals (F), (G), and (H) (c.f., Figure 8.1). Lastly, the third treatment was a Control in which no goal RELAXation was performed. For each treatment, we performed 50 trials for statistical significance.

Figure 8.5 presents boxplots of the average fitness values calculated for the RDM application over each experimental treatment. These results indicate that automatically RELAXing goals can yield a higher overall fitness as compared to performing no goal RELAXation (Wilcox-Mann-Whitney U-test, $p < 0.05$). However, manually RELAXing the goal model did not yield significantly higher fitness values than the Control, indicating that a RELAXed goal model will not always yield higher fitness values when scaling the RDM application. Furthermore, we manually RELAXed other goals to determine if different configurations of RELAX operators would perform better. In each case, overall fitness did not significantly improve compared to the Control experiment.

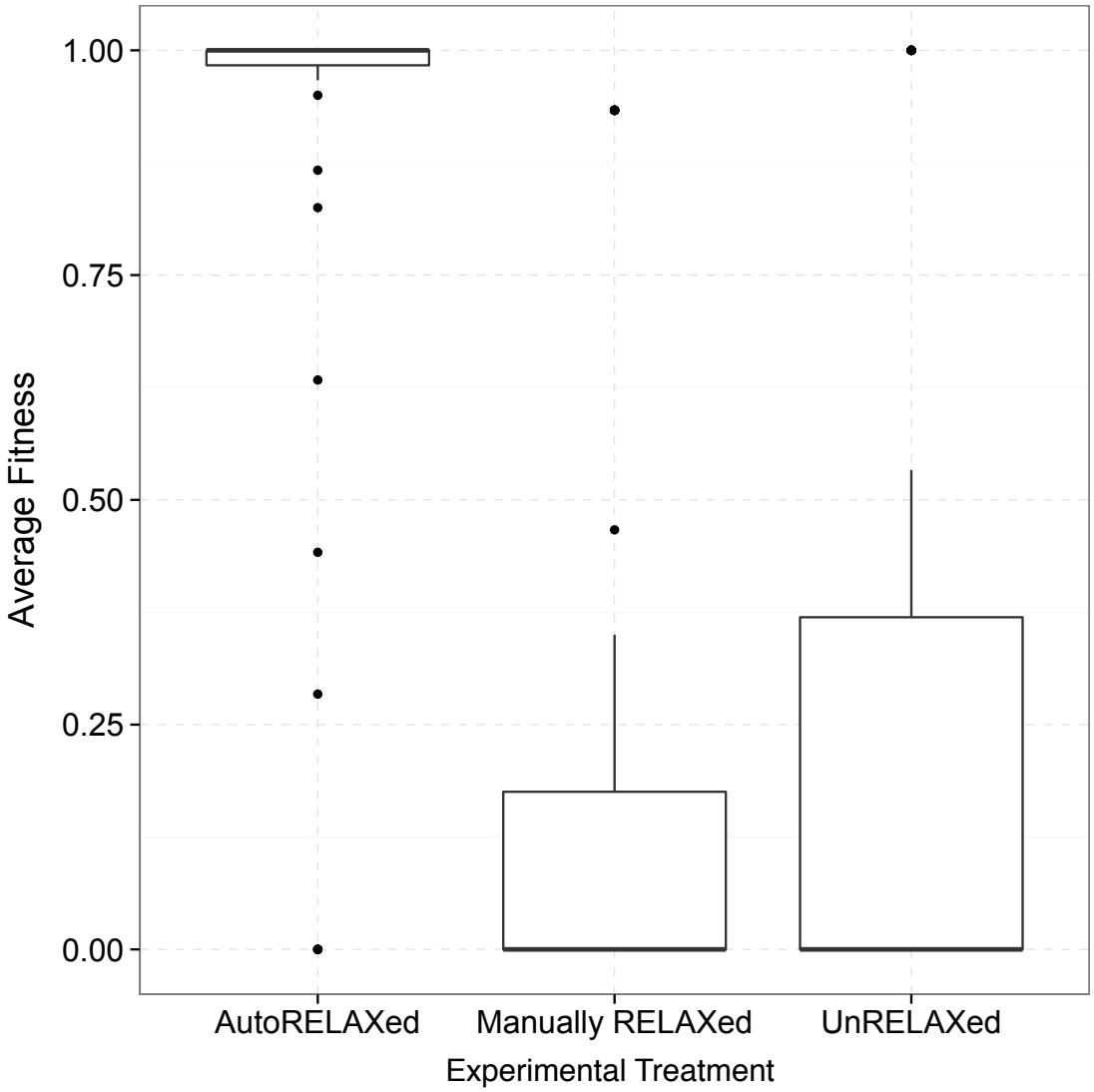


Figure 8.5: Fitness values comparison between RELAXed and unRELAXed goal models for a scaled RDM.

While scaling the RDM application is feasible, the amount of data generated was exponentially larger than the dataset generated by the normally-configured application. In particular, each experimental replicate generated a dataset several gigabytes in size and moreover required at least three days to run on a high-performance computing cluster. In total, the combined number of executed replicate trials required roughly three weeks of computing time to run. To summarize, this experiment has demonstrated the feasibility of scaling

Table 8.5: End-to-end Fenrir configuration.

Configuration Parameter	**Value**
Number of generations	15
Population size	20
Crossover rate	25.0%
Mutation rate	50.0%
Novelty threshold	10.0%

both the RDM application and AutoRELAX while presenting the overhead encountered when performing AutoRELAX in an upward-scaled environment.

8.3 Code-Based Assurance

This section presents how we used Fenrir to address assurance for the RDM's implementation to address code-based uncertainty. In particular, we examine how an RDM engineer can analyze output provided by Fenrir to identify sections of code that may require an update. The RDM engineer can then update the associated system goal model or requirements specification as necessary to reflect the required changes to the DAS implementation.

8.3.1 Fenrir Analysis

We applied RELAX operators to the goals previously identified by AutoRELAX (c.f., Figure 8.4) to determine the RDM's response to uncertainty when augmented with RELAX. In particular, we applied RELAX operators to Goals (C), (F), and (T). Following, we executed the instrumented, RELAXed RDM application over 50 trials to provide a statistically significant suite of possible execution behaviors. Table 8.5 presents the configuration of the novelty search algorithm implemented by Fenrir.

As such, we selected a subset of the resulting RDM execution traces that exhibited the largest number of errors for manual analysis. We then identified a section of code that continually triggered an error condition in the data distribution portion of the RDM codebase. In particular, the RDM was attempting to send messages via faulty data mirrors, thereby triggering an assertion error. We then rectified the problem by ensuring that a faulty data mirror is never selected for message distribution.

Figure 8.6 presents boxplots that compare the average number of errors exhibited by the original RDM codebase to the average number of errors exhibited by the updated RDM codebase. These results demonstrate that the bugfix significantly reduces the number of errors encountered throughout the simulation (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). Moreover, the introduced bugfix reduces the maximum number of errors to 1, indicating that the updated section of code was responsible for the majority of errors logged within the RDM application.

The data presented within this section, particularly in Figure 8.6, demonstrate that Fenrir enables a DAS engineer to effectively identify and rectify problems within a DAS codebase. Specifically, the novel set of system and environmental conditions generated by Fenrir provide a repository of operational contexts with which to exercise DAS behavior in uncertain situations.

8.4 Run-Time Testing-Based Assurance

This section discusses the application of Proteus and Veritas to the RDM application to provide run-time testing assurance. We first discuss the derivation of the test specification and then analyze the results from applying both Proteus and Veritas to the RDM, respectively.

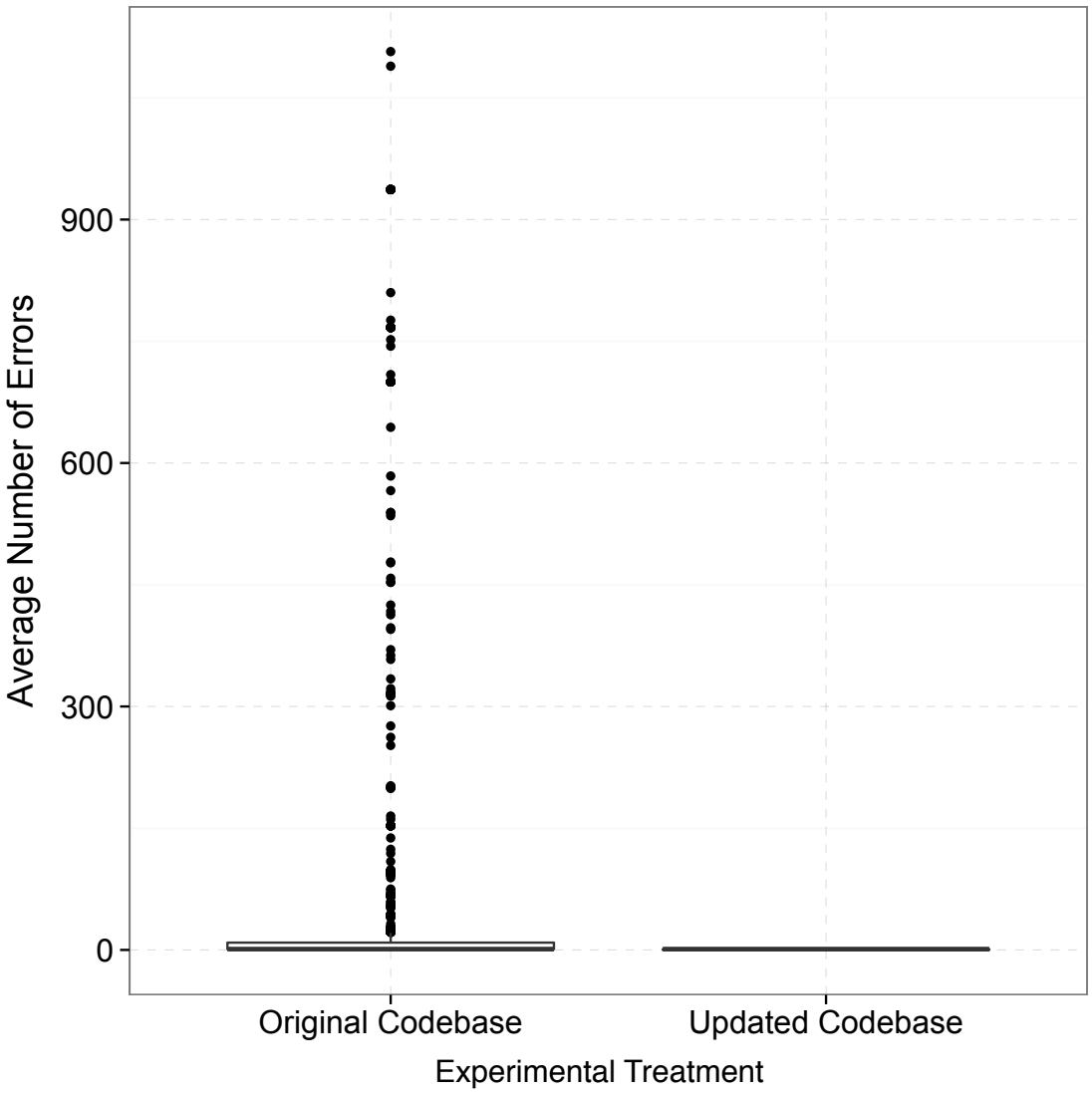


Figure 8.6: Comparison of average number of RDM errors between original and updated RDM application across 50 trials.

8.4.1 Derivation of Test Specification

To perform run-time testing on the RDM, a test specification was derived based upon its requirements. As such, we derived a representative set of test cases based on the goal model presented in Figure 8.4. Furthermore, we analyzed the RDM codebase to create extra test cases to validate system functionality. In total, 34 test cases were derived for run-time execution on the RDM application. Each test case was correlated to at least one goal in

the goal model to measure test case relevance, as described in Chapter 6.4.2. Of those 34 test cases, 7 test cases were considered invariant and therefore precluded from adaptation. Moreover, invariant test cases were re-executed each testing cycle to ensure that system constraints are continually satisfied. The remaining 27 test cases were labeled non-invariant and were therefore targets for adaptation.

To provide assurance at the run-time testing level, the RDM application was outfitted with the Proteus run time testing framework, including management of run-time testing activities and coarse-grained test plan adaptation. Furthermore, Veritas was also implemented to provide fine-grained test case adaptation capabilities. The following sections present results in which run-time testing activities, including both types of test adaptation, were performed on the RDM application.

8.4.2 Proteus Analysis

To evaluate the impact of Proteus test plan adaptation on the RDM application, we compared Proteus adaptive test plans to a manually-derived test plan. Moreover, Veritas was disabled to focus solely on the effects of Proteus test plan adaptation. The manually-derived, or Control, test plan comprised all test cases from the test specification. The test cases in the Control test plan were only executed when the conditions for test case execution, as specified by each individual test case, were met. Lastly, to provide statistical significance, we performed 50 trials of each experiment.

First, we examined the amount of executed irrelevant test cases to demonstrate how Proteus can reduce the expended effort of the testing framework. Figure 8.7 presents the cumulative amount of irrelevant test cases executed using Proteus adaptive test plans and the Control test plan, respectively. As the boxplots in Figure 8.7 demonstrate, Proteus adaptive test plans significantly reduce the amount of executed irrelevant test cases (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). This result enables us to conclude that using adaptive test plans can reduce the amount of unnecessary effort expended by the testing framework.

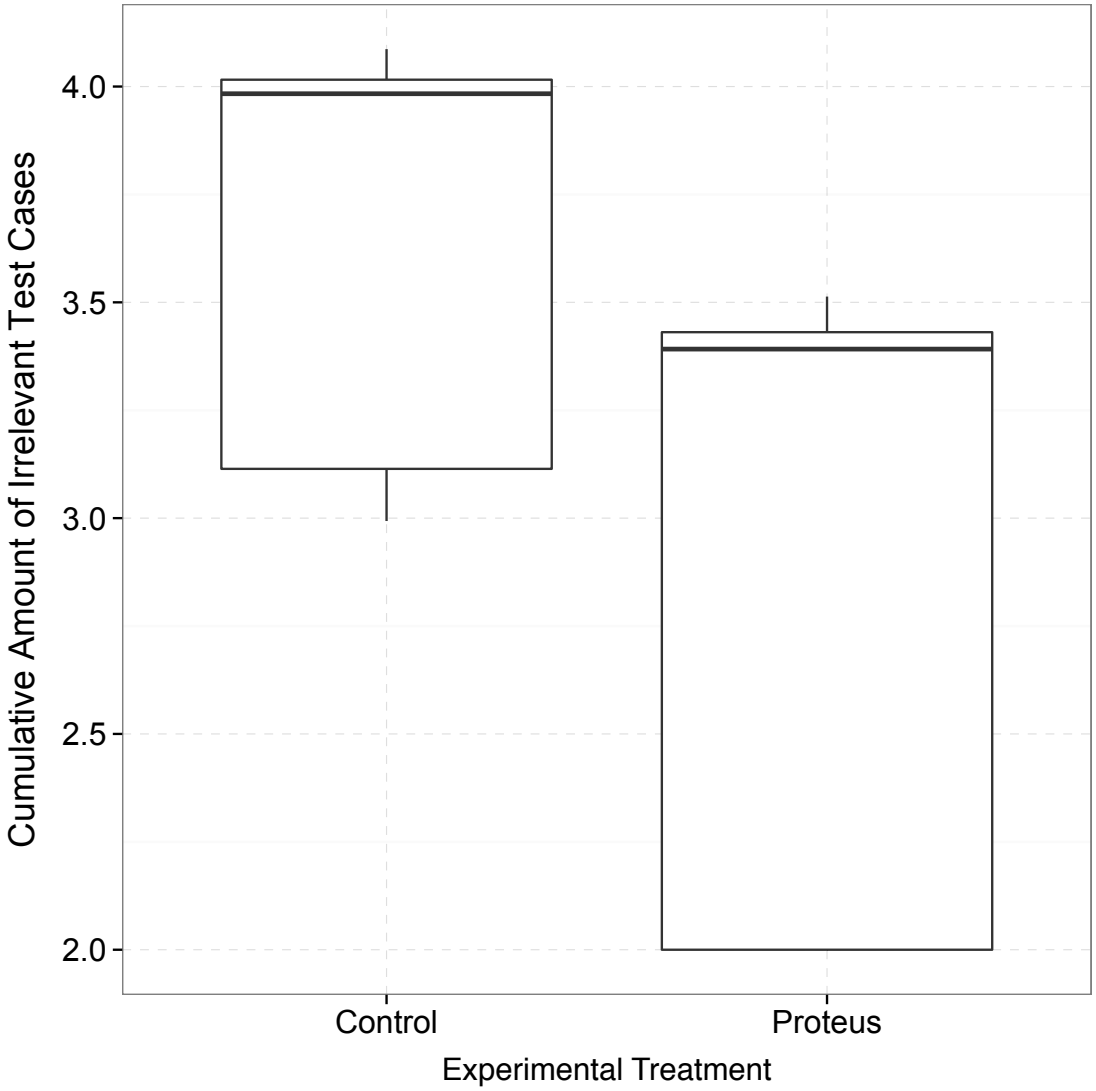


Figure 8.7: Cumulative number of irrelevant test cases executed for each experiment.

Figure 8.8 next presents the amount of *false positive* (i.e., test case passed but the correlated utility function was violated) test results found when performing run-time testing. As the results demonstrate, Proteus adaptive test plans can significantly reduce the amount of false positive test results (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), indicating that Proteus adaptive test plans can reduce the amount of executed test cases that are not relevant to current operating conditions.

Next, Figure 8.9 depicts the amount of *false negative* (i.e., test cases that fail but the correlated utility function is satisfied) test cases. Here, Proteus significantly reduces the amount

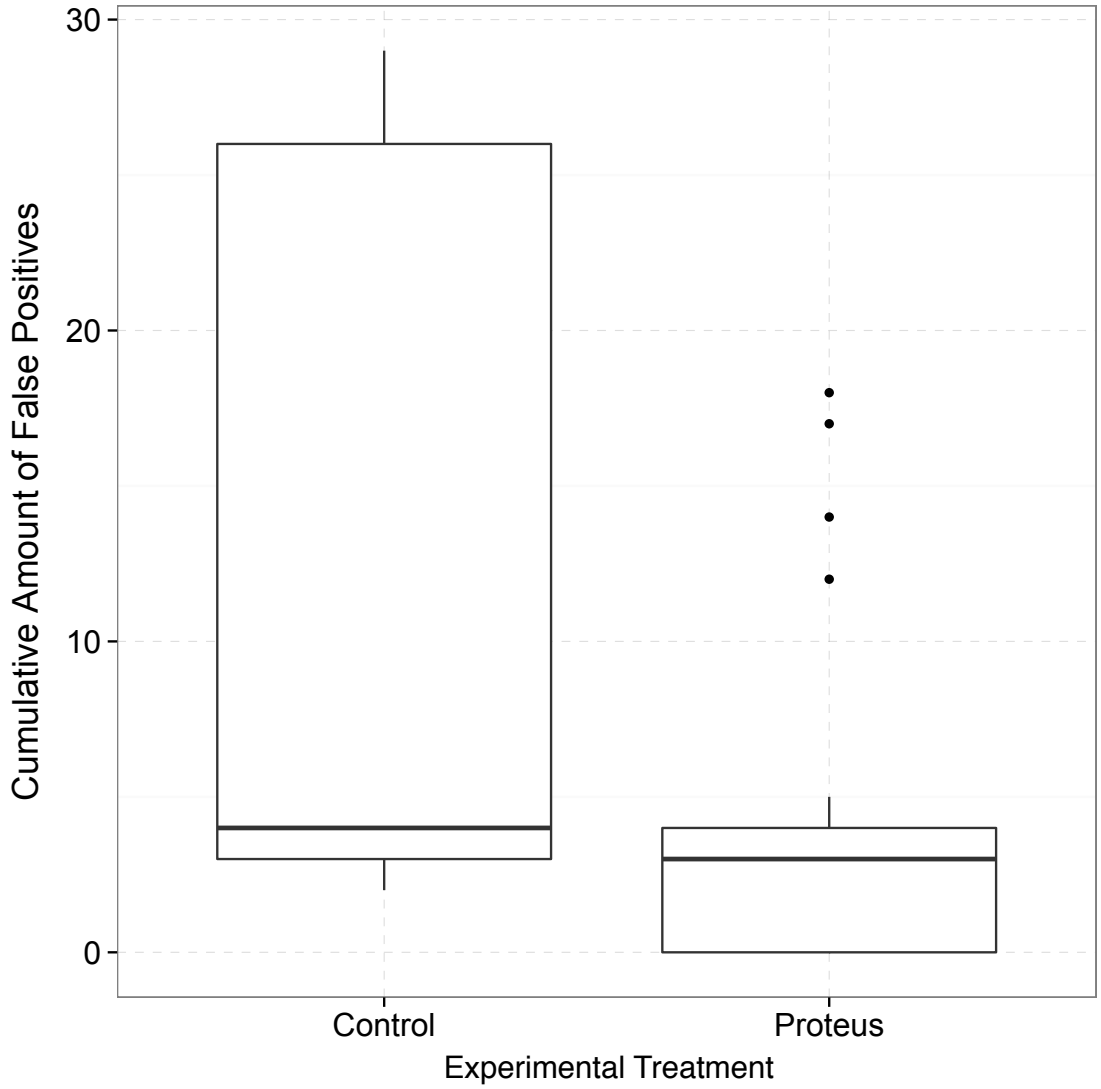


Figure 8.8: Cumulative number of false positive test cases for each experiment.

of false negative test cases (Wilcoxon-Mann-Whitney U-test, $p < 0.05$), demonstrating that adaptive test plans can reduce the overall amount of adaptations required by the testing framework, thereby reducing the overall cost of testing on the DAS.

Lastly, Figure 8.10 presents the total number of executed test cases for both Proteus and the Control. This figure demonstrates that adaptive test plans significantly reduce the amount of test case executions at run-time (Wilcoxon-Mann-Whitney U-test, $p < 0.05$). This result enables the conclusion that using Proteus adaptive test plans can again reduce

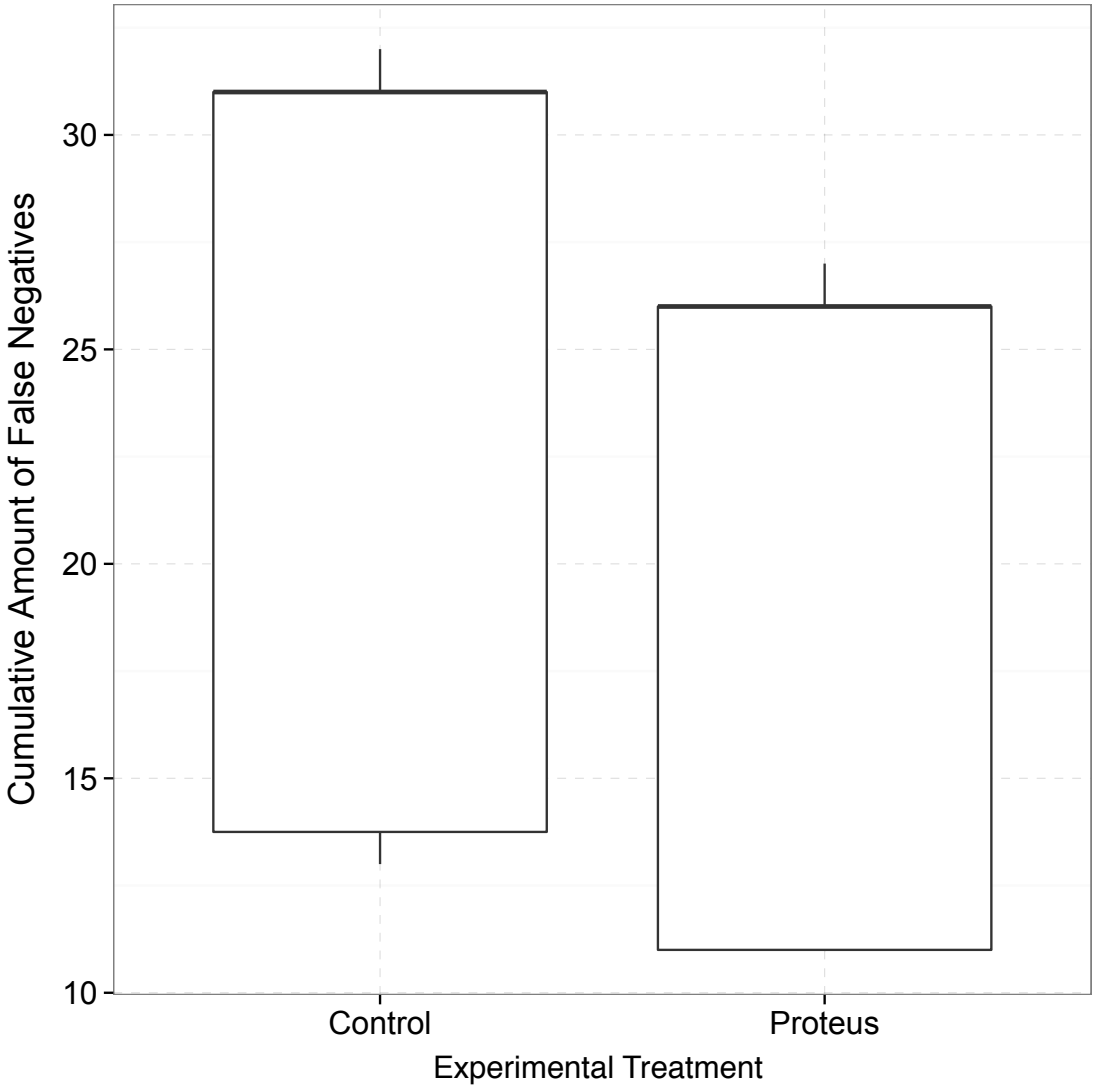


Figure 8.9: Cumulative number of false negative test cases for each experiment.

the effort required by the testing framework, as the total number of test cases executed at run time has been minimized.

8.4.3 Veritas Analysis

To evaluate the impact of fine-grained test case adaptation on the run-time testing framework we enabled Veritas and augmented the original test specification with acceptable adaptation ranges for each test case. We then compared the results to a Control in which

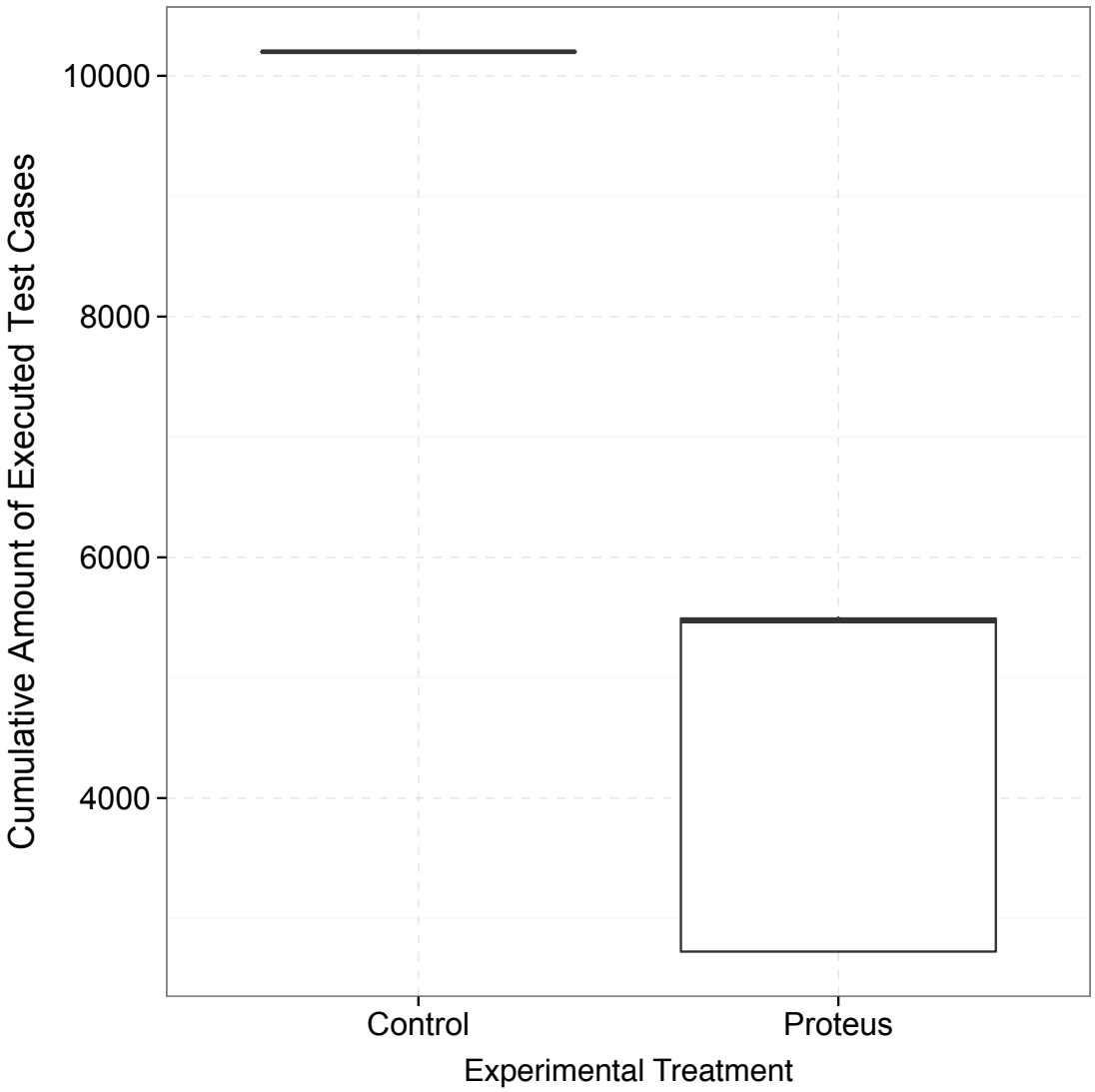


Figure 8.10: Cumulative number of executed test cases for each experiment.

test case adaptation was not performed. For this experiment, we disabled Proteus test plan adaptation to focus solely on the results provided by Veritas.

In particular, we examine the average test case fitness value through execution of the RDM. Test case fitness is considered to be the difference between the test case expected value and the value measured by the testing framework (test case fitness is formally defined in Chapter 6.4.2). Figure 8.11 presents the average test case fitness values calculated throughout execution of the RDM application. These results indicate that Veritas can adapt test case parameter values to be more relevant towards their environment, as Veritas returned

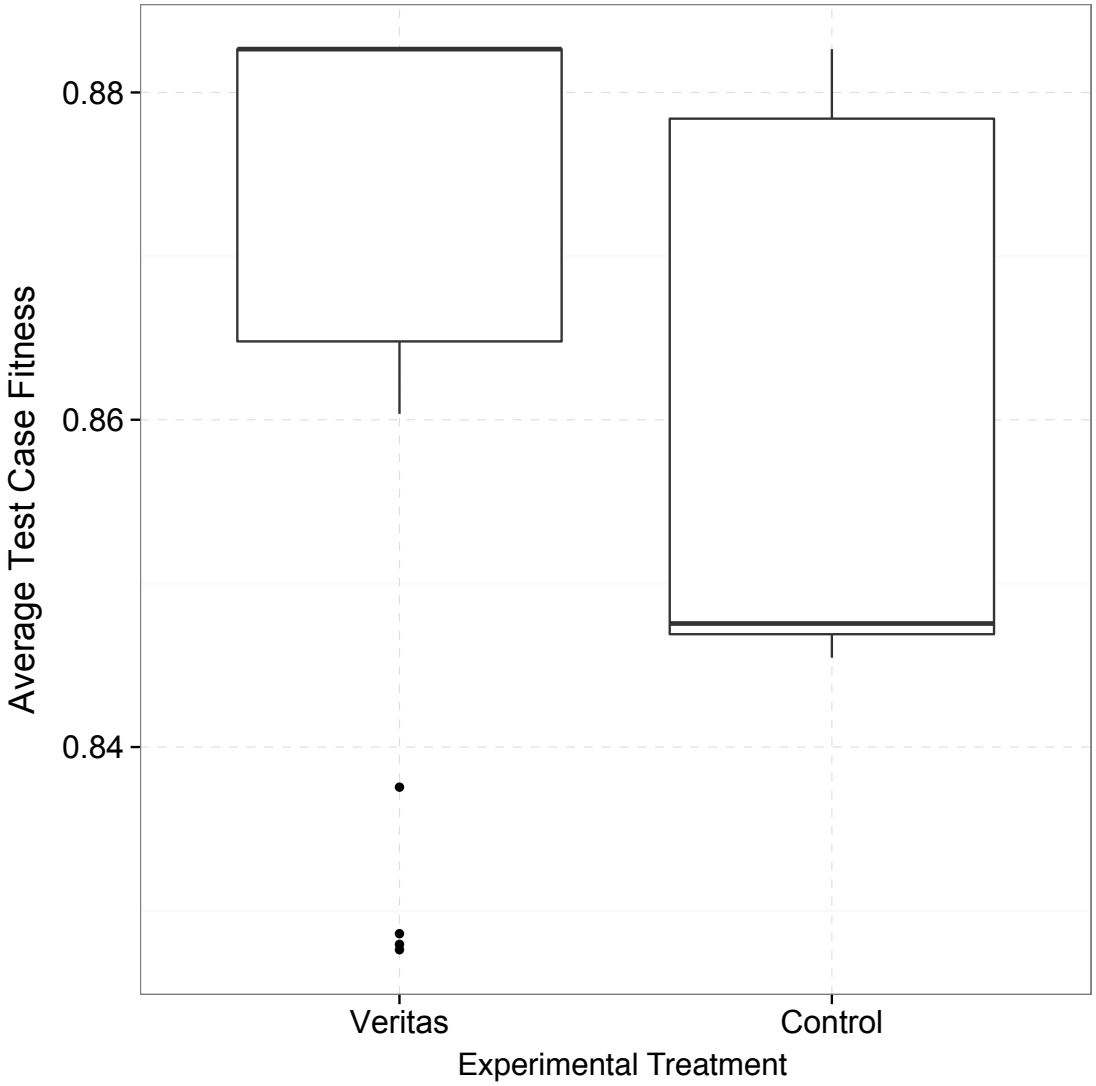


Figure 8.11: Average test case fitness values calculated for each experiment.

significantly higher fitness values during the course of execution (Wilcoxon-Mann-Whitney U-test, $p < 0.05$).

8.5 Conclusion

This chapter has discussed our investigation into applying each assurance technique presented within this dissertation sequentially to the RDM application to demonstrate the effectiveness of our techniques. First, we explored how AutoRELAX provides requirements-

based assurance by automatically RELAXing a system goal model. To this end, we discovered that introducing RELAX operators to the RDM goal model can increase overall fitness of the application, and moreover Goals (C), (F), and (T) are most amenable to RELAXation. We also explored the scalability of the RDM application. We performed the AutoRELAX experiment on an RDM application that was required to distribute an exponentially higher number of messages than was required for the normal RDM. The scaled version of the RDM successfully satisfied its requirements, however it incurred significant overhead in both execution time and memory overhead.

Next, we applied Fenrir to the RELAXed RDM application to enable implementation-based assurance. In particular, analyzing Fenrir output enabled us to discover a major bug within the RDM application that accounted for the majority of run-time errors. Lastly, we performed run-time testing on the RDM application to enable run-time assurance that the RDM is behaving in accordance with its requirements. As such, we enabled Proteus test plan adaptation and Veritas test case parameter value adaptation to provide flexibility when executing tests in uncertain environments, ensuring that test cases remained relevant throughout changing conditions.

Chapter 9

Conclusions and Future Investigations

This chapter summarizes our contributions to the field of software engineering by providing DAS run-time assurance at different levels of abstraction. We also propose future investigations to complement this body of work.

Software systems are becoming increasingly complex to enable execution in uncertain environments. In particular, software systems can adapt at run time in the face of uncertainty to continue execution while still satisfying requirements. DASs provide self-reconfiguration capabilities to change configuration states when exposed to different types of environmental conditions. As such, providing continuous assurance that a DAS is executing correctly and safely is a concern, especially as DASs take part in increasingly critical systems, such as health care infrastructure and power grid management. To this end, researchers have been exploring methods of providing run-time assurance through system monitoring and run-time testing, particularly when faced with uncertain environmental conditions.

This dissertation presented techniques for providing run-time assurance for a DAS at different levels of abstraction, particularly at design time and run time. First, we described how to provide assurance at the requirements level with AutoRELAX [42, 88] and AutoRELAX-SAW [42], two techniques that add flexibility to a system goal model and tailor fitness sub-function weights to particular environmental contexts, respectively. Next, we presented our

work in exploring how assurance can be provided during design time at the implementation level with Fenrir [45], our technique for exploring how a DAS executes in different operational contexts. Following, we outlined a guiding approach to enabling run-time testing with the MAPE-T feedback loop [44]. MAPE-T is derived from the DAS MAPE-K feedback loop, and as such, can be used in tandem with MAPE-K to provide run-time assurance. We also outlined the key challenges and enabling technologies to realize MAPE-T. Then, we introduced two techniques for providing run-time assurance with adaptive testing. Specifically, we introduced Proteus [41], a technique for test suite adaptation, and Veritas [43], a technique for adapting test case parameter values. We also investigated the feasibility of using RELAX operators to provide flexibility for run-time test case results. To demonstrate the effectiveness of our techniques, we applied Fenrir, AutoRELAX, and AutoRELAX-SAW to an application that must dynamically reconfigure an RDM application. Furthermore, we also applied AutoRELAX, AutoRELAX-SAW, Proteus, and Veritas to a cyber-physical system that must efficiently, effectively, and safely vacuum a room.

We then studied the impact that run-time testing imparts on a DAS. In particular, we examined DAS performance and behavior while run-time testing was enabled. We discovered that a DAS outfitted with a run-time testing framework requires significantly more time to execute than a DAS that does not perform run-time testing. However, our testing framework did not consume significantly extra memory, and moreover did not significantly impact the DAS from a behavioral standpoint. Therefore, we conclude that, as long as the DAS can tolerate the extra time required, run-time testing provides a considerable benefit for run-time assurance. Lastly, we performed an end-to-end investigation of each of our techniques as applied to the RDM application to demonstrate how each technique enhances assurance at different levels of abstraction.

9.1 Summary of Contributions

There were three overarching objectives for this body of research:

1. Provide a set of techniques that effectively and efficiently mitigate uncertainty by providing assurance at different levels of DAS abstraction.

2. Maximize the use of automation in designing, performing, and analyzing each technique.

3. Provide a well-defined feedback loop to enable run-time testing of a DAS.

As such, this dissertation has presented the following research contributions:

- Defined a suite of techniques that provide assurance for a DAS at its requirements level [42, 88], implementation level [45], and run-time testing level [41, 43, 44].

- Explored how search-based software engineering techniques, particularly evolutionary computation, can be used to enable assurance for a DAS at different levels of abstraction. Specifically, we investigated how a genetic algorithm and hyper-heuristic can mitigate requirements-based uncertainty [42, 88], how novelty search can mitigate implementation-based uncertainty [45], and how an online evolutionary algorithm can mitigate run-time testing-based uncertainty [43]. Moreover, we explored how automated techniques can be used to create, execute, and maintain run-time software test suites [41].

- Proposed and instantiated a well-defined DAS testing feedback loop for execution at run time [44], and moreover provided an implementation of the different aspects of the feedback loop [41, 43].

9.2 Future Investigations

The investigations described within this dissertation demonstrate approaches for providing software assurance at different levels of abstraction to address uncertainty. Based on the presented results, we now describe complementary investigations that can be pursued to extend our line of work in automatically enabling DAS assurance. We next describe each investigation in turn.

9.2.1 Exploration of Different Evolutionary Computation Techniques

Evolutionary computation is used as our main approach for automatically searching a solution space for an optimal solution. AutoRELAX uses a genetic algorithm to search for optimal configurations of RELAXed goal models, and moreover uses a hyper-heuristic, SAW, to search for an optimal combination of fitness sub-function weights. Fenrir leverages novelty search to find a solution set comprising the most different solutions within the search space. Lastly, Veritas implements an online evolutionary algorithm, the (1+1)-ONLINE EA, to find optimal combinations of test case parameter values.

We envision that exploration of other search-based techniques can provide an interesting comparison point to this body of work to determine the effect that other approaches may have on providing assurance, as other evolutionary approaches may uncover unexplored areas of a solution space that our techniques may not have found. In particular, approaches such as multi-objective optimization, simulated annealing, or particle swarm optimization can be used to compare to the results found by the evolutionary approaches we have used. Moreover, exploration into the extension of these techniques to the run-time domain would also be of great benefit to the field of software engineering, as few run-time evolutionary algorithms currently exist, and many advanced applications (e.g., power grid systems, autonomous vehicles, etc.) may benefit from an available suite of run-time, search-based heuristics.

Furthermore, different types of distance metrics can be explored for use in Fenrir to determine if a different approach can uncover solutions that provide a richer set of expressed DAS behaviors. For instance, we use the Manhattan distance metric, whereas others, such as Euclidean distance, could be explored in comparison. Lastly, probabilistic methods [4, 15, 38, 84] may also be used to explore the search space.

9.2.2 Interfacing with the DAS MAPE-K Loop

We have provided sample instantiations of the MAPE-T feedback loop for run-time testing. We envision that future investigations can more tightly link the DAS MAPE-K loop with the MAPE-T loop to enable fine-grained control over DAS reconfiguration capabilities. In particular, test cases that have been correlated with a utility function could be used to validate different aspects of the utility function, enabling a fine-grained analysis of each utility function. As such, the DAS reconfiguration engine could consume this information and perform a *targeted reconfiguration*, thereby reducing the overall impact of reconfiguration to the DAS and potentially reducing unexpected side effects of a total reconfiguration.

9.2.3 Hardware Realization of the MAPE-T Loop

To validate the techniques presented in this dissertation, we have simulated both a high-level networking application and a low-level onboard control for an embedded system. However, given the reality gap that naturally exists between simulation and implementation, we envision that a hardware-based realization of a DAS with which to implement the MAPE-T loop can provide an even richer testbed for experimentation in both validating our simulated results and enabling further research into the MAPE-T loop.

9.2.4 Incorporation of MAS Architecture

Lastly, unifying the DAS and MAS architectures to create a new type of adaptive system could be a particularly interesting path of research. Specifically, the use of agents in the DAS architecture could provide a significant advantage in that processor- or memory-intensive tasks could be offloaded to a separate agent, thereby freeing the DAS to fully focus on monitoring its environment and performing self-reconfigurations. Given the prior work we have previously described in agent-based testing [78], we envision that using agents to perform software testing can significantly reduce the impact of run-time testing on the DAS architecture.

APPENDIX

Appendix

Test Specifications

This appendix presents the test specifications used for both the smart vacuum system (SVS) and remote data mirroring (RDM) case studies within this dissertation. For each case study, we first present the full test specification, including a description of each test case, expected value(s), and the safety tolerances used to enable safe adaptation of test case parameter values. We then provide the traceability links between tests and requirements, particularly, how each test case is correlated to a particular goal within the corresponding goal model.

Smart Vacuum System Test Specification

This section overviews the test specification used for the SVS case study within this dissertation. Particularly, Table A.1 presents the test specification that is used as a basis for run-time testing and adaptation. For each test case, we present the test case identifier (ID), the test case type (i.e., invariant or non-invariant), a description, the expected value(s), and the safety tolerance used for adaptation. Furthermore, if the safety tolerance is listed as N/A, then the test case is considered non-invariant, however cannot be physically adapted as no defined flexibility was provided. Lastly, while some test cases appear to be duplicates,

they may measure different variables, execute under different circumstances, or provide a slightly different safety tolerance to explore different test case adaptations.

Table A.1: Smart vacuum system test specification.

TC1	
Test case type	*Non-invariant*
Description	Room sensor can detect the amount of dirt cleaned by the SVS.
Expected value	$RoomSensorActive = True$
Safety tolerance	N/A
TC2	
Test case type	*Non-invariant*
Description	SVS executes a power conservation plan to maximize the amount of time it spends vacuuming the room.
Expected value	$PowerModifier \in [0.0, 1.0)$
Safety tolerance	N/A
TC3	
Test case type	*Non-invariant*
Description	SVS executes a path plan to maximize the amount of coverage while vacuuming the room.
Expected value	$PathPlan \in \{RANDOM, SPIRAL, STRAIGHT\}$
Safety tolerance	N/A
TC4	
Test case type	*Non-invariant*
Description	SVS internal sensor can detect battery usage and percentage of room covered.

Continued on next page

Expected value	$InternalSensorActive = True$
Safety tolerance	N/A

TC5	
Test case type	*Non-invariant*
Description	SVS must be in motion.
Expected value	$Speed > 0.0$
Safety tolerance	$Speed \geq 0.0$

TC6	
Test case type	*Non-invariant*
Description	SVS suction must be active.
Expected value	$SuctionActive = True$
Safety tolerance	N/A

TC7	
Test case type	*Invariant*
Description	Bumper sensor tested by internal health monitor SHALL always be healthy.
Expected value	$BumperSensor.Healthy = True$
Safety tolerance	N/A

TC8	
Test case type	*Invariant*
Description	No large dirt particles SHALL be inside vacuum at any point in time.
Expected value	$NoLargeDirtDetected = True$

Continued on next page

Safety tolerance	N/A
TC9	
Test case type	*Invariant*
Description	SVS SHALL never fall off cliff.
Expected value	$FallDetected \ = \ False$
Safety tolerance	N/A
TC10	
Test case type	*Invariant*
Description	SVS SHALL never collide with non-collideable objects.
Expected value	$InvalidCollision \ = \ False$
Safety tolerance	N/A
TC11	
Test case type	*Invariant*
Description	SVS SHALL never collide with liquid objects.
Expected value	$InvalidCollision \ = \ False$
Safety tolerance	N/A
TC12	
Test case type	*Non-invariant*
Description	If OBJECT detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToObject \ = \ 0.5m$
Safety tolerance	$\pm 0.2m$
TC13	

Continued on next page

Test case type	*Non-invariant*
Description	If LIQUID detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToLiquid = 0.5m$
Safety tolerance	$\pm 0.2m$

TC14	
Test case type	*Non-invariant*
Description	If CLIFF detected by SVS cliff sensor, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToCliff = 0.5m$
Safety tolerance	$\pm 0.2m$

TC15	
Test case type	*Non-invariant*
Description	If LARGE DIRT PARTICLE detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToLargeDirtParticle = 0.5m$
Safety tolerance	$\pm 0.2m$

TC16	
Test case type	*Invariant*
Description	Wheel sensor tested by internal health monitor SHALL always be healthy.
Expected value	$WheelSensor.Healthy = True$
Safety tolerance	N/A

Continued on next page

TC17	
Test case type	*Non-invariant*
Description	If LEFT BUMPER array triggered, SVS SHALL reorient by $\pi/4.0$ radians clockwise.
Expected value	$CurrentPathAngle = CurrentPathAngle + (\pi/4.0)rad$
Safety tolerance	$\pm 1.0rad$

TC18	
Test case type	*Non-invariant*
Description	If RIGHT BUMPER array triggered, SVS SHALL reorient by $\pi/4.0$ radians counter-clockwise.
Expected value	$CurrentPathAngle = CurrentPathAngle - (\pi/4.0)rad$
Safety tolerance	$\pm 1.0rad$

TC19	
Test case type	*Non-invariant*
Description	If CENTER BUMPER array triggered, SVS SHALL reorient by $\pi/4.0$ radians in the preferred (configured) direction.
Expected value	$CurrentPathAngle = CurrentPathAngle \pm (\pi/4.0)rad$
Safety tolerance	$\pm 1.0rad$

TC20	
Test case type	*Non-invariant*
Description	If BatteryPower falls below 50%, SVS reduces power to either SUCTION or MOTOR.
Expected value	$BatteryPower \leq 50\%$

Continued on next page

Expected value	$Velocity \geq 0.0$
Safety tolerance	N/A
TC21	
Test case type	*Non-invariant*
Description	If BatteryPower falls below 25%, SVS reduces power to either SUCTION or MOTOR, whichever is still operating at FULL POWER.
Expected value	$BatteryPower \leq 25\%$
Safety tolerance	N/A
TC22	
Test case type	*Non-invariant*
Description	If BatteryPower falls below 10%, SVS reduces power further to SUCTION and MOTOR.
Expected value	$BatteryPower \leq 10\%$
Safety tolerance	N/A
TC23	
Test case type	*Invariant*
Description	Suction sensor tested by internal health monitor SHALL always be healthy.
Expected value	$SuctionSensor.Healthy = True$
Safety tolerance	N/A
TC24	
Test case type	*Invariant*

Continued on next page

Table A.1 (cont'd)

Description	SVS suction must not be impeded by large objects or liquids.
Expected value	$InvalidCollision = False$
Safety tolerance	N/A
TC25	
Test case type	*Non-invariant*
Description	If RANDOM selected, SVS SHALL execute RANDOM path plan for 10 seconds.
Expected value	$PathPlanTimer = 10sec$
Safety tolerance	$\pm 5.0sec$
TC26	
Test case type	*Non-invariant*
Description	If STRAIGHT selected, SVS SHALL execute STRAIGHT path plan for 10 seconds.
Expected value	$PathPlanTimer = 10sec$
Safety tolerance	$\pm 5.0sec$
TC27	
Test case type	*Non-invariant*
Description	SVS continually provides forward motion while providing suction.
Expected value	$Velocity \geq 0.0$
Safety tolerance	N/A
TC28	
Test case type	*Invariant*

Continued on next page

Description	If internal sensor or object sensor has failed, then SVS must disable the wheel motors.
Expected value	$WheelMotors = DISABLED$
Safety tolerance	N/A

TC29	
Test case type	*Invariant*
Description	If large dirt particle is detected internally, then SVS SHALL shut off all power.
Expected value	$Power = OFF$
Safety tolerance	N/A

TC30	
Test case type	*Invariant*
Description	If liquid detected internally, then SVS SHALL shut off all power.
Expected value	$Power = OFF$
Safety tolerance	N/A

TC31	
Test case type	*Invariant*
Description	If collision with non-collideable object, SVS SHALL shut off all power.
Expected value	$Power = OFF$
Safety tolerance	N/A

TC32	
Test case type	*Invariant*

Continued on next page

Description	If SVS fell off cliff, then SVS SHALL shut off all power.
Expected value	$Power = OFF$
Safety tolerance	N/A

TC33	
Test case type	*Non-invariant*
Description	If OBJECT detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToObject = 1.0m$
Safety tolerance	$\pm 0.2m$

TC34	
Test case type	*Non-invariant*
Description	If LIQUID detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToLiquid = 0.5m$
Safety tolerance	$\pm 0.2m$

TC35	
Test case type	*Non-invariant*
Description	If CLIFF detected by SVS cliff sensor, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistancetoCliff = 0.5m$
Safety tolerance	$\pm 0.2m$

TC36	
Test case type	*Non-invariant*

Continued on next page

Description	If LARGE DIRT PARTICLE detected within 0.5m of SVS, SVS SHALL execute BACKUP procedure for 2 seconds.
Expected value	$DistanceToObject \ = \ 0.5m$
Safety tolerance	$\pm 0.2m$

TC37	
Test case type	*Non-invariant*
Description	If ROOM SENSOR cannot measure amount of dirt in room, SVS controller SHALL select a random path plan.
Expected value	$RoomSensor.Healthy \ = \ True$
Safety tolerance	N/A

TC38	
Test case type	*Non-invariant*
Description	SVS is moving at the prescribed reduced speed.
Expected value	$VelocityModifier \ \in \ (0.0, 1.0)$
Safety tolerance	$[0.0, 1.0]$

TC39	
Test case type	*Non-invariant*
Description	SVS has at least 25% battery power remaining.
Expected value	$BatteryPower \ \geq \ 0.25$
Safety tolerance	$[0.05, 1.0]$

TC40	
Test case type	*Non-invariant*
Description	SVS is moving at the initially prescribed speed.

Continued on next page

Expected value	$VelocityModifier = 1.0$
Safety tolerance	$[0.0, 1.0]$
TC41	
Test case type	*Non-invariant*
Description	SVS has at least 75% battery power remaining.
Expected value	$BatteryPower \geq 0.75$
Safety tolerance	$[0.05, 1.0]$
TC42	
Test case type	*Non-invariant*
Description	SVS has successfully entered a REDUCED SUCTION POWER mode.
Expected value	$SuctionModifier \in (0.0, 1.0)$
Safety tolerance	$[0.0, 1.0]$
TC43	
Test case type	*Non-invariant*
Description	SVS provides suction power of at least the base measure and less than the power provided in NORMAL SUCTION POWER mode.
Expected value	$SuctionPower \geq 0.25$ AND $SuctionPower < 1.0$
Safety tolerance	N/A
TC44	
Test case type	*Non-invariant*

Continued on next page

Description	SVS continually moves in a forward motion while providing reduced suction power.
Expected value	$VelocityModifier \geq 0.25$ AND $SuctionPower \in (0.0, 1.0)$
Safety tolerance	N/A

TC45	
Test case type	*Non-invariant*
Description	SVS has not entered a REDUCED SUCTION POWER mode.
Expected value	$SuctionPower$ *is always* 1.0
Safety tolerance	$[0.0, 1.0]$

TC46	
Test case type	*Non-invariant*
Description	SVS provides suction power of the prescribed base measure.
Expected value	$SuctionPower = 0.25$
Safety tolerance	$[0.0, 1.0]$

TC47	
Test case type	*Non-invariant*
Description	SVS continually provides forward motion while providing base suction power.
Expected value	$VelocityModifier \geq 0.25$ AND $SuctionPower \geq 0.25$
Safety tolerance	$[0.0, 1.0]$

TC48	
Test case type	*Non-invariant*

Continued on next page

Description	If RANDOM selected, SVS SHALL execute RANDOM path plan for 10 seconds.
Expected value	$PathTimer = 10.0sec$
Safety tolerance	$\pm 5.0sec$

TC49	
Test case type	*Non-invariant*
Description	If RANDOM selected, SVS SHALL select a random angle on $[0.0, \pi]rad$ every 5 seconds.
Expected value	$PathTimer = 10.0sec$
Safety tolerance	$\pm 5.0sec$

TC50	
Test case type	*Non-invariant*
Description	If STRAIGHT selected, SVS SHALL execute STRAIGHT path plan for 10 seconds.
Expected value	$PathTimer = 10.0sec$
Safety tolerance	$\pm 5.0sec$

TC51	
Test case type	*Non-invariant*
Description	If STRAIGHT selected and collision occurs, SVS will select a new angle $(\pi/4.0)rad$ from the prior angle.
Expected value	$PathAngle \pm (\pi/4.0)rad$
Safety tolerance	N/A

TC52

Continued on next page

Table A.1 (cont'd)

Test case type	*Non-invariant*
Description	Wheel motors can be actuated to specified velocity.
Expected value	$VelocityLeft \geq SPECIFIED\ VELOCITY$ AND $VelocityRight \geq SPECIFIED\ VELOCITY$
Safety tolerance	N/A

TC53	
Test case type	*Non-invariant*
Description	SVS moves in a forward motion while following SPIRAL path plan.
Expected value	$VelocityLeft \geq 0.0$ AND $VelocityRight \geq 0.0$
Safety tolerance	N/A

TC54	
Test case type	*Non-invariant*
Description	SVS executes the spiral path plan for 20 seconds.
Expected value	$PathTimer = 20.0sec$
Safety tolerance	$\pm 5.0sec$

TC55	
Test case type	*Non-invariant*
Description	SVS continually provides at least the base measure of suction power while executing spiral path plan.
Expected value	$SuctionPower \geq 0.25$
Safety tolerance	N/A

TC56

Continued on next page

Test case type	*Non-invariant*
Description	If SPIRAL selected, SVS SHALL execute SPIRAL path plan for 20 seconds.
Expected value	$PathPlan = SPIRAL$ AND $PathTime = 20sec$
Safety tolerance	$\pm 5.0sec$

TC57	
Test case type	*Non-invariant*
Description	Object sensor tested by internal health monitor SHALL always be healthy.
Expected value	$ObjectSensor.Healthy$ *is never* $False$
Safety tolerance	N/A

TC58	
Test case type	*Non-invariant*
Description	Cliff sensor tested by internal health monitor SHALL always be healthy.
Expected value	$CliffSensor.Healthy$ *is never* $False$
Safety tolerance	N/A

TC59	
Test case type	*Non-invariant*
Description	Internal sensor tested by internal health monitor SHALL always be healthy.
Expected value	$InternalSensor.Healthy$ *is never* $False$
Safety tolerance	N/A

Continued on next page

TC60	
Test case type	*Invariant*
Description	No large dirt particles SHALL be inside vacuum at any point in time.
Expected value	$NoLargeDirtParticles\ is\ never\ False$
Safety tolerance	N/A

TC61	
Test case type	*Non-invariant*
Description	SPIRAL path plan is executed for 20 seconds.
Expected value	$PathTimer\ =\ 20.0sec$
Safety tolerance	$\pm 5.0sec$

TC62	
Test case type	*Non-invariant*
Description	RANDOM path plan is executed for 10 seconds.
Expected value	$PathTimer\ =\ 10.0sec$
Safety tolerance	$\pm 5.0sec$

TC63	
Test case type	*Non-invariant*
Description	STRAIGHT path plan is executed for 10 seconds.
Expected value	$PathTimer\ =\ 10.0sec$
Safety tolerance	$\pm 5.0sec$

TC64	
Test case type	*Non-invariant*

Continued on next page

Description	SVS selects a new angle on $[0.0, \pi]rad$ every 5.0 seconds.
Expected value	$PathAngle \in [0.0, \pi]rad$
Safety tolerance	$[-(\pi/2.0), \pi + (\pi * 2.0)]rad$
TC65	
Test case type	*Non-invariant*
Description	Upon notification of a left bumper collision, the SVS reorients clockwise by $(\pi/4.0)rad$.
Expected value	$PathAngle + (\pi/4.0)rad$
Safety tolerance	$[(\pi/4.0) - 1.0, (\pi/4.0) + 1.0]rad$
TC66	
Test case type	*Non-invariant*
Description	Upon notification of a right bumper collision, the SVS reorients counter-clockwise by $(\pi/4.0)rad$.
Expected value	$PathAngle - (\pi/4.0)rad$
Safety tolerance	$[-(\pi/4.0) - 1.0, -(\pi/4.0) + 1.0]rad$
TC67	
Test case type	*Non-invariant*
Description	Upon notification of a center bumper collision, the SVS reorients by $(\pi/4.0)rad$ towards the configured direction preference.
Expected value	$PathAngle \pm (\pi/4.0)rad$
Safety tolerance	$[-(\pi/4.0) - 1.0, (\pi/4.0) + 1.0]rad$
TC68	
Test case type	*Non-invariant*

Continued on next page

Description	The SVS executes a backup procedure for 2.0sec upon detection of an object within 0.5m.
Expected value	$DistanceToObject \ = \ 0.5m$
Safety tolerance	$[0.3, 0.7]m$
TC69	
Test case type	*Non-invariant*
Description	The SVS executes a backup procedure for 2.0sec upon detection of a liquid spill within 0.5m.
Expected value	$DistanceToLiquid \ = \ 0.5m$
Safety tolerance	$[0.3, 0.7]m$
TC70	
Test case type	*Non-invariant*
Description	The SVS executes a backup procedure for 2.0sec upon detection of a downward step within 0.5m.
Expected value	$DistanceToCliff \ = \ 0.5m$
Safety tolerance	$[0.3, 0.7]m$
TC71	
Test case type	*Non-invariant*
Description	The SVS executes a backup procedure for 2.0sec upon detection of a large object within 0.5m.
Expected value	$DistanceToObject \ = \ 0.5m$
Safety tolerance	$[0.3, 0.7]m$
TC72	

Continued on next page

Table A.1 (cont'd)

Test case type	*Non-invariant*
Description	Room sensor can measure the amount of dirt within the room.
Expected value	$RoomSensor.Active = True$
Safety tolerance	N/A

Smart Vacuum System Goal Correlation

This section provides an overview of how each test case is correlated to a particular goal, thereby providing a method for validating test results at run time. We now reproduce the SVS goal model previously presented in Figure 2.2 to simplify readability.

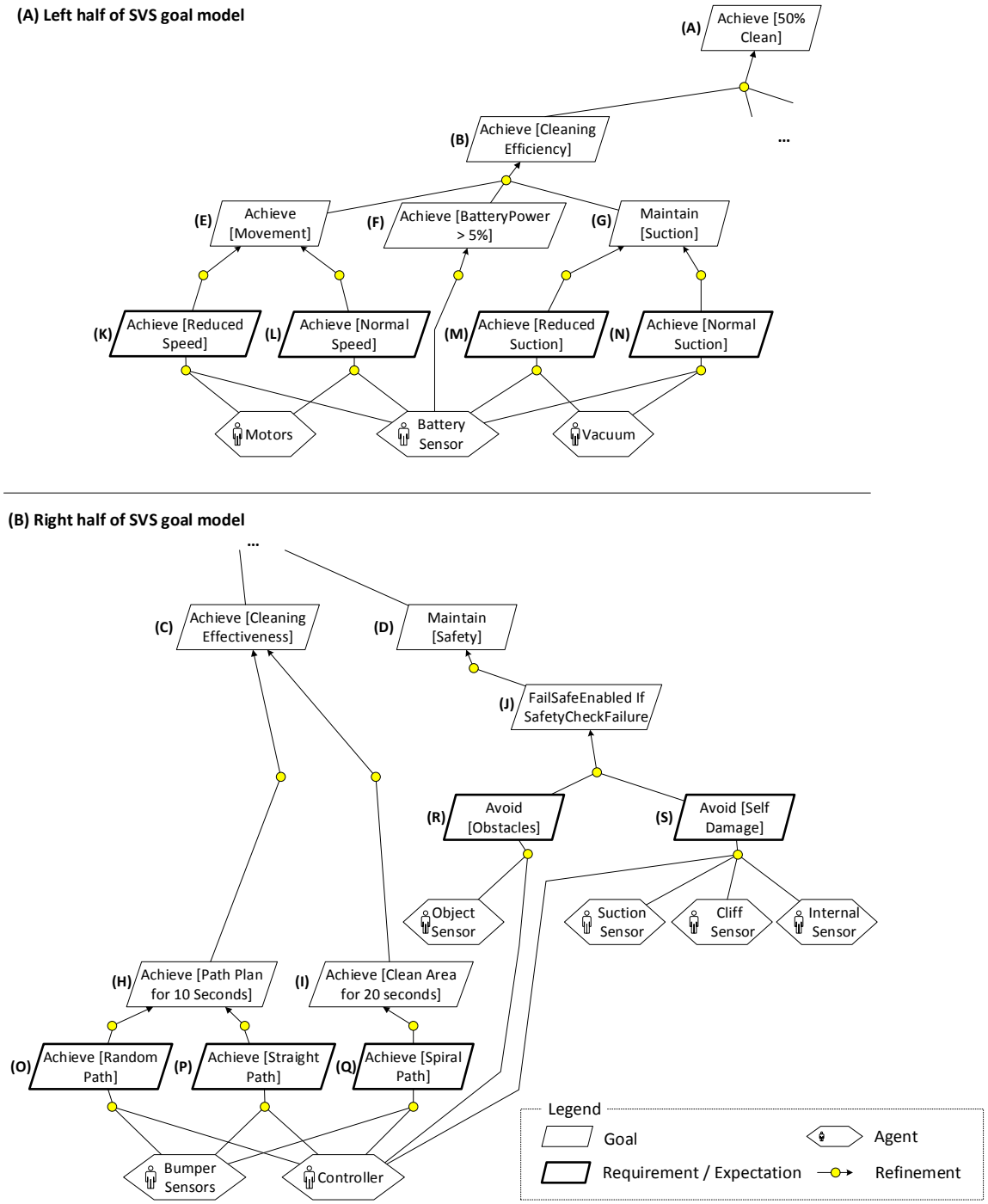


Figure A.1: KAOS goal model of the smart vacuum system application.

Table A.2 next presents the traceability links for each test case. Particularly, the first column presents the test case identifier and the second column lists the goals to which the test case is correlated. The traceability defined here enables run-time validation of test cases,

as is described in Chapter 6.4.2. Furthermore, we only linked test cases to *leaf-level* goals for the SVS, as they represent system requirements.

Table A.2: Traceability links for smart vacuum system application.

Test Case ID	**Correlated Goal(s)**
TC1	(K), (L), (M), (N), (O), (P), (Q), (R), (S)
TC2	(K), (L), (M), (N), (O), (P), (Q), (R), (S)
TC3	(K), (L), (M), (N), (O), (P), (Q), (R), (S)
TC4	(K), (L), (M), (N), (O), (P), (Q)
TC5	(K), (L), (M), (N), (O), (P), (Q)
TC6	(K), (L), (M), (N), (O), (P), (Q)
TC7	(R)
TC8	(R)
TC9	(S)
TC10	(R), (S)
TC11	(R), (S)
TC12	(R)
TC13	(R)
TC14	(S)
TC15	(R), (S)
TC16	(K), (L)
TC17	(K), (L)
TC18	(K), (L)
TC19	(K), (L)
TC20	(K), (M)
TC21	(K), (M)

Continued on next page

Table A.2 (cont'd)

Test Case ID)	**Correlated Goal(s)**
TC22	(K), (M)
TC23	(M), (N)
TC24	(M), (N)
TC25	(O)
TC26	(P)
TC27	(K), (L), (O), (P), (Q)
TC28	(R), (S)
TC29	(R), (S)
TC30	(R), (S)
TC31	(R), (S)
TC32	(S)
TC33	(R), (S)
TC34	(R), (S)
TC35	(S)
TC36	(R), (S)
TC37	(S)
TC38	(K)
TC39	(K)
TC40	(L)
TC41	(L)
TC42	(M)
TC43	(M)
TC44	(M)

Continued on next page

Table A.2 (cont'd)

Test Case ID)	**Correlated Goal(s)**
TC45	(N)
TC46	(N)
TC47	(N), (O), (P)
TC48	(O)
TC49	(O)
TC50	(P)
TC51	(P)
TC52	(Q)
TC53	(Q)
TC54	(Q)
TC55	(Q)
TC56	(Q)
TC57	(R)
TC58	(S)
TC59	(S)
TC60	(S)
TC61	(G), (N)
TC62	(F), (L), (M)
TC63	(L)
TC64	(D)
TC65	(D)
TC66	(D)
TC67	(O), (P), (Q)

Continued on next page

Table A.2 (cont'd)

Test Case ID)	**Correlated Goal(s)**
TC68	(O)
TC69	(P), (Q), (R)
TC70	(O), (P), (R)
TC71	(O), (P), (Q), (R)
TC72	(P)

Remote Data Mirroring Test Specification

This section overviews the test specification used for the RDM case study within this dissertation. Particularly, Table A.3 presents the test specification that is used as a basis for run-time testing and adaptation. For each test case, we present the test case identifier (ID), the test case type (i.e., invariant or non-invariant), a description, the expected value(s), and the safety tolerance used for adaptation. Furthermore, if the safety tolerance is listed as N/A, then the test case is considered non-invariant, however cannot be physically adapted.

Table A.3: Remote data mirroring test specification.

TC1	
Test case type	*Invariant*
Description	Data has been fully replicated throughout the network at the end of execution.
Expected value	$RDM.DataCopies = RDM.NumServers$
Safety tolerance	N/A
	Continued on next page

TC2	
Test case type	*Invariant*
Description	Operational cost has never exceeded the budget.
Expected value	$RDM.Cost < RDM.Budget$
Safety tolerance	N/A

TC3	
Test case type	*Non-invariant*
Description	Link sensor and RDM sensor can collectively measure network properties.
Expected value	$LinkSensor.Active = True$ AND $RDMSensor.Active = True$
Safety tolerance	N/A

TC4	
Test case type	*Non-invariant*
Description	Network actuator only enables the minimum number of links to replicate data.
Expected value	$NetworkActuator.ActiveNetworkLinks = (RDM.NumServers - 1)$
Safety tolerance	$[RDM.NumServers - 1, RDM.NumServers + 1]$

TC5	
Test case type	*Non-invariant*
Description	Network actuator has never partitioned network.
Expected value	$NetworkActuator.NumPartitions = 0$

Continued on next page

Table A.3 (cont'd)

Safety tolerance	$[0, 2]$
TC6	
Test case type	*Non-invariant*
Description	Risk threshold is never exceeded while sending data synchronously.
Expected value	$NetworkController.SynchronousSend = True$ AND $NetworkController.Risk \leq RiskThreshold$
Safety tolerance	$[RiskThreshold, RiskThreshold + 20.0]$
TC7	
Test case type	*Non-invariant*
Description	Diffusion time never exceeds maximum time while sending data asynchronously.
Expected value	$NetworkController.AsynchronousSend = True$ AND $DiffusionTime \leq MaxTime$
Safety tolerance	$[DiffusionTime - 5.0, DiffusionTime + 5.0]$
TC8	
Test case type	*Non-invariant*
Description	Network adaptation costs must remain 0.
Expected value	$AdaptationController.AdaptationCosts = 0.0$
Safety tolerance	$[0.0, 200.0]$
TC9	
Test case type	*Non-invariant*
Description	Link sensor can physically measure cost.

Continued on next page

Expected value	$LinkSensor.Active = True$
Safety tolerance	N/A

TC10	
Test case type	*Invariant*
Description	Link sensor is active.
Expected value	$LinkSensor.Active = True$
Safety tolerance	N/A

TC11	
Test case type	*Non-invariant*
Description	Link sensor can physically measure activity.
Expected value	$LinkSensor.Active = True$
Safety tolerance	N/A

TC12	
Test case type	*Non-invariant*
Description	Link sensor can physically measure loss rate.
Expected value	$LinkSensor.Active = True$
Safety tolerance	N/A

TC13	
Test case type	*Non-invariant*
Description	RDM sensor can physically measure workload.
Expected value	$RDMSensor.Active = True$
Safety tolerance	N/A

TC14

Continued on next page

Test case type	*Invariant*
Description	RDM sensor is active.
Expected value	$RDMSensor.Active = True$
Safety tolerance	*N/A*

TC15	
Test case type	*Non-invariant*
Description	RDM sensor can physically measure capacity.
Expected value	$RDMSensor.TotalCapacity > 0.0$
Safety tolerance	*N/A*

TC16	
Test case type	*Non-invariant*
Description	Network actuator can measure link state.
Expected value	$NetworkActuator.NumActiveLinks \geq 0.0$ AND $NetworkController.NumSynchronousLinks \geq 0.0$ AND $NetworkController.NumAsynchronousLinks \geq 0.0$
Safety tolerance	*N/A*

TC17	
Test case type	*Non-invariant*
Description	Network actuator can physically deactivate a link.
Expected value	$NetworkActuator.NumActiveLinks \geq 0.0$
Safety tolerance	*N/A*

TC18	
Test case type	*Invariant*

Continued on next page

Description	Network actuator is active.
Expected value	$NetworkActuator.Active = True$
Safety tolerance	N/A

TC19	
Test case type	*Non-invariant*
Description	Network actuator can physically activate a link.
Expected value	$NetworkActuator.NumActiveLinks \geq 0.0$
Safety tolerance	N/A

TC20	
Test case type	*Non-invariant*
Description	Network controller can synchronously send data.
Expected value	$NetworkController.NumSynchronousLinks \geq 0.0$
Safety tolerance	N/A

TC21	
Test case type	*Non-invariant*
Description	Network controller can send data over the network.
Expected value	$NetworkController.NumInsertedMessages > 0$ AND $NetworkActuator.NumActiveLinks > 0$
Safety tolerance	N/A

TC22	
Test case type	*Invariant*
Description	Network controller is active.
Expected value	$NetworkController.Active = True$

Continued on next page

Safety tolerance	N/A
TC23	
Test case type	*Non-invariant*
Description	Network controller can physically measure data sent.
Expected value	$NetworkController.NumSentMessages \geq 0$
Safety tolerance	N/A
TC24	
Test case type	*Non-invariant*
Description	Network controller can physically measure data received.
Expected value	$NetworkController.NumReceivedMessages \geq 0$
Safety tolerance	N/A
TC25	
Test case type	*Non-invariant*
Description	Network controller records that the data received equals the data sent.
Expected value	$NetworkController.NumReceivedMessages = NetworkController.NumSentMessages$
Safety tolerance	$[0.0, 10.0]$
TC26	
Test case type	*Non-invariant*
Description	Network controller can asynchronously send data.
Expected value	$NetworkController.NumAsynchronousLinks \geq 0.0$
Safety tolerance	N/A

Continued on next page

TC27	
Test case type	*Non-invariant*
Description	Adaptation controller can physically measure number of active data mirrors.
Expected value	$RDM.NumActiveServers \geq 0$
Safety tolerance	N/A

TC28	
Test case type	*Non-invariant*
Description	Adaptation controller can physically measure number of data mirrors.
Expected value	$RDM.NumServers \geq 0$
Safety tolerance	N/A

TC29	
Test case type	*Non-invariant*
Description	Adaptation controller has ensured that all data mirrors are active.
Expected value	$RDM.NumServers = RDM.NumActiveServers$
Safety tolerance	$[RDM.NumServers * 0.4, RDM.NumServers]$

TC30	
Test case type	*Invariant*
Description	Adaptation controller is active.
Expected value	$AdaptationController.Active = True$
Safety tolerance	N/A

Continued on next page

TC31	
Test case type	*Non-invariant*
Description	Adaptation controller can physically measure number of passive data mirrors.
Expected value	$RDM.NumPassiveServers \geq 0$
Safety tolerance	N/A

TC32	
Test case type	*Non-invariant*
Description	Adaptation controller has not put a data mirror into passive state.
Expected value	$RDM.NumPassiveServers = 0$
Safety tolerance	$[0, RDM.NumServers * 0.4]$

TC33	
Test case type	*Non-invariant*
Description	Adaptation controller can physically measure number of quiescent data mirrors
Expected value	$RDM.NumQuiescentServers \geq 0$
Safety tolerance	N/A

TC34	
Test case type	*Non-invariant*
Description	Adaptation controller has not put a data mirror into quiescent state.
Expected value	$RDM.NumQuiescentServers = 0$

Continued on next page

Table A.3 (cont'd)

Safety tolerance	$[0, RDM.NumServers * 0.4]$

Remote Data Mirroring Goal Correlation

This section provides an overview of how each test case is correlated to a particular goal, thereby providing a method for validating test results at run time. We now reproduce the RDM goal model previously presented in Figure 2.4 to simplify this section.

Table A.4 next presents the traceability links for each test case. Particularly, the first column presents the test case identifier and the second column lists the goals to which the test case is correlated. Again, the traceability defined in this section enables run-time validation of test cases, as was previously described in Chapter 6.4.2. For the RDM, we correlated test cases to all goals within the goal model to examine a different approach for test case derivation and correlation.

Table A.4: Traceability links for remote data mirroring application.

Test Case ID	**Correlated Goal(s)**
TC1	(A), (C)
TC2	(A), (B)
TC3	(D)
TC4	(E)
TC5	(F)
TC6	(G)
TC7	(H)

Continued on next page

Table A.4 (cont'd)

Test Case ID)	**Correlated Goal(s)**
TC8	(I)
TC9	(J)
TC10	(J), (K), (L)
TC11	(K)
TC12	(L)
TC13	(M)
TC14	(M), (N)
TC15	(N)
TC16	(O), (P)
TC17	(O)
TC18	(O), (P)
TC19	(P)
TC20	(Q)
TC21	(Q), (S)
TC22	(Q), (R), (S), (T)
TC23	(R), (T)
TC24	(R), (T)
TC25	(R), (T)
TC26	(S)
TC27	(U)
TC28	(U)
TC29	(U)
TC30	(U), (V), (W)

Continued on next page

Table A.4 (cont'd)

Test Case ID)	**Correlated Goal(s)**
TC31	(V)
TC32	(V)
TC33	(W)
TC34	(W)

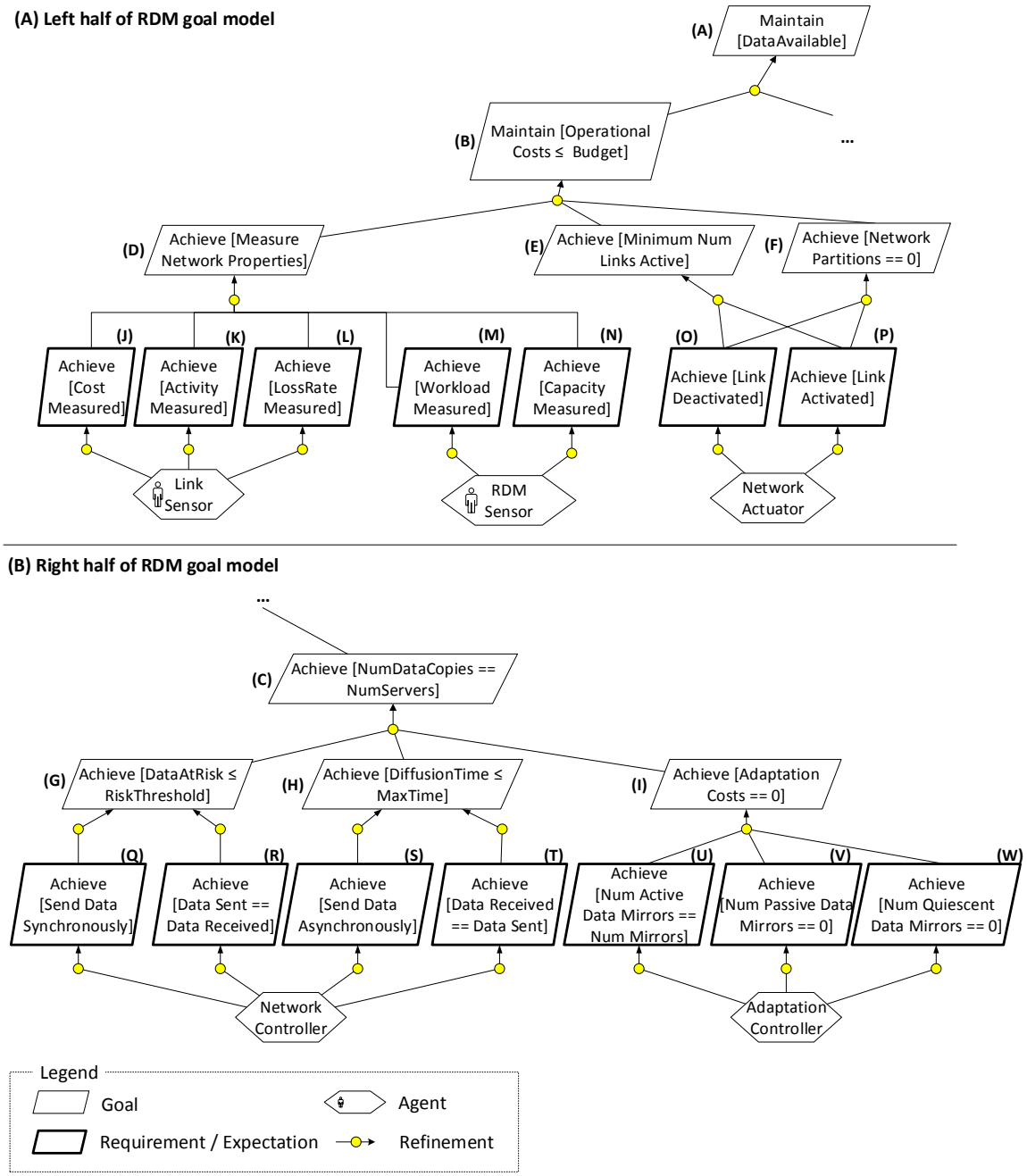


Figure A.2: KAOS goal model of the remote data mirroring application.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Shin-Young Ahn, Sungwon Kang, Jongmoon Baik, and Ho-Jin Choi. A weighted call graph approach for finding relevant components in source code. In *10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*, pages 539–544. IEEE, 2009.

[2] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

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

[4] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *Software, IET*, 1(6):219–232, 2007.

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

[6] Jonathon A Bauer and Alan B Finger. Test plan generation using formal grammars. *Proceedings of the 4th International Conference on Software Engineering*, pages 425–432, 1979.

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

[8] Nelly Bencomo, Jon Whittle, Peter Sawyer, Anthony Finkelstein, and Emmanuel Letier. Requirements reflection: Requirements as runtime entities. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 199–202, Cape Town, South Africa, May 2010. ACM.

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

[10] Paul E. Black. Dictionary of algorithms and data structures. *U.S. National Institute of Standards and Technology*, May 2006.

[11] Renesys Blog. Pakistan hijacks youtube. `http://www.renesys.com/2008/02/pakistan-hijacks-youtube-1/`, 2008.

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

[13] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, volume 57, pages 457–474. Springer US, 2003.

[14] Ilene Burnstein. *Practical software testing: a process-oriented approach*. Springer, 2003.

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

[16] Marco Canini, Vojin Jovanovic, Daniele Venzano, Boris Spasojevic, Olivier Crameri, and Dejan Kostic. Toward online testing of federated and heterogeneous distributed systems. In *USENIX Annual Technical Conference*, 2011.

[17] Marco Canini, Dejan Novaković, Vojin Jovanović, and Dejan Kostić. Fault prediction in distributed systems gone wild. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, pages 7–11, 2010.

[18] Mei-Hwa Chen, Michael R Lyu, and W Eric Wong. Effect of code coverage on software reliability measurement. *IEEE Transactions on Reliability*, 50(2):165–170, 2001.

[19] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. Using models at runtime to address assurance for self-adaptive systems. In *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science*, pages 101–136. Springer International Publishing, 2014.

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

[21] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In *Proc. of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 468–483, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Shang-Wen Cheng, David Garlan, and Bradley 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*, pages 2–8. ACM, 2006.

[23] O. Chikumbo, E. Goodman, and K. Deb. Approximating a multi-dimensional pareto front for a land use management problem: A modified moea with an epigenetic silencing metaphor. In *2012 IEEE Congress on Evolutionary Computation (CEC)*, 2012.

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

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

[26] Scott Dawson, Farnam Jahanian, Todd Mitton, and Teck-Lee Tung. Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of Annual Symposium on Fault Tolerant Computing*, pages 404–414. IEEE, 1996.

[27] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6(2):182 –197, apr 2002.

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

[29] R.F. DeMara and Kening Zhang. Autonomous fpga fault handling through competitive runtime reconfiguration. In *NASA/DoD Conference on Evolvable Hardware*, pages 109 – 116, 2005.

[30] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

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

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

[33] Ilenia Epifani, Carlo Ghezzi, Raffaela Mirandola, and Giordano Tamburrelli. Model evolution by run-time parameter adaptation. In *Proc. of the 31st International Conference on Software Engineering*, pages 111–121, Washington, DC, USA, 2009. IEEE Computer Society.

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

[35] Naeem Esfahani, Ehsan Kouroshfar, and Sam Malek. Taming uncertainty in self-adaptive software. In *Proc. of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 234–244. ACM, 2011.

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

[37] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. In *Requirements Engineering, 1995., Proc. of the Second IEEE International Symposium on*, pages 140 – 147, mar 1995.

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

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

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

[41] Erik M. Fredericks and Betty H. C. Cheng. Automated generation of adaptive test plans for self-adaptive systems. In *Accepted to Appear in Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '15, 2015.

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

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

[44] Erik M. Fredericks, Andres J. Ramirez, and Betty H. C. Cheng. Towards run-time testing of dynamic adaptive systems. In *Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '13, pages 169–174. IEEE Press, 2013.

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

[46] Christian Gagné and Marc Parizeau. Genericity in evolutionary computation software tools: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(02):173–194, 2006.

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

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

[49] Heather J. Goldsby and Betty H. C. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Model Driven Engineering Languages and Systems*, pages 568–583. Springer, 2008.

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

[51] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, 2009.

[52] Mark Harman, Phil McMinn, Jerffeson Teixeira Souza, and Shin Yoo. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification*, volume 7007 of *Lecture Notes in Computer Science*, pages 1–59. Springer Berlin Heidelberg, 2012.

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

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

[55] Michael Jackson and Pamela Zave. Deriving specifications from requirements: an example. In *Proceedings of the 17th International Conference on Software Engineering*, pages 15–24. ACM, 1995.

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

[57] Keneth A. De Jong. Evolutionary computation, a unified approach. *The MIT Press*, March 2002.

[58] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing. Preprint, 12 pages, available online `http://homes.cs.washington.edu/~rjust/publ/mutants_real_faults_fse_2014.pdf`, 2014.

[59] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, and Giuseppe Valetto. An approach to autonomizing legacy systems. Technical report, DTIC Document, 2005.

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

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

[62] John R. Koza. Genetic programming: On the programming of computers by means of natural selection (complex adaptive systems). *The MIT Press*, December 1992.

[63] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *Software Engineering, IEEE Transactions on*, 16(11):1293–1306, 1990.

[64] Joel Lehman and Kenneth O. Stanley. Exploiting open-endedness to solve problems through the search for novelty. In *Proceedings of the Eleventh International Conference on Artificial Life (ALIFE XI)*. MIT Press, 2004.

[65] Emmanuel Letier, David Stefan, and Earl T. Barr. Uncertainty, risk, and information value in software requirements and architecture. In *Proceedings of the 36th International Conference on Software Engineering*, pages 883–894, 2014.

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

[67] Khaled Mahbub and George Spanoudakis. A framework for requirements monitoring of service based systems. In *Proc. of the 2nd international conference on Service oriented computing*, pages 84–93. ACM, 2004.

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

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

[70] Atif M Memon, Martha E Pollack, and Mary Lou Soffa. Hierarchical gui test case generation using automated planning. *IEEE Transactions on Software Engineering*, 27(2):144–155, 2001.

[71] Edison Mera, Pedro Lopez-García, and Manuel Hermenegildo. Integrating software testing and run-time checking in an assertion verification framework. In *Logic Programming*, pages 281–295. Springer, 2009.

[72] Bertrand Meyer. Seven principles of software testing. *Computer*, 41(8):99–101, 2008.

[73] Harlan D Mills, Michael Dyer, and Richard C Linger. Cleanroom software engineering. *IEEE Software*, 1987.

[74] S. Mitra, T. Wongpiromsarn, and R.M. Murray. Verifying cyber-physical interactions in safety-critical systems. *IEEE Security Privacy*, 11(4):28–37, 2013.

[75] Subhasish Mitra, W.-J. Huang, N.R. Saxena, S.-Y. Yu, and E.J. McCluskey. Reconfigurable architecture for autonomous self-repair. *Design Test of Computers, IEEE*, 21(3):228 – 240, may-june 2004.

[76] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44 –51, oct. 2009.

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

[78] Cu D. Nguyen, Anna Perini, Paolo Tonella, and Fondazione Bruno Kessler. Automated continuous testing of multiagent systems. In *The Fifth European Workshop on Multi-Agent Systems (EUMAS)*, 2007.

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

[80] Charles Ofria and Claus O Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial life*, 10(2):191–229, 2004.

[81] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54 –62, 1999.

[82] Thomas Ostrand. *Encyclopedia of Software Engineering*. Wiley, 2002.

[83] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[84] Esteban Pavese, Víctor Braberman, and Sebastian Uchitel. Automated reliability estimation over partial systematic explorations. In *Software Engineering (ICSE), 2013 35th International Conference on*, 2013.

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

[86] 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 *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 568 –571, nov. 2011.

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

[88] Andres J. Ramirez, Erik M. Fredericks, Adam C. Jensen, and Betty H. C. Cheng. Automatically relaxing a goal model to cope with uncertainty. In Gordon Fraser and Jerffeson Teixeira de Souza, editors, *Search Based Software Engineering*, volume 7515, pages 198–212. Springer Berlin Heidelberg, 2012.

[89] Andres J Ramirez, David B Knoester, Betty HC Cheng, and Philip K McKinley. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th international conference on Autonomic computing*, pages 97–106, 2009.

[90] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, pages 216–226, 1979.

[91] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proc. of the 14th International Symposium on Software Reliability Engineering*, pages 281–292. IEEE Computer Society, 2003.

[92] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *Proc. of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 76–85, Boston, Massachusetts, USA, 2004. ACM.

[93] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14:1–14:42, May 2009.

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

[95] Nirmal R. Saxena, Santiago Fernandez-Gomez, Wei-Je Huang, Subhasish Mitra, Shu-Yi Yu, and Edward J. McCluskey. Dependable computing and online testing in adaptive and configurable systems. *IEEE Des. Test*, 17(1):29–41, January 2000.

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

[97] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat-fault injection based automated testing environment. In *Eighteenth International Symposium on Fault-Tolerant Computing, 1988. FTCS-18, Digest of Papers*. IEEE, 1988.

[98] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.

[99] V.E.S. Souza, A. Lapouchnian, and J. Mylopoulos. (requirement) evolution requirements for adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems, 2012 ICSE Workshop on*, pages 155 –164, 2012.

[100] V.E.S. Souza and J. Mylopoulos. From awareness requirements to adaptive systems: A control-theoretic approach. In *Requirements@Run.Time (RE@RunTime), 2011 2nd International Workshop on*, pages 9–15. IEEE Computer Society, 2011.

[101] Pascale Thevenod-Fosse and Helene Waeselynck. Statemate applied to statistical software testing. *ACM SIGSOFT Software Engineering Notes*, 18(3):99–109, 1993.

[102] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient instrumentation for code coverage testing. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 86–96. ACM, 2002.

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

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

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

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

[107] Margus Veanes, Pritam Roy, and Colin Campbell. Online testing with reinforcement learning. In *Formal Approaches to Software Testing and Runtime Verification*, pages 240–253. Springer, 2006.

[108] Norha M. Villegas, Hausi A. Müller, Gabriel Tamura, Laurence Duchien, and Rubby Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 80–89, Waikiki, Honolulu, Hawaii, USA, 2011. ACM.

[109] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Proc. of the First IEEE International Conference on Autonomic Computing*, pages 70–77. IEEE Computer Society, 2004.

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

[111] K. Welsh, P. Sawyer, and N. Bencomo. Towards requirements aware systems: Runtime resolution of design-time assumptions. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 560–563, 2011.

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

[113] 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*, pages 79–88, 31 2009-sept. 4 2009.

[114] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380. ACM, 2006.