APPLYING EVOLUTIONARY COMPUTATION TECHNIQUES TO ADDRESS ENVIRONMENTAL UNCERTAINTY IN DYNAMICALLY ADAPTIVE SYSTEMS

By

Andres J. Ramirez

A DISSERTATION

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

Computer Science - DOCTOR OF PHILOSOPHY

2013

ABSTRACT

APPLYING EVOLUTIONARY COMPUTATION TECHNIQUES TO ADDRESS ENVIRONMENTAL UNCERTAINTY IN DYNAMICALLY ADAPTIVE SYSTEMS

By

Andres J. Ramirez

A dynamically adaptive system (DAS) observes itself and its execution environment at run time to detect conditions that warrant adaptation. If an adaptation is necessary, then a DAS changes its structure and/or behavior to continuously satisfy its requirements, even as its environment changes. It is challenging, however, to systematically and rigorously develop a DAS due to environmental uncertainty. In particular, it is often infeasible for a human to identify all possible combinations of system and environmental conditions that a DAS might encounter throughout its lifetime. Nevertheless, a DAS must continuously satisfy its requirements despite the threat that this uncertainty poses to its adaptation capabilities. This dissertation proposes a model-based framework that supports the specification, monitoring, and dynamic reconfiguration of a DAS to explicitly address uncertainty. The proposed framework uses goal-oriented requirements models and evolutionary computation techniques to derive and fine-tune utility functions for requirements monitoring in a DAS, identify combinations of system and environmental conditions that adversely affect the behavior of a DAS, and generate adaptations on-demand to transition the DAS to a target system configuration while preserving system consistency. We demonstrate the capabilities of our model-based framework by applying it to an industrial case study involving a remote data mirroring network that efficiently distributes data even as network links fail and messages are dropped, corrupted, and delayed.

To my family, who has always supported me in every possible way.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Dr. Betty H.C. Cheng for guiding this dissertation research from its inception to its conclusion. Her guidance, feedback, attention to detail, and high expectations have shaped this dissertation research.

In addition, I want to thank my dissertation committee members: Dr. Erik Goodman, Dr. Charles Ofria, Dr. Philip McKinley, and Dr. Xiaobo Tan. I greatly appreciate their valuable feedback, insights, and willingness to read through hundreds of pages multiple times.

I would also like to thank Dr. Percy Pierre and Dr. Barbara O'Kelly for recruiting me to the Alfred P. Sloan program at Michigan State University and giving me the opportunity of a lifetime. Both of you took a chance on me and changed the rest of my life for the better. I hope to pay it forward.

Many others provided help and encouragement along the way. I wish to thank current and past SENS and DevoLab members. Dave Knoester, Heather Goldsby, Ben Beckmann, and Adam Jensen helped set the foundation for ideas that would evolve into this dissertation. Likewise, Jared Moore, Tony Clark, Brian Connelly, Erik Fredericks, Chad Byers, Anu Pakanati, Erick Nieves, Jorge Cintron, and Nelson Sepulveda provided technical and motivational support during key stages of this dissertation.

Finally, I would like to thank Amanda and Odin for putting up with me through this long, draining, and sometimes infuriating process. I would not have finished this dissertation without them, and I appreciate the sacrifices that have been made so I could focus and complete my research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

Software systems increasingly interact with real-world physical devices upon which humans critically depend. This shared cyber-physical boundary exposes software systems to a myriad of environmental conditions, including some that may prevent the system from satisfying its requirements. For many safety-critical application domains, such as power grids, extended downtimes where humans manually inspect, analyze, and modify application code in response to system and environmental changes is neither practical nor safe. A dynamically adaptive system (DAS) observes itself and its execution environment and, if necessary, dynamically changes its structure and behavior to continuously satisfy its objectives [73]. Developing a DAS that satisfies its requirements, however, is a challenging task due to system and environmental uncertainty. For this dissertation, system and environmental uncertainty refer to combinations in inaccuracies and imprecisions in the measurements of environmental properties, as well as unanticipated combinations of environmental conditions, respectively. Consequently, automated techniques for exploring and mitigating how the environment affects a DAS at run time are increasingly important.

Most software engineering techniques and approaches for developing a DAS [35, 53, 62, 85, 120] have focused on identifying, at design time, all possible conditions

that might warrant adaptation after the system is deployed. In particular, these approaches often enumerate and encode specific adaptations to address system and environmental conditions conducive to a requirements violation. For example, several object-oriented adaptation-enabling frameworks [11, 35, 83] and middleware approaches [17, 62, 75] use rule-based repositories to map monitoring information to adaptation strategies that mitigate specific undesirable conditions. Nevertheless, these rule-based techniques enable a DAS to adapt only in response to conditions identified at design time under specific assumptions about what the environment should be at run time. Unfortunately, inadequate assumptions may prevent a DAS from safely and correctly adapting itself [28, 93, 111, 112].

This dissertation presents a model-based framework for specifying, monitoring, and adapting a DAS to satisfy functional and non-functional goals while explicitly addressing system and environmental uncertainty. We present investigation results that explore how goal-oriented requirements models and evolutionary computation can be leveraged to detect and mitigate obstacles or conditions that prevent a DAS from satisfying its requirements, without explicitly specifying adaptation conditions and mitigation strategies at design time. In particular, the proposed model-based framework automatically generates and fine-tunes an executable abstraction of a goal-oriented requirements model in the form of utility functions for monitoring how a DAS satisfies requirements at run time. This framework also supports the automatic identification of system and environmental conditions that prevent a DAS from satisfying its requirements, thereby suggesting goals that require revisions. Moreover, this framework also supports the generation of *safe* adaptations that balance competing concerns. Each adaptation specifies structural and behavioral changes that a DAS should perform, as well as the set of reconfiguration instructions for transitioning the system to its target configuration while preserving consistency. For some application domains, these safe adaptations can be generated on-demand at run time.

1.1 Problem Description

A key objective in software engineering is to design and implement software systems that continuously satisfy their stakeholder's objectives, even if unanticipated conditions arise during execution. As a result, the field of DASs is gaining attention from the software engineering community as it provides a means for a software system to observe, analyze, and respond to system and environmental changes at run time. To this end, software engineering for self-adaptive systems focuses on identifying, analyzing, and integrating adaptation-specific concerns throughout the *entire* development lifecycle of a DAS, starting with the requirements that the DAS must satisfy and continuing through the maintenance of the DAS. In this manner, elicited requirements, including adaptation needs and constraints, guide the design, development, testing, deployment, and even execution of a DAS.

Based on DAS-focused techniques described in the literature [7, 14, 120], when developing a new DAS, a requirements engineer first identifies invariant and non-invariant requirements that the DAS must satisfy. A requirements engineer then analyzes and decomposes these requirements into goals to determine dependencies, constraints, and assumptions that the DAS must satisfy at run time [13]. Goal models enable requirements engineers to identify and resolve obstacles that can prevent a DAS from satisfying its requirements [28, 29, 107]. Developers then implement adaptation strategies to reconfigure the system in response to these anticipated obstacles [15, 35, 83] while preserving system consistency before, during, and after adaptation [64, 121]. At run time, the DAS monitors itself and its execution environment to detect the occurrence of these conditions and, if necessary, self-reconfigures to prevent or mitigate the obstacle.

This development process enables a DAS to monitor and self-adapt in response to conditions and obstacles identified at design time, under specific sets of assumptions about what the system and its environment should be like at run time. Nev-

ertheless, complete identification of all possible obstacles may be unachievable by a requirements engineer as it is often infeasible for a human to exhaustively identify all possible combinations of environmental inputs that a DAS will encounter throughout its lifetime [13, 28, 114]. Furthermore, it is equally challenging for a requirements engineer to anticipate and evaluate how these environmental inputs may affect the behavior of a DAS. These observations imply that certain key adaptation-centric decisions must be deferred until run time, when actual system and environmental conditions are known. As a result, throughout this design and development process, the DAS research community faces three key challenges in terms of how to:

- effectively and efficiently monitor requirements satisfaction in a DAS.
- anticipate sources of uncertainty and their effects upon a DAS.
- safely reconfigure a DAS in response to changing system and environmental conditions while satisfying functional and non-functional requirements.

1.2 Thesis Statement

This research defines a model-based framework for explicitly addressing system and environmental uncertainty in a DAS. This framework uses a goal-oriented requirements model as a point of reference to support the automatic specification, monitoring, and adaptation in a DAS.

Thesis Statement: *Evolutionary computation can be harnessed to support a model-based framework for specifying, monitoring, and dynamically adapting a system to explicitly address system and environmental uncertainty.*

1.3 Research Contributions

Three major overarching research objectives guide the investigations described in this dissertation. First, we emphasize the use of lightweight, computationally inexpensive techniques to assess the environment and its impact on system functionality. Second, we maximize the use of automation for the overall process of specifying, monitoring, and adapting a DAS. Lastly, we minimize the need to identify a predetermined set of adaptations in response to anticipated reconfiguration scenarios at design time. To achieve these objectives, the proposed model-based framework integrates goal-oriented requirements models and evolutionary computation techniques to support the automatic monitoring and dynamic adaptation of a DAS, respectively.

Guided by these research objectives, we now state the research contributions of this dissertation. Specifically, our model-based framework supports the:

1. Automatic generation and fine-tuning of utility functions for requirements monitoring in a DAS even in the presence of system and environmental uncertainty. These utility functions enable a DAS to identify conditions conducive to a requirements violation and diagnose potential causes for a requirements violation.

2. Automatic exploration of different combinations of system and environmental conditions that a DAS may encounter throughout its lifetime. Analyzing how system and environmental conditions affect the behavior of a DAS more rigorously facilitates the identification of alternate design choices for resolving obstacles at the requirements level.

3. Automatic generation of adaptations that specify the target system configuration and the series of reconfiguration steps necessary for safely reaching that target configuration. These adaptations can be generated by only specifying, at a high level of abstraction, the general concerns and functional and non-functional objectives that the DAS should satisfy.

This dissertation realizes these research contributions in the form of a suite of techniques and tools. The data flow diagram in Figure 1.1 depicts each component in our model-based framework. Specifically, Athena uses a goal model of the DAS to automatically derive utility functions for requirements monitoring at run time. Using these utility functions, Loki generates combinations of system and environmental conditions that produce interesting and representative DAS behaviors, including requirements violations and latent behaviors, which are undesirable behaviors that do not violate known requirement but should be disallowed nonetheless. AutoRELAX uses these identified sources of uncertainty to fine-tune the utility functions derived by Athena such that minor and transient sources of uncertainty do not cause unnecessary adaptations. Together, these first three techniques support the first and second research contributions by enabling a DAS to detect requirements violations even in environments with uncertain conditions.

As Figure 1.1 also illustrates, Plato generates suites of target reconfigurations that specify what the configuration of a DAS should be, including its monitoring infrastructure, in response to changes in its environment. Hermes, on the other hand, generates adaptation paths, or series of reconfiguration instructions that a DAS can apply at run time to safely transition itself to its target configuration while preserving system consistency. Together, Plato and Hermes enable a DAS to self-reconfigure at run time without having to encode adaptations in response to specific system and environmental conditions, thereby supporting the third contribution.

1.4 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 reviews background material on the remote data mirroring application and dynamically adaptive systems, as well as several enabling techniques that include goal-oriented require-

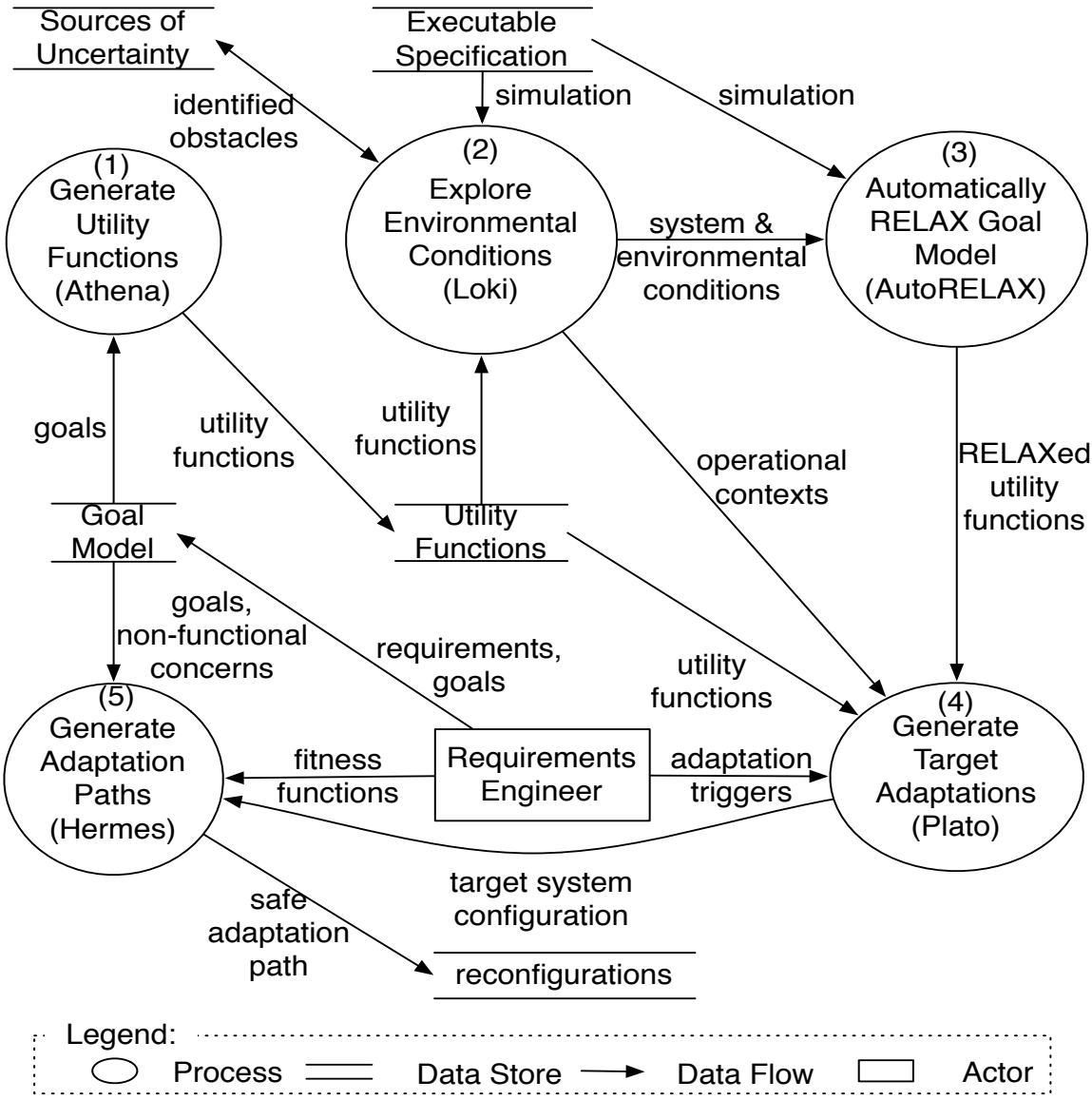


Figure 1.1: Data flow diagram overviewing our model-based framework for specifying, monitoring, and dynamically adapting a DAS.

ments engineering, the RELAX requirements specification language for DASs, and evolutionary computation. Chapter 3 introduces Athena and describes the process it uses to derive utility functions for different types of goals. Chapter 4 describes how Loki generates operational contexts that produce interesting DAS behaviors, including requirements violations. Chapter 5 describes how AutoRELAX leverages sources of uncertainty discovered by Loki to fine-tune the utility functions derived by Athena.

Chapters 6 and 7 describe how Plato and Hermes search for target system reconfigurations and safe adaptation paths, respectively. Although each chapter presents experimental results, Chapter 8 illustrates how the suite of techniques can be integrated by applying them to an end-to-end case study involving a remote data mirroring network that must distribute data messages across unreliable networks. Chapter 9 overviews related work on model-based approaches to dynamic adaptation. Lastly, Chapter 10 presents conclusions, summarizes the contributions of this dissertation, and discusses future directions.

Appendices include additional descriptions of how individual techniques were applied to another case study involving an intelligent vehicle system that performs adaptive cruise control, lane keeping, and collision avoidance.

Chapter 2

Background

This chapter provides background information on the remote data mirroring application and three topics fundamental to this dissertation: dynamically adaptive systems, goal-oriented requirements engineering, and evolutionary computation. First, we overview the objectives and constraints of remote data mirroring, the application domain used to illustrate the techniques developed in this research. We then overview the key objectives and theoretical foundations of adaptive systems. Next, we describe how a goal-oriented requirements model captures the objectives, requirements, assumptions, and constraints of a software system. Lastly, we overview evolutionary computation and describe how it searches for optimal or near-optimal solutions in vast and complex solution spaces.

2.1 Remote Data Mirroring

Remote data mirroring (RDM) is a data protection technique that replicates and stores copies of critical data at one or more secondary sites [1, 50, 54, 55, 56, 115]. RDM are intended to guarantee continuous access to important data by keeping two or more copies of important information physically isolated from each other, thereby protecting data from failures that may affect the primary copy. In the event of a

failure, recovery typically involves either a site failover to another data mirror or data reconstruction. RDM has been previously applied to efficiently replicate and distribute on-demand television media across limited bandwidth channels [48]. In addition, RDM has also been applied at companies such as Google and Yahoo to support their heavily distributed search-engine infrastructure.

Designing and deploying a remote mirror, however, is a complex and expensive task that should be done only when the cost of losing data outweighs the cost of protecting it [50]. For instance, *ad hoc* solutions may provide inadequate data protection, poor write performance, and incur high network costs [55]. Similarly, over-engineered solutions may incur expensive operational costs to defend against negligible risks.

Two important RDM design choices include the type of network link used to connect the mirrors and the remote mirroring protocol. Each network link incurs an operational cost and has a measurable throughput, latency, and loss rate that collectively determines the overall RDM design and its performance and reliability [50]. Remote mirroring protocols affect both network performance and data reliability, and can be categorized as either synchronous or asynchronous. In *synchronous propagation*, the secondary site receives and applies each write before the write completes at the primary site. In batched *asynchronous propagation*, updates accumulate at the primary site and are periodically propagated to the secondary site, which then applies each batch atomically. As Table 2.1 illustrates, synchronous propagation achieves zero potential data loss but consumes large amounts of network bandwidth. In contrast, batched asynchronous propagation achieves better network performance than synchronous propagation, but may have a higher potential data loss.

For this dissertation, the emphasis is on dynamically reconfiguring an RDM network. Specifically, we focus on the construction and maintenance of an overlay network of RDMs such that data may be distributed to all data mirrors. To this end, an RDM network shall:

Table 2.1: Propagation methods time intervals and data sizes [55].

Time Interval	Avg. Data Batch Size
0	0 GB
1 min.	0.0436 GB
5 min.	0.2067 GB
1 hr.	2.091 GB
4 hrs.	6.595 GB
12 hrs.	15.12 GB
24 hrs.	27.388 GB

- Remain connected at all times.
- Never exceed the allocated monetary budget, as specified by the end-user.
- Distribute data as efficiently as possible by minimizing the amount of bandwidth consumed when diffusing data.
- Shall never lose or corrupt protected data.

All considerations combined, data diffusion is a multi-objective problem where data must be distributed as efficiently as possible while minimizing expenses and potential data loss.

For this dissertation, we implemented an executable specification that satisfies these RDM requirements and constraints. In particular, we modeled and implemented an RDM network simulation as a completely connected undirected graph where each node and edge represents an RDM or a network link, respectively. The specific number of data mirrors and the underlying network link topology can be configured in different ways to explore different RDM design scenarios. Moreover, the operational characteristics of each RDM node and network link, such as workload or capacity, were randomly generated using different statistical distributions (i.e, normal, uniform) based on RDM operational models previously presented by Keeton *et al.* [55].

The RDM controller performs several key processes during each simulation time step. First, the RDM controller collects system and environmental monitoring data

from its sensors. The RDM controller then replicates, archives, and distributes new data messages to neighboring data mirrors. Simultaneously, the RDM controller also evaluates its utility functions to detect adverse conditions that might warrant adaptation. To self-adapt, the RDM network selects an alternate network configuration that best addresses current system and environmental conditions. If an adaptation is required, then a target configuration is selected and an adaptation path is generated to safely transition the executing RDM network to its target configuration.

2.2 Dynamically Adaptive Systems

It is often impossible to know or enumerate, at design time, all possible combinations of system and environmental conditions that a software system will encounter during its lifetime [13, 114]. Furthermore, functional and non-functional requirements may change after the software system is deployed. The primary objective of a dynamically adaptive system (DAS) is to continuously satisfy its requirements by modifying its structure and behavior in response to changing requirements and environmental conditions [73, 83]. As Figure 2.1 illustrates, a DAS comprises a set of finite steady-state machines, each of which satisfies specific requirements and constraints within a domain or environment [120]. An adaptation, therefore, corresponds to a transition from one *source* steady-state system to another *target* steady-state system.

Kephart and Chess [58] proposed the concept of an autonomic computing system to explicitly address the growing complexity of managing large-scale software systems. Akin to the self-governing human nervous system, the elements of an autonomic computing system are mostly self-managed, guided only by high-level objectives provided by a systems administrator. Specifically, each component in an autonomic computing system adopts a MAPE-K architecture [113] to monitor and analyze its environment, as well as plan and execute reconfigurations as necessary. Furthermore, each of these

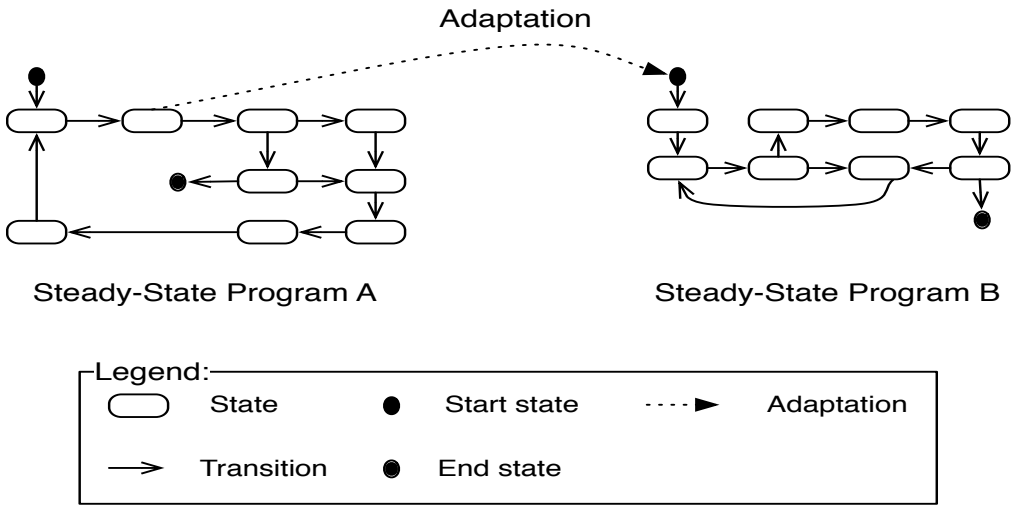


Figure 2.1: A DAS comprises a set of finite steady-state machines.

processes is supported by a knowledge repository that maps monitored conditions to specific reconfigurations. These processes enable an autonomic computing system to leverage self-managing properties, such as self-configuration, self-healing, self-optimization, and self-protection; for brevity, we use the term "self-*" when referring to these terms collectively. Within the context of an autonomic computing system, self-configuration modifies the interactions between system components, self-healing detects and recovers from faults, self-protection identifies and fends-off malicious attacks, and self-optimization improves the delivery of services to clients.

Although all autonomic computing systems are self-adaptive in nature, not all adaptive systems are necessarily autonomic. The key distinction between an autonomic computing system and an adaptive system is the level of automation in the self-reconfiguration process. Specifically, an autonomic computing system is mostly self-managed, only accepting high-level inputs from a systems administrator with regards to the system's objectives. In contrast, an adaptive system is not necessarily self-managed; key adaptation-specific decisions, such as when and how to reconfigure an application, may be determined by a system's administrator at run time. Through-

out this dissertation, we only consider DAS's that are primarily self-managed, and thus use the term DAS to include autonomic computing systems unless otherwise noted.

2.2.1 Processes of a DAS

A DAS performs introspection and intercession in order to detect and respond to changing requirements and environmental conditions [73]. Introspection is the ability of a software system to observe its own behavior, as well as measure properties of its surrounding execution environment. Intercession, on the other hand, is the ability for a software system to reason about data collected during introspection and modify its behavior in response. Often, a DAS adopts a feedback control loop to automatically and continuously apply these introspection and intercession processes at run time, respectively [10, 79, 102]. Following the MAPE-K architectural model [58], a DAS performs introspection via monitoring, and intercession via analysis, planning or decision-making, and execution or reconfiguration.

The data flow diagram in Figure 2.2, based on the traditional MAPE-K architectural model, provides additional details on how a DAS monitors both system and environmental properties as part of introspection in order to detect conditions leading to a requirements violation [28, 29, 97]. More specifically, *internal monitoring* enables a DAS to observe its own operational state, and *external monitoring* enables a DAS to measure properties of its surrounding execution environment. A DAS may gather this external monitoring information via active or passive monitoring, depending on whether sensors automatically report changes in the system and its environment, or whether the DAS must periodically probe sensors to obtain the desired data, respectively. Although monitoring is a key process in a DAS, it is often intrusive, computationally expensive, and difficult to design [36]. As a result, tradeoffs between monitoring costs (overhead) and accuracy must be carefully balanced when selecting

which system and environmental properties to monitor, as well as how often and at what granularity to collect the data.

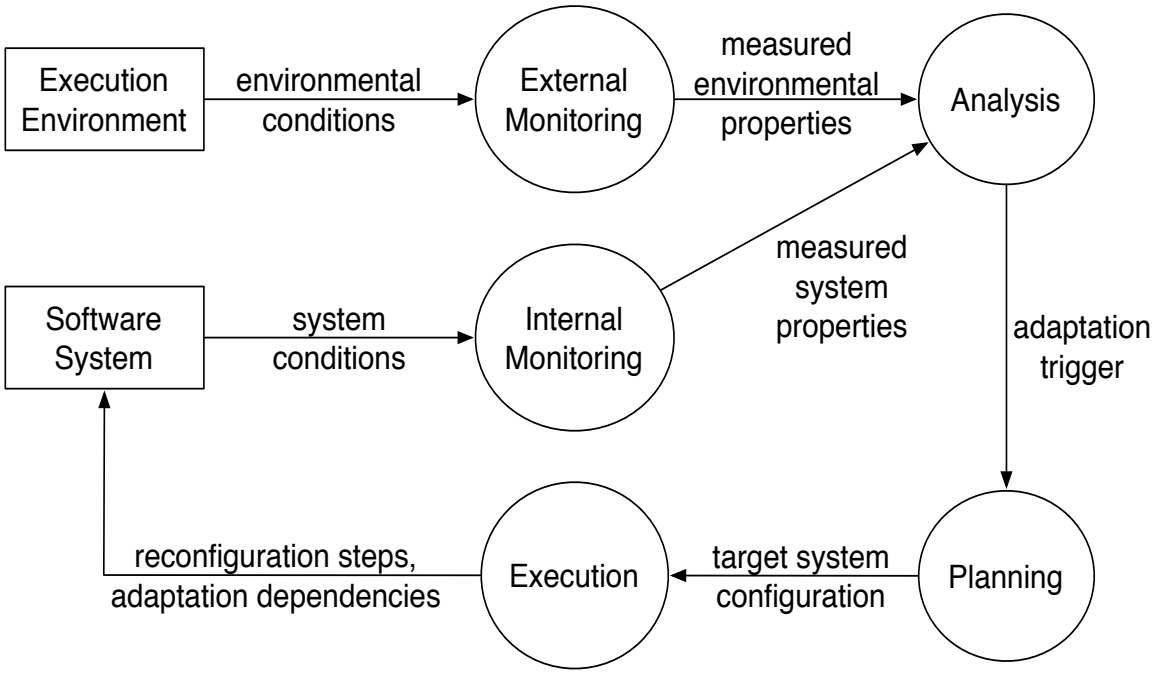


Figure 2.2: Data flow diagram depicting the monitoring, analysis, planning, and execution processes of a DAS.

As part of intercession, a DAS performs decision-making by analyzing monitoring data and, if necessary, planning for any necessary adaptations. As the data flow diagram in Figure 2.2 depicts, a DAS analyzes monitoring data to determine if an adaptation is necessary in order to mitigate undesirable changes in the system or its execution environment. If an adaptation is necessary, then the DAS must also plan *how* and *when* to safely adapt the executing system in order to preserve system consistency (i.e., prevent the loss or corruption of application data). To achieve this objective, a DAS first selects a target system configuration, and then either selects or generates a series of reconfiguration instructions to safely reach that target configuration [64].

Software adaptation can be accomplished either at the parameter or composi-

tional level [73, 117]. In parameter adaptation, the DAS adjusts operational variables and strategies to achieve optimal behavior. More specifically, parameter adaptation can switch between *existing* strategies already built into the system, but may not adopt new strategies or components after deployment. Thus, while parameter adaptation is relatively simple to implement, the possible range of adaptation changes supported by this approach is limited by whichever scenarios were considered at design-time. In contrast, compositional adaptation enables a DAS to add, remove, and modify algorithmic and structural components at run time. Although more difficult to implement, compositional adaptation provides greater flexibility in terms of reconfiguration than parameter adaptation.

Adaptation techniques can be classified either as static (closed) or dynamic (open) [73]. Static adaptation occurs either at development-, compile-, or load-time. Development-time composition hard codes reconfiguration strategies into an application, and thus requires manual modifications to the code base in order to incorporate new adaptive behaviors. Similarly, compile-time composition requires recompiling or relinking different components in order to adapt the application to suit specific environments. Load-time composition delays the decision of which components to load until run time. Dynamic composition, in contrast, refers to tunable and mutable methods that modify the DAS's behavior or structure at run time. Tunable reconfiguration supports the optimization of crosscutting concerns in response to changing environmental conditions. Mutable reconfiguration, the most flexible adaptation form, supports changes to the entire application. While this added flexibility increases the difficulty associated with ensuring the correctness and integrity of a DAS across adaptations, dynamic composition is more powerful than static composition as it can achieve new behaviors at run time that were not available at design time.

2.2.2 Adaptation Semantics

A DAS must preserve system consistency before, during, and after adaptation [120]. To achieve this objective, a DAS must apply a dynamic change management protocol that guides components towards active, passive, and quiescent states in bounded time [64]. An *active* component may initiate and service transaction requests. In contrast, a *passive* component may accept and service transaction requests, but may not initiate new requests nor be currently engaged in a transaction it initiated. A *quiescent* component, on the other hand, is neither engaged in a transaction nor will it receive or initiate new transaction requests on its own. For a component to reach a quiescent state, however, it needs to cooperate with all of its neighboring components to establish a region of passive components. Guiding all neighboring components towards a passive state ensures that no new transaction requests will be initiated with the component attempting to reach quiescence, thus "freezing" the state and communication channels of the component before an adaptation takes place.

Within the context of our adaptive RDM network, each data mirror implements the DCM protocol such that it may reach active, passive, and quiescent states in bounded time. In particular, each active data mirror can receive, replicate, archive, and distribute data messages to other adjacent data mirrors in the network. Likewise, each passive data mirror can receive, replicate, and archive data messages but may not distribute these to other data mirrors, as that would entail initializing a new transaction. For operational reasons, it is preferable to maximize and minimize the number of active and passive data mirrors at run time, respectively. Moreover, from a performance perspective, it is extremely detrimental for data mirrors to reach quiescence as they will be unable to receive, replicate, archive, and distribute data messages throughout the network, thus creating the possibility for data loss and data unavailability.

Adaptation semantics capture the intricacies between the possible states of a

DAS during adaptation. As previously defined by Zhang and Cheng [119], the three most commonly encountered adaptive behaviors are one-point, guided, and overlap adaptation. As Figure 2.3 shows, these adaptations differ in *when* an adaptation begins and terminates. In *one-point* adaptation, a single transition transfers execution from the source system to the target system. In *guided* adaptation, the source program must first reach a quiescent state such that the adaptive transition does not leave the system in an inconsistent state. The source program reaches a quiescent state by entering a restricted mode (R_{cond}) where some features are disabled (i.e., a passive state), and then applies a one-point adaptation to transfer execution to the target program. In *overlap* adaptation, both source and target behaviors may coincide as the target system begin to execute before the source system has terminated. Eventually, the source behavior terminates and only the target behavior is observable.

One-point adaptation can be the most disruptive of the three adaptation semantics. For example, in the RDM application, one-point adaptation guides all data mirrors to quiescence, enacts the corresponding reconfigurations, and then completes the reconfiguration process by reactivating all data mirrors. This pause in the source behavior implies that no message can be received, replicated, archived, or distributed during the adaptation process. In contrast, two-point adaptation, which corresponds to the traditional DCM protocol, only guides data mirrors affected by an adaptation to a quiescent state, thereby allowing other data mirrors to continue receiving, replicating, archiving, and distributing messages as parts of the network are being adapted. Lastly, overlap adaptation interleaves independent reconfiguration steps in order to minimize the number of data mirrors that reach quiescence simultaneously while maximizing the functionality provided by active data mirrors during the reconfiguration process.

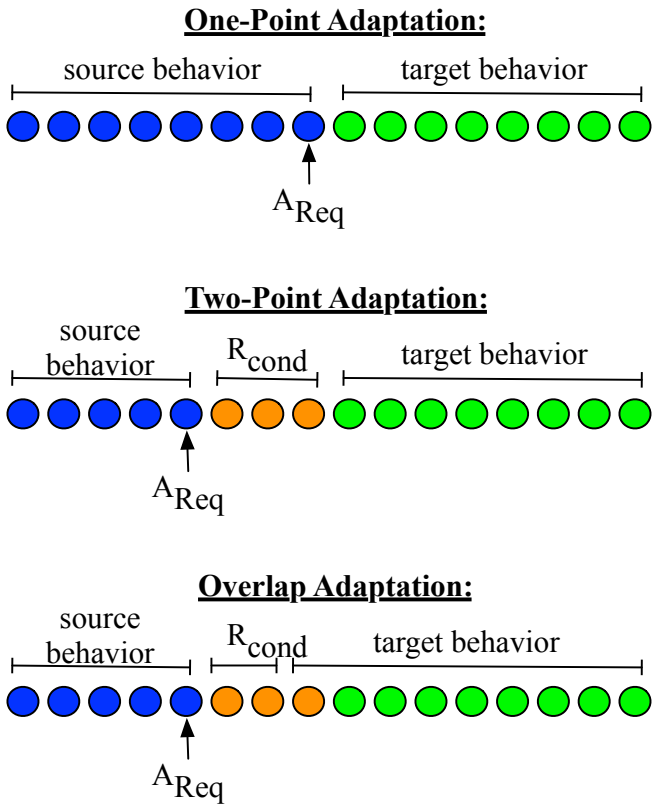


Figure 2.3: One-point, guided, and overlap adaptation semantics [119]. For interpretation of the references to color in this and all other figures, the reader is referred to the electronic version of this dissertation.

2.3 Goal-Oriented Requirements Engineering

The key requirements engineering activities include eliciting, exploring, analyzing, and documenting the various objectives, constraints, and assumptions that a software system-to-be must meet in order to solve a specific problem for a stakeholder [105]. As Jackson and Zave formalized in their 4-variable model [47], the problem to be solved by the system-to-be often arises within an organizational, technical, or physical world context. As a result, the system-to-be shares some phenomena with the problem world. The phenomena that arise at the shared boundary between the system-to-be and its execution environment define the interface through which the system-to-be interacts with the world and its stakeholders. Thus, in order to

satisfy its objectives, constraints, and assumptions, the system-to-be must monitor and control parts of the shared boundary.

Goal-oriented requirements engineering (GORE) extends Jackson's 4-variable model with the concept of a goal that guides the elicitation, evaluation, documentation, and analysis of requirements based on the objectives of the system-to-be. A goal captures the intentions of a stakeholder on the system-to-be, as well as the assumptions and expectations upon its execution environment [105]. In particular, a goal captures the intended behavior of a system by declaratively prescribing the states it may reach during execution. In this manner, goal-orientation plays a central role in requirements engineering by providing the rationale for a requirement, facilitating the structuring of complex specifications at differing levels of detail, delimiting the scope of the system-to-be, and enabling the reasoning about alternative design options [18, 105].

A *functional* goal declares a service that the system-to-be must provide to its clients, and a *non-functional* goal imposes a quality constraint or criterion upon the delivery of those services. Functional goals are often expressed as ‘‘`Achieve`’’ or ‘‘`Maintain`’’ goals. An achieve goal prescribes a behavior where some target condition must sooner or later become true, and a maintain goal prescribes a behavior where some condition must remain true at all times. In this manner, a maintain goal is an invariant that the system-to-be must always satisfy and an achieve goal is a non-invariant that may become temporarily unsatisfied at run time. While the satisfaction of a hard goal can be assessed in a clear-cut, true or false sense, the satisfaction of a soft goal cannot be precisely determined because it involves subjective and potentially conflicting preferences from multiple stakeholders [118]. Thus, a hard goal can be achieved whereas a soft goal can be *satisficed*, or satisfied to some degree [16, 52, 118].

2.3.1 Goal-Oriented Requirements Modeling

The GORE process gradually decomposes high-level functional and non-functional goals into finer-grained subgoals [105]. The semantics of goal decomposition are formally and graphically captured in a directed acyclic graph where each node represents a goal and each edge represents a goal refinement. Figure 2.4 presents a goal model for the RDM application in the KAOS modeling language [18, 105]. As this figure illustrates, KAOS depicts goals and refinements as parallelograms and refinements as directed arrows pointing in the direction of their higher-level or parent goal, respectively. KAOS supports AND/OR refinements, where a goal that has been AND-decomposed can only be satisfied if all of its subgoals are satisfied, and a goal that has been OR-decomposed can be satisfied if at least one of its subgoals is satisfied. For example, the goal (A) "`Maintain[DataAvailable]`" can only be satisfied if both subgoals (B) "`Maintain[OperationalCosts <= Budget]`" and (C) "`Achieve[NumberDataCopies == NumberServers]`" are satisfied. In contrast, goal (G) "`Achieve[DataAtRisk <= RiskThreshold]`" can be satisfied if either subgoals (Q) "`Achieve[Send Data Synchronously]`" or (R) "`Achieve[Send Data Asynchronously]`" is satisfied. In general, AND-refinements capture milestones that must be performed in order to satisfy a single higher-level goal, whereas an OR-refinement provides alternative paths for achieving a given goal.

Goal decomposition continues until each goal is assigned to a single agent capable of fully achieving that goal. An agent is an active system component that restricts its behavior to fulfill leaf-goals in a goal model [105]. Two different types of agents can fulfill goals: system and environmental agents. While a system agent is an automated component controllable by the system-to-be, an environmental agent is often a human being or some automated component that cannot be controlled by the system-to-be. As Figure 2.4 illustrates, the KAOS modeling language depicts an agent with a hexagon connected to a bolded leaf-goal via an *assignment* link. KAOS

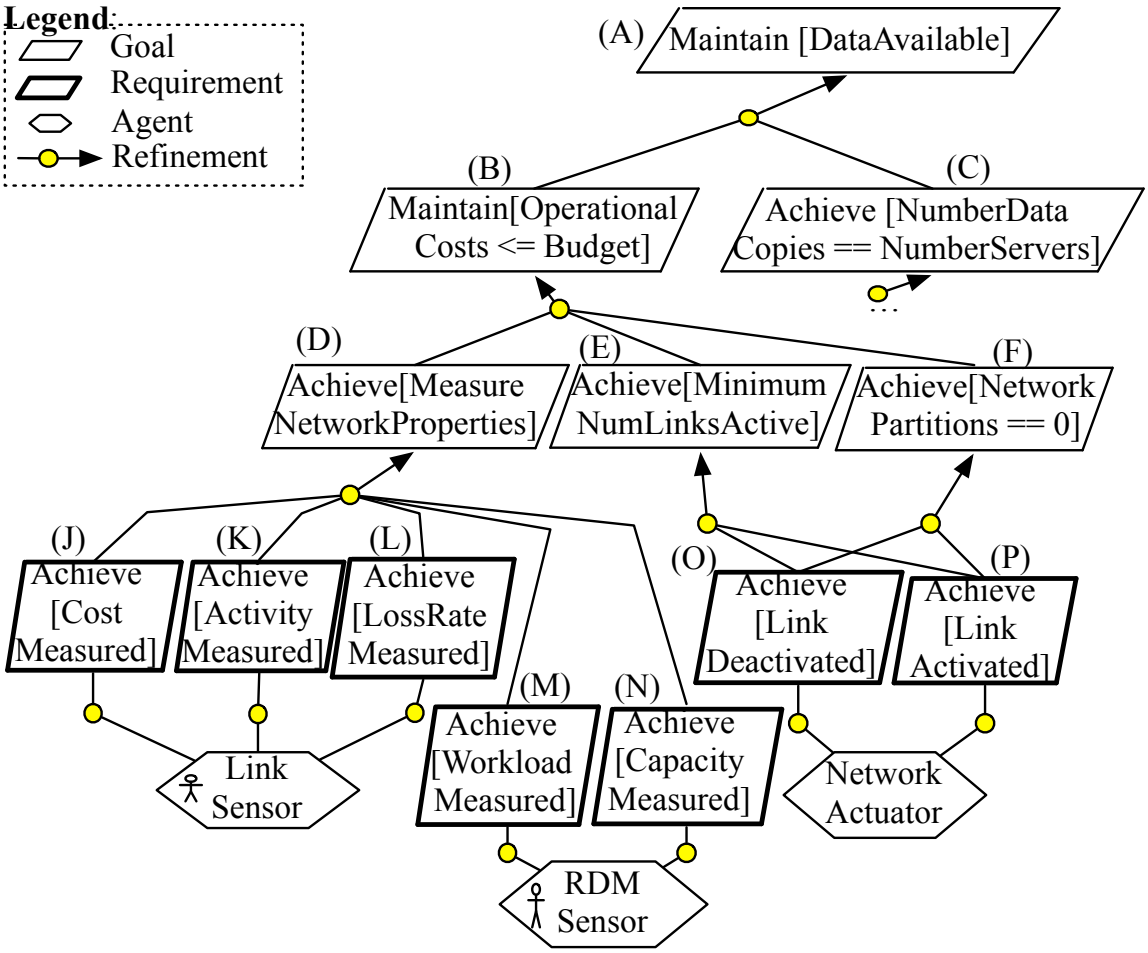


Figure 2.4: KAOS Goal model for RDM application.

also denotes the difference between a system and an environmental agent by including a stick figure icon within the hexagon of an environmental agent, such as agents `RDM Sensor` and `Link Sensor` in Figure 2.4. A goal under the responsibility of a system agent is a *requirement* and a goal under the responsibility of an environmental agent is an *expectation*.

Goal Refinement Patterns. Darimont and van Lamsweerde [19] developed a set of goal refinement patterns to assist requirements engineers in decomposing high-level goals into finer-grained goals. Each of these goal refinement patterns have been proven correct and complete. Thus, a requirements engineer may instantiate these patterns while preserving their correctness and completeness. For instance,

the unmonitorability refinement pattern [19, 105] captures the interactions across the shared boundary between system and environmental agents. This refinement pattern is applicable when a system agent alone is unable to monitor a condition in a goal formulation. Since the system agent is unable to directly observe a specific environmental property, this refinement pattern reassigns the monitoring responsibility to an environmental agent capable of doing so. The system agent is then able to achieve its original objective by processing the monitoring information gathered by the environmental agent.

Obstacle Mitigation. In addition to gradually decomposing goals, goal-oriented requirements engineering also focuses on obstacle identification, analysis, and mitigation. Specifically, an obstacle is a precondition for the non-satisfaction of an assertion that is a goal or some assumption made about the software application domain [105]. In other words, an obstacle is some event or condition that prevents the satisfaction of the goal with which it is associated. Conceptually, goals and obstacles are dual opposites of each other. While a goal prescribes a maximal set of admissible behaviors for the system-to-be, an obstacle captures a minimal set of inadmissible behaviors [105, 107]. More specifically, the definition of an obstacle, and therefore the satisfaction of a goal, can be formalized as follows:

$$\neg O_1 \wedge \neg O_2 \wedge ... \wedge \neg O_n \models G$$

where O_i is an obstacle to a goal G. This definition states that if all obstacles are negated (i.e., resolved), then goal G is necessarily satisfied. Based on this definition, Letier and van Lamsweerde [107] proposed a goal-oriented requirements-model based approach for obstacle analysis and mitigation. In general, this obstacle analysis and mitigation approach results in a more robust software system, though completeness and correctness is ultimately bound by what a requirements engineer knows about

the application domain [28].

Letier and van Lamsweerde [105, 107] also presented an iterative process for identifying, assessing, and controlling obstacles in KAOS. In particular, a requirements engineer identifies obstacles by first selecting a goal, assumption or hypothesis in the goal model. Preferably, the selected goal should be a leaf node in the goal mode as the finer-grained the target, the finer-grained the obstacle is to identify, assess, and resolve. Next, a requirements engineer negates the selected goal G to obtain the root obstacle, $\neg G$. Subsequently, the root obstacle $\neg G$ must also be AND/OR refined according to the desired level of detail. This process continues until obstruction pre-conditions are reached (i.e, obstacles are sufficiently fine-grained) to enable an assessment of their feasibility, likelihood, and resolvability.

To facilitate obstacle analysis, Letier and van Lansweerde [105, 107] also extended the KAOS modeling language to formally and graphically capture obstacles and their relationships to goals. Similar to a goal model, an obstacle diagram is a directed acyclic graph with AND/OR refinements anchored on a particular assertion, such as a goal or assumption, in a goal model. As obstacle `Activity Not Measured` in Figure 2.5 shows, KAOS depicts obstacles as inverted parallelograms connected by a crossed-out association link to the goal it obstructs. Obstacles can also be progressively refined into finer-grained obstacles that are eventually connected to a countermeasure goals through resolution links. An AND-refinement from a parent obstacle specifies the set of conjoined sub-obstacles whose satisfaction is sufficient for the satisfaction of the parent obstacle. For instance, `Link Failed` and `NetworkLink Not Faulty` are two necessary preconditions for the obstacle `Link Not Activated` to occur. In contrast, an OR-refinement from a parent obstacle captures the different ways that the obstacle may be satisfied given the domain properties and hypothesis. For instance, in this example, either `NetworkLink Faulty` or `Sensor Data Dropped` prevents the satisfaction of goal (K) `Activity Measured`.

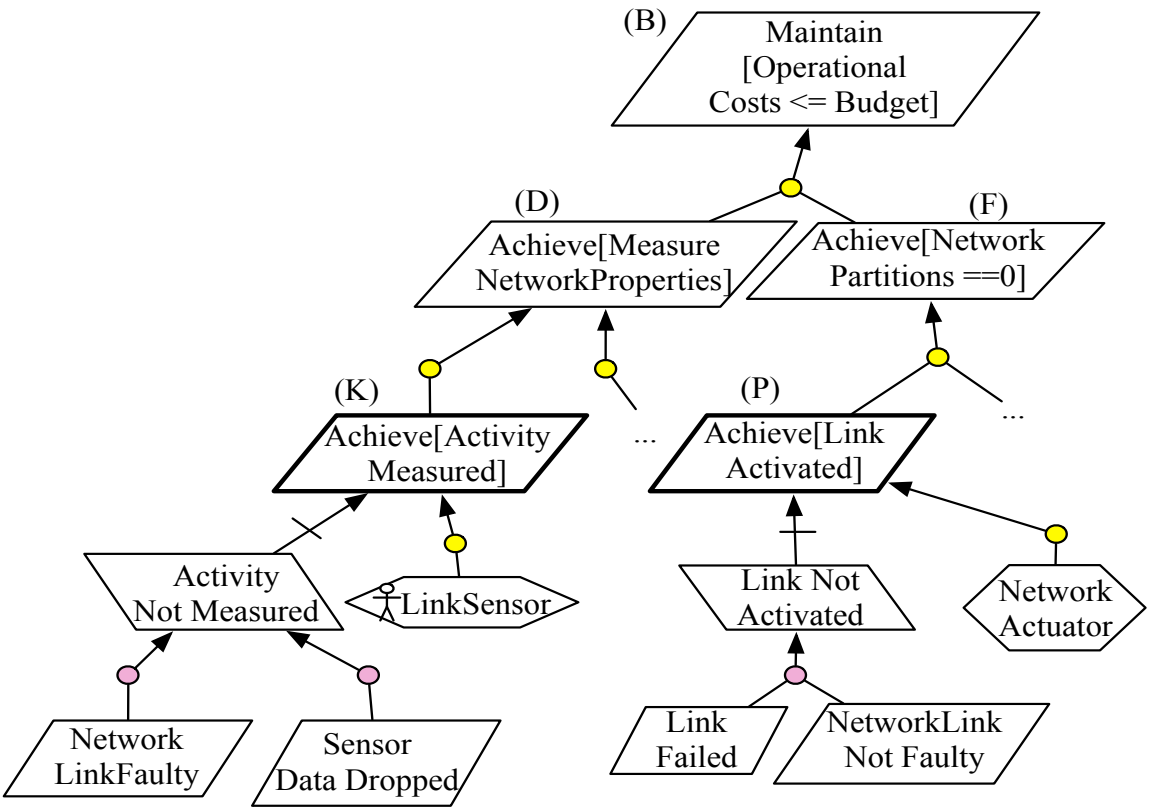


Figure 2.5: Goal-oriented obstacle decomposition in KAOS.

Once a requirements engineer has identified and refined a set of obstacles to the desired level of granularity, these need to be evaluated in terms of their likelihood and criticality. First, it must be shown that an obstacle can arise in the software system by assessing whether each leaf obstacle is compatible with elicited domain knowledge, as well as find some behavior of the system-to-be that satisfies it. Next, the likelihood and criticality of each obstacle must be determined by applying several application-specific rules and heuristics. For instance, the likelihood that an obstacle occurs propagates bottom-up in an obstacle diagram, from independent sub-obstacles to parent obstacles, as follows:

- If obstacle O is AND-refined by sub-obstacles SO_i, then the likelihood of O is computed by taking the minimum value of the likelihood of SO_i.

- If obstacle O is OR-refined by sub-obstacles SO_i, then the likelihood of O is computed by taking the maximum value of the likelihood of SO_i.

Such propagation rules produce rough estimates that may need revision through domain-specific weakening, in the case of an AND-refinement, or strengthening, in the case of an OR-refinement.

2.3.2 RELAX Specification Language

RELAX [114] is a requirements specification language for identifying and mitigating sources of environmental uncertainty in a DAS. Instead of enumerating all possible ways that a goal might become obstructed, RELAX focuses on *declaratively* specifying the sources and impacts of uncertainty at the shared boundary between the system-to-be and its execution environment [47]. A requirements engineer organizes this information into ENV, MON, and REL elements. ENV specifies environmental properties that may or may not be directly observable by the monitoring infrastructure of a DAS; MON specifies the elements that make up the DAS's monitoring infrastructure; and REL defines how to compute the values of ENV properties from available MON elements.

The semantics of RELAX operators are defined in terms of fuzzy logic in order to explicitly account for and constrain the extent to which a non-invariant goal may become temporarily unsatisfied [114]. Table 2.2 gives the current set of RELAX operators and describes how each can be used to evaluate the satisfaction of a non-invariant goal. For instance, if the goal (F) in Figure 2.4 is RELAXed into `Achieve[AS FEW Network Partitions AS POSSIBLE]`, then this goal now states that it is *temporarily* acceptable for the network to become partitioned at run time while data messages continue to be replicated and distributed. The fuzzy logic underpinnings of RELAX support both ordinal and temporal constraints upon goal satisfaction.

Table 2.2: Table of RELAX operators and their semantics [114]

RELAX Operator	Informal Definition
AS EARLY AS POSSIBLE ϕ	ϕ becomes true in some state as close to the current time as possible
AS LATE AS POSSIBLE TO ϕ	ϕ becomes true in some state as close to time $t = \infty$ as possible
AS CLOSE AS POSSIBLE TO f ϕ	ϕ is true at periodic intervals where the period is as close to f as possible
AS CLOSE AS POSSIBLE TO q ϕ	there is some function Δ such that $\Delta(\phi)$ is quantifiable and $(\Delta(\phi) - q)$ is as close to 0 as possible
AS MANY AS POSSIBLE ϕ	there is some function Δ such that $\Delta(\phi)$ is as close ∞ as possible
AS FEW AS POSSIBLE ϕ	there is some function Δ such that $\Delta(\phi)$ is quantifiable and is as close as possible to 0

RELAX Process. Cheng *et al.* [13] proposed an approach for introducing RELAX operators into non-invariant requirements in a KAOS goal model. To apply their goal modeling approach, a requirements engineer must first informally specify the various ENV, MON, and REL elements. Next, each goal in the model must be classified either as an invariant or non-invariant goal. For a non-invariant goal, a requirements engineer must then determine if environmental uncertainty may cause that goal to become temporarily unsatisfied. If the non-invariant goal may become unsatisfied at run time, then a requirements engineer must also apply a corresponding RELAX operator to constrain the degree to which that goal may be temporarily violated. This last step must be manually repeated for each non-invariant goal in the model.

2.4 Evolutionary Computation

Evolutionary computation (EC) is a subfield of artificial and computational intelligence [33, 51]. Within these fields, EC is a family of stochastic search-based techniques that includes, but is not limited to, approaches such as genetic algorithms [46], genetic programming [63], evolution strategies [95], digital evolution [82],

ant colony optimization [23], and particle swarm optimization [57]. These EC approaches and techniques are often applied to solve complex optimization problems in a wide range of domains, including software engineering [39, 40, 44, 49], antenna design, and robotics [63]. Most EC approaches harness the concept of Darwinian evolution by natural selection in order to guide the search process towards promising areas of the solution space. EC approaches tend to be most successful when the solution space is not deceptive or misleading, and provides sufficient stepping stones for the evolutionary algorithm to gradually compose solutions of greater complexity.

The data flow diagram in Figure 2.6 depicts the key processes in an evolutionary algorithm. Specifically, once a developer decides to apply a particular EC approach or technique for a given problem, she must first *encode* the elements or parameters that comprise a candidate solution to the problem being solved. This encoding maps solution elements to specific data structures that can be operated on by the corresponding evolutionary algorithm [33]. Using biology-based terminology, the elements and parameters that an evolutionary algorithm can manipulate are known as *genes*, and the set of genes comprising a candidate solution are known as a *genome*. The particular encoding of a solution can significantly affect the likelihood of a particular solution being found by reducing or expanding the overall search space. As a result, different EC approaches tend to encode solutions in different structures depending on what constitutes a solution and how it may be evaluated. Moreover, in EC candidate solutions are often referred to as *individuals*.

In general, EC approaches tend to simultaneously examine multiple individuals in parallel, storing these candidate solutions in a *population*. The EC algorithm must therefore initialize the starting population either by randomly generating a collection of individuals or by resuming the search process from a previously stored population. Next, evolutionary algorithms apply a set of fitness functions to evaluate the quality, or *fitness*, of each individual in the current population and thus guide the search

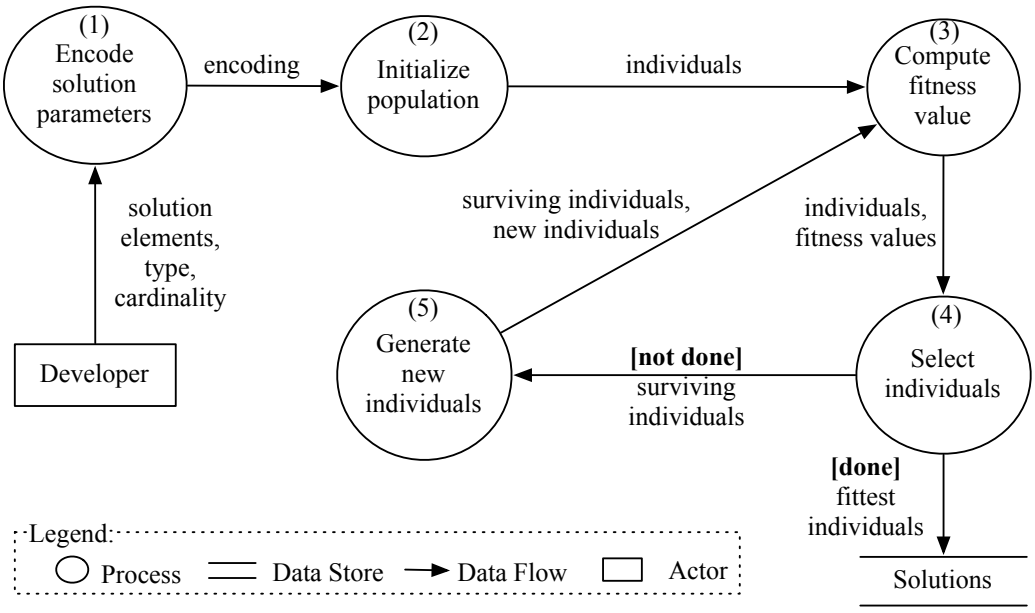


Figure 2.6: Data flow diagram illustrating key processes in evolutionary algorithms.

process towards more promising areas of the solution space. Essentially, a fitness function maps the solution encoded in an individual to a scalar fitness value that is proportional to how well it addresses the various concerns of the problem being solved.

Evolutionary algorithms leverage this fitness value to compare the relative qualities of different individuals in the population and determine where to search in subsequent iterations. Several *selection* methods can be applied, such as tournament and roulette wheel selection [46]. Tournament selection randomly selects k individuals from the population, and the individual with the highest fitness value survives onto the new population. In contrast, roulette wheel, or fitness proportionate, selection assigns a survival probability to each individual that is proportional to its fitness value. Ideally, but not necessarily, those individuals who survive this selection process represent the best solutions found thus far by the evolutionary algorithm.

Next, an evolutionary algorithm generates new solutions by applying operators such as crossover and mutation. The *crossover* operator exchanges sets of genes in

a genome from two existing individuals in the population, thereby forming two new individuals. The *mutation* operator, in contrast, creates new individuals by randomly inserting, removing, swapping, and/or modifying genes in an existing individual. Ideally, the crossover operator generates new individuals with higher fitness values by exchanging building blocks, or important groups of genes, between individuals. In contrast, the mutation operator adds diversity, in the form of new solutions, into a population that might otherwise not be explored through the crossover operator alone.

As Figure 2.6 depicts by the bolded conditional guard, EC approaches sequentially apply evaluation, selection, individual generation operators (i.e., crossover and mutation) either for a given number of iterations or *generations*, or until a sufficiently good solution is found. Lastly, after an evolutionary algorithm terminates, it returns either a single individual with the highest fitness in the population, or a set of individuals, each representing different solutions that represent different tradeoffs between competing concerns.

2.4.1 Genetic Algorithms

A genetic algorithm [46] is a stochastic search-based heuristic that generates solutions to complex optimization problems by replicating several biologically-inspired processes of natural selection; in this work, the Loki, AutoRELAX, and Plato techniques make use of genetic algorithms. In general, a genetic algorithm encodes candidate solutions in a representation amenable to evaluation and manipulation. Although not mandatory, this candidate configuration is often fixed in length throughout the evolutionary process. Figure 2.7 presents two commonly-used representations in a genetic algorithm. For instance, Figure 2.7(A) illustrates the simplest and most common representation in a genetic algorithm, a fixed-length binary string. Similarly, Figure 2.7(B) illustrates an alternate, fixed-length, vector-based representation where

genes can be represented directly as integers, floating-point values, and other more complex data structures. Each encoded individual can be mapped back to a unique element in the problem domain.

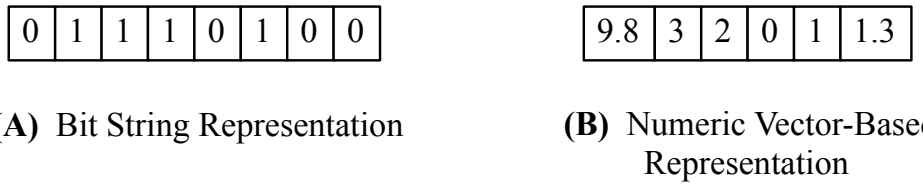
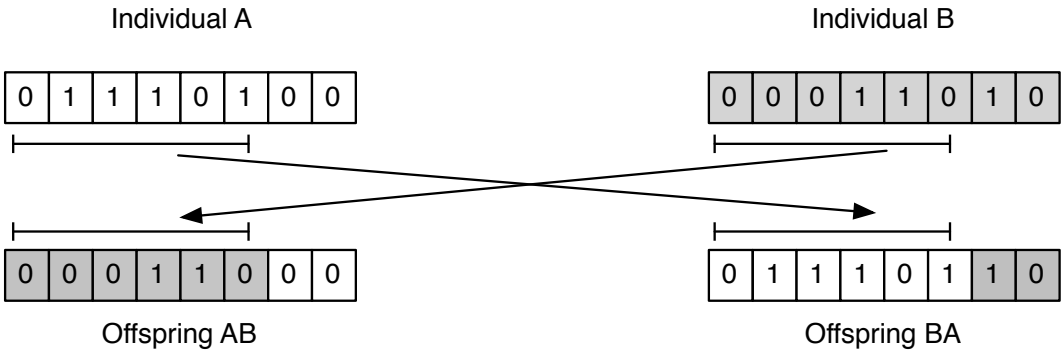


Figure 2.7: Examples of encodings in a genetic algorithm.

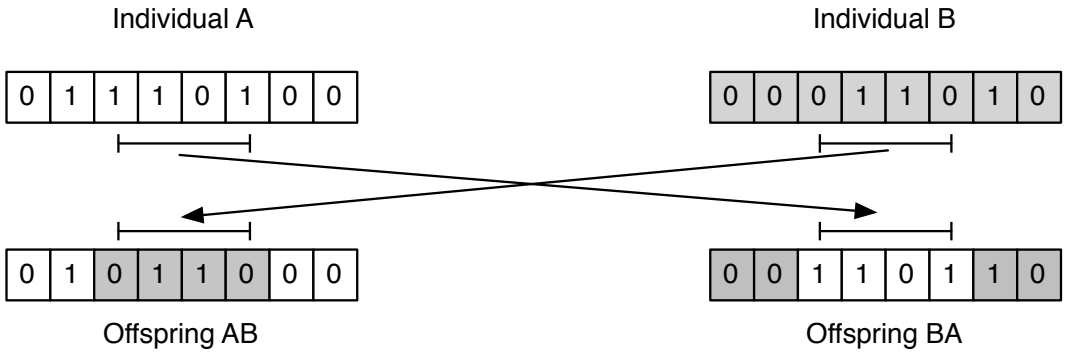
The fitness of each individual in the population must be computed in order to determine which areas of the solution space the genetic algorithm should explore next. To compute an individual's fitness value, a genetic algorithm applies either a single or multiple domain-specific fitness functions that map an individual's encoded solution to a scalar value proportional to its quality. Multiple fitness functions can be combined in different ways, such as a linear weighted sum. Moreover, each fitness function can be associated with different weight coefficients, thereby guiding the genetic algorithm towards a specific type of solution.

A genetic algorithm typically applies a crossover operator in order to generate new candidate solutions from existing individuals in the population. Two crossover operators are commonly applied in a genetic algorithm, one- and two-point crossover. As Figure 2.8 illustrates, in one-point crossover, a genetic algorithm selects a gene in the genome and then exchanges genes beyond that boundary between the two parents. In contrast, in two-point crossover, a genetic algorithm randomly selects two gene boundaries in the genome and then exchanges the parent's genes that lie within those bounds.

As with most other evolutionary algorithms, a genetic algorithm also applies mutation operators to maintain a diverse range of solutions in the population. A genetic algorithm normally mutates an individual by randomly changing its encoded



(A) One-point crossover operator for a bit string representation in a genetic algorithm.



(B) Two-point crossover operator for a bit string representation in a genetic algorithm.

Figure 2.8: One-point and two-point crossover in a genetic algorithm.

elements. For instance, Figure 2.9 illustrates a random *flip* mutation where the value of a single gene changes from zero to one. More complex genomic representations might require mutation operators to replace existing genes with randomly drawn values.

2.4.2 Genetic Programming

Genetic programming [63], often considered to be a specific instance of a genetic algorithm [46], is a search-based technique that generates *executable* programs to solve

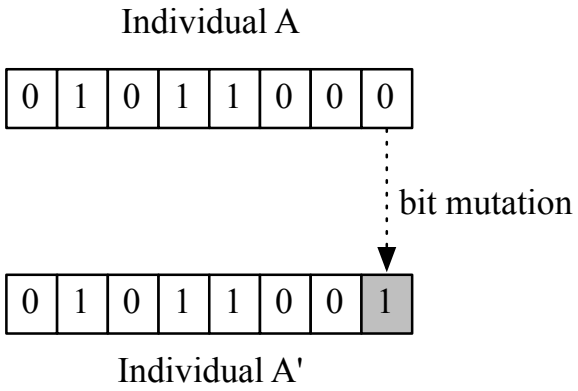


Figure 2.9: Flip mutation operator in a genetic algorithm.

specific tasks; in this work, the Hermes technique makes use of genetic programming. As Figure 2.10 illustrates, in a genetic program, an individual encodes a candidate executable program in a genome that comprises a set of operators (circles) and terminals (squares). Operators typically include mathematical and procedural functions, and terminals include variables and constant values. Each genetic program can be executed either directly on a computer, or on a virtual interpreted environment. For instance, the individual in Figure 2.10 can be executed to compute the value of the function $x^2 + x + \frac{x}{y} - x$.

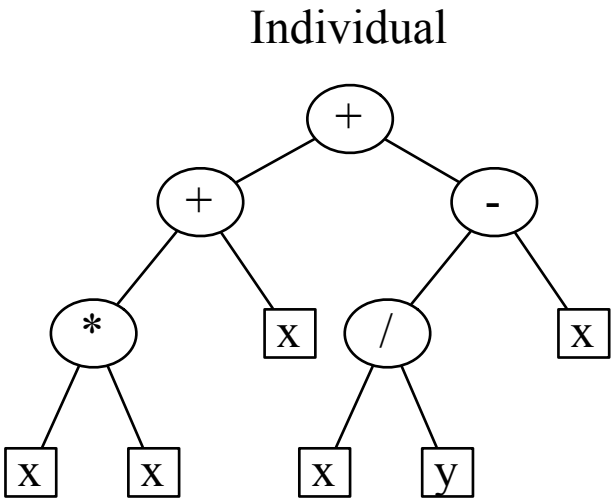


Figure 2.10: Tree-based representation of a genetic programming.

A genetic program begins by generating a population of random individuals. Three methods are commonly applied to generate random candidate programs: full, grow, and ramped half-and-half [63]. The *full* method generates a candidate program where the terminals, or leaves of a tree-based program representation, are all at the same depth level. The *grow* method generates a candidate program where the height of the tree-based representation can vary up to some maximum value. In contrast with the full and grow methods, which tend to not generate varied tree shapes, the *ramped half-and-half* method applies the full method to generate the first half of the population, and the grow method to generate the second half of the population. The resulting population stores the starting set of candidate individuals that will be explored and evaluated by the genetic program.

As with a genetic algorithm, a genetic program also applies a set of fitness functions to compute an individual's fitness value. In contrast to a genetic algorithm, a genetic program generally computes an individual's fitness value by *executing* the individual's encoded program. Furthermore, often the objective of a fitness function in a genetic program is to measure the difference between a candidate program's actual and expected outputs for a given set of input test cases. In traditional Koza fitness [63], the best possible fitness value that an individual can achieve is 0.0, and it implies that the evolutionary algorithm found a solution capable of producing exactly the expected output for every input test case considered.

Genetic programming applies two key operators to generate new candidate programs: crossover and mutation. In particular, the crossover operator exchanges genetic material, such as operators and terminals, from two existing individuals in the population to create a new individual that represents a potentially different solution from either parent. As Figure 2.11 illustrates, the crossover operator for a tree-based genetic program randomly selects two individuals from the population, and then it creates a new individual by replacing the subtree of one individual with a randomly

selected subtree from the other individual. Although not common in practice, this crossover operator can also produce another sibling by combining the genetic material not used for the first offspring.

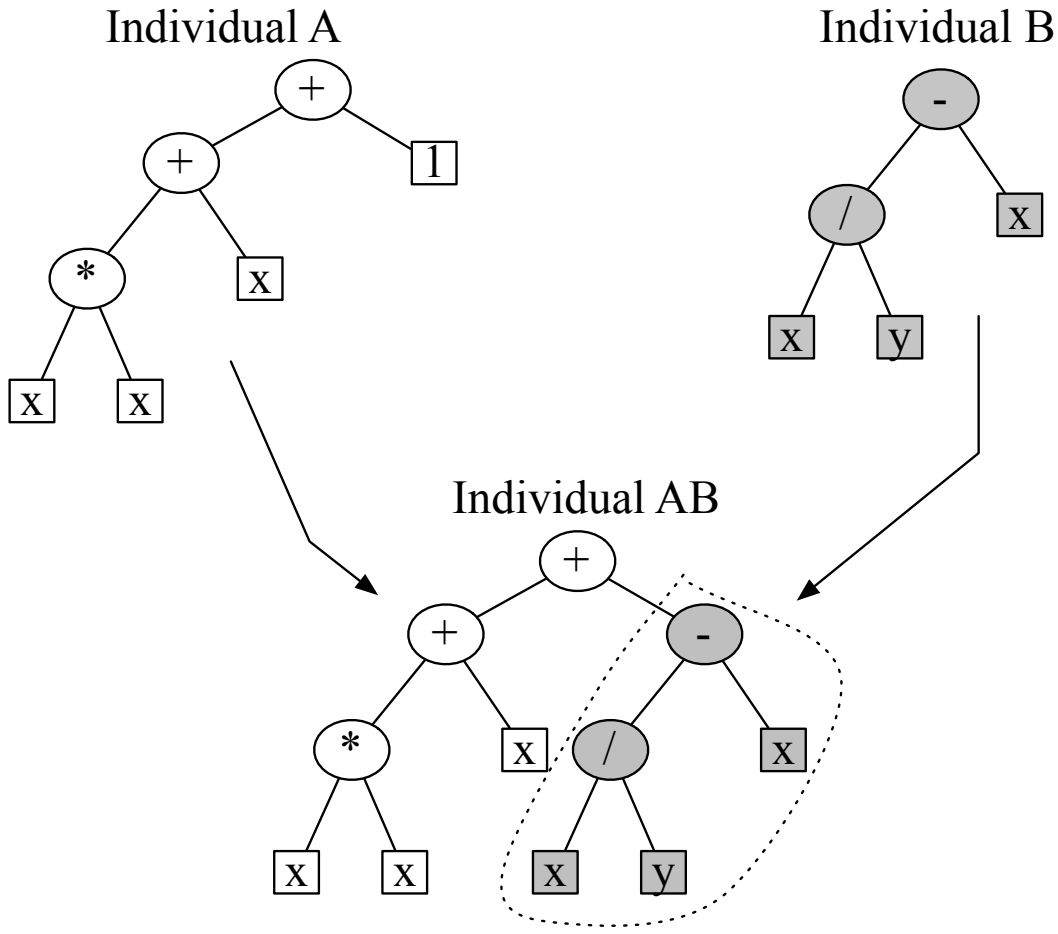


Figure 2.11: Crossover operator for a tree-based representation genetic program.

In addition to the crossover operator, a genetic program also applies several key mutation operators to generate new candidate programs from existing ones in the population. In general, genetic programming mutation operators operate on a random subtree element of an individual. As Figure 2.12 illustrates, mutation operators insert, remove, modify, or swap subtree elements to form new programs. In particular, insertion randomly adds new operators and terminals to a candidate program; removal randomly deletes a set of operators or terminals from a candidate

program; modification replaces a single operator or terminal in a candidate program; and swap exchanges operators or terminals within a candidate program.

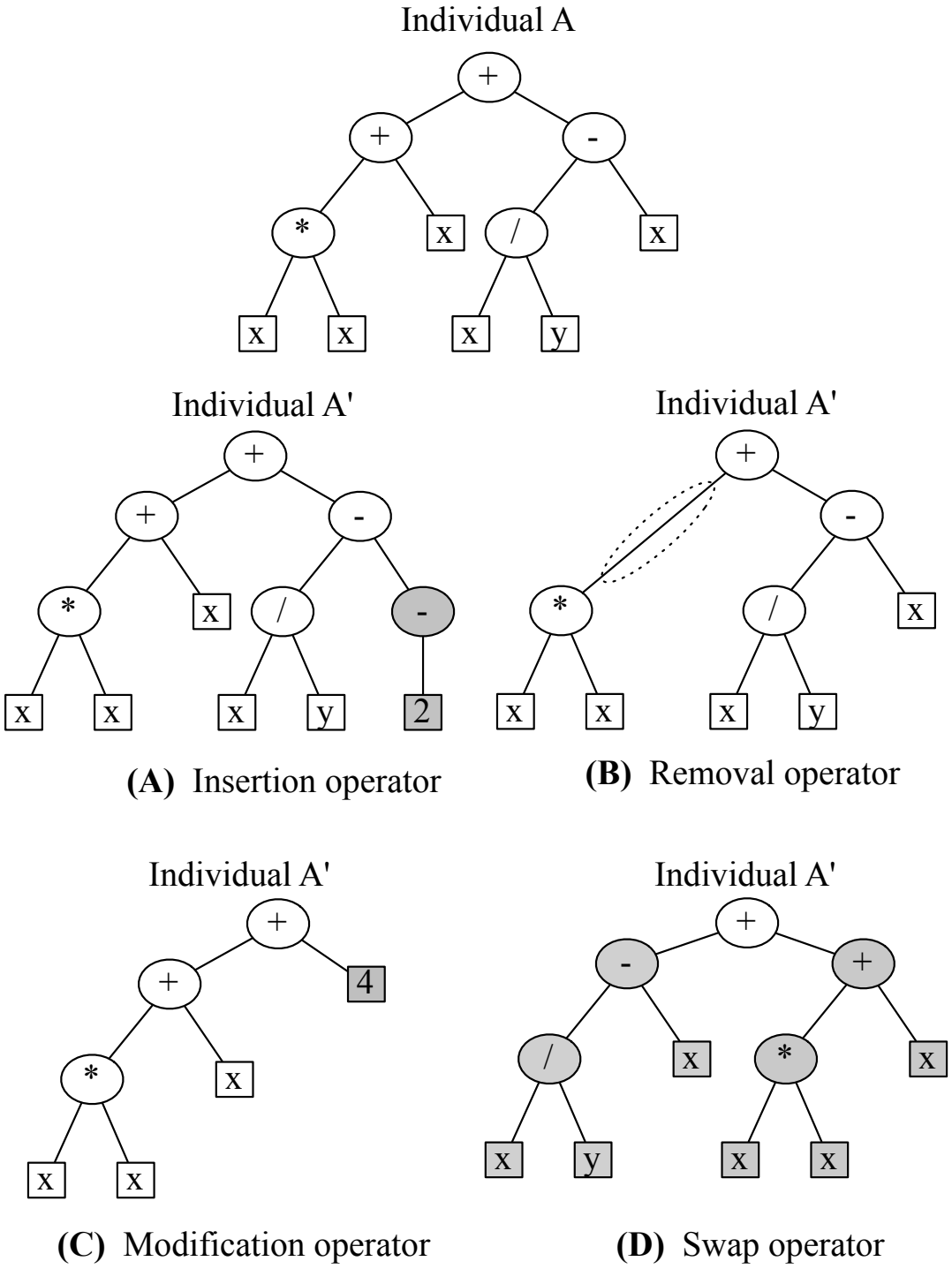


Figure 2.12: Insertion, removal, modification, and swap mutation operators for a tree-based genetic program.

A genetic program typically executes these operations for a fixed number of generations, or until the Koza fitness of a candidate program achieves a predetermined quality threshold value. Once the genetic program terminates, the individual with highest fitness is returned as the answer to the problem. This candidate program may be the direct solution to the problem being solved or, in the case of an interpreted virtual instruction set, may need to be translated into executable instructions for the specific target environment or machine.

2.4.3 Linear Genetic Programming

While a tree-based genomic representation is often used in genetic programming to facilitate a hierarchical evaluation of the program, other representations also exist. For instance, the Push-GP language [104] defines a simplified set of instructions that use type-specific stacks to structure and execute evolved programs. Likewise, as Figure 2.13 illustrates, linear genetic programming [9] encodes candidate genomes in a one-dimensional vector of virtual or interpreted instructions that can be sequentially executed from left to right. As this figure also illustrates, abstract instruction sets, such as the one used in this example where each operation corresponds to a DCM Protocol operation [64], can be remapped to executable instructions. By combining simple instruction sets with flexible execution environments both Push-GP and linear genetic programs tend to evolve modular programs that are often easier to analyze and understand than traditional tree-based representations.

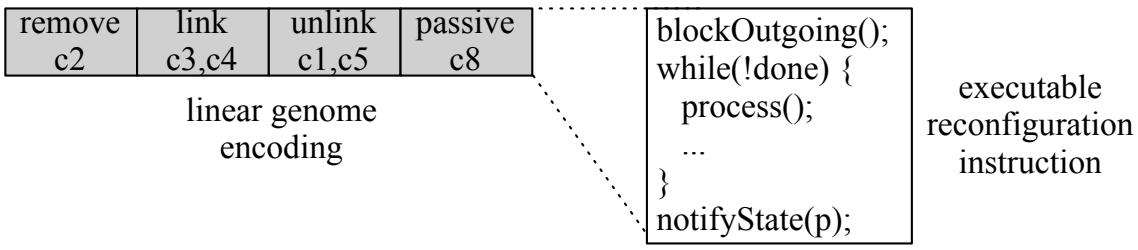


Figure 2.13: Linear genetic programming representation with DCM protocol instructions as instruction set.

As with traditional genetic programming, linear genetic programming also applies crossover and mutation operators to generate new candidate solutions. As Figure 2.14 illustrates, in a linear representation, two-point crossover executes as in a genetic algorithm, by first randomly selecting two individuals from the population as parents, A and B. Two indices, such as A_1, A_2 , B_1, and B_2, are then randomly selected from each parent to indicate the range of instructions that will be exchanged. These instructions are then swapped, creating two new individuals, AB and BA.

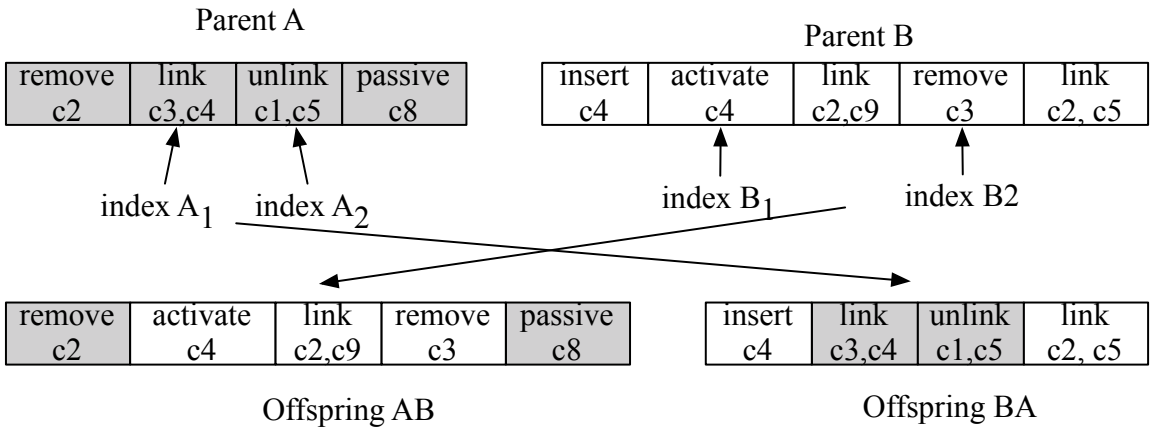


Figure 2.14: Two-point crossover in linear genetic program.

In contrast to the crossover operator, the *mutation* operator introduces variation into the population by randomly exploring points in the solution space that perhaps are not currently in the population [63]. To this end, mutation in linear genetic programming randomly inserts, removes, and replaces instructions and terminals in randomly selected genomes. Figure 2.15 illustrates an individual who is selected from the population and mutated. Specifically, in Figure 2.15, "`remove(c2)`" is replaced by "`insert(c4)`" and "`insert(c3)`" is inserted into the genome, thus producing A'. Ideally, the mutation operator introduces variation into the population that will form part of the overall solution.

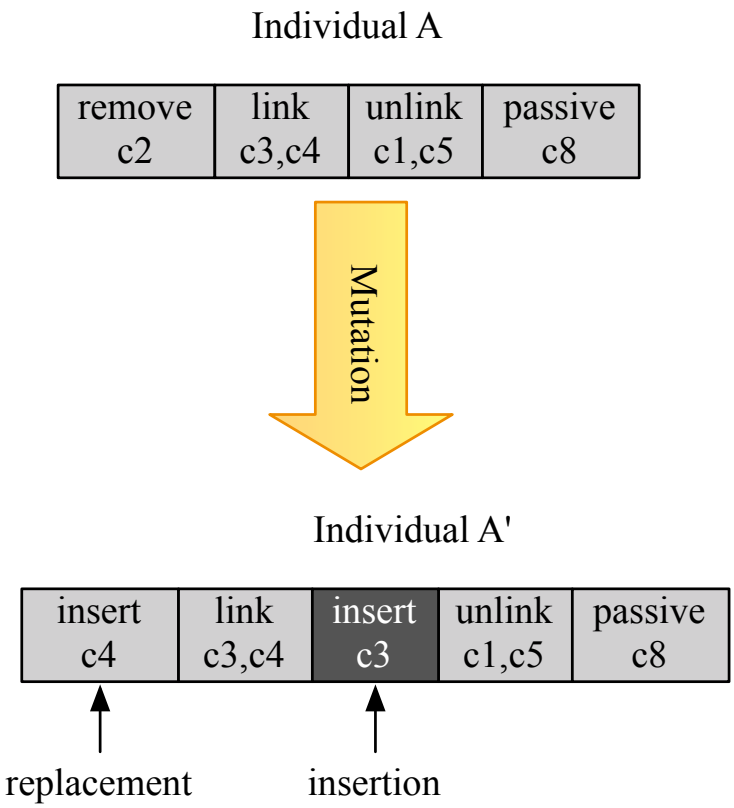


Figure 2.15: Mutation in linear-based genetic program.

Chapter 3

Automatic Derivation of Utility Functions for Requirements Monitoring

This chapter presents how our model-based framework supports requirements monitoring in a DAS. First, we motivate the need for automatically deriving utility functions for monitoring the satisfaction of functional, non-functional, and RELAXed requirements. We then introduce Athena, the component in our framework responsible for generating utility functions for requirements monitoring, and state its assumptions, inputs, and expected outputs. Next, we use a goal model that captures the requirements and constraints of the RDM application as an example to describe the process that Athena uses to generate utility functions. Subsequently, we apply Athena to the RDM application to show how derived utility functions quantify and relate different system and environmental conditions with specific requirements at run time. Lastly, we discuss and summarize main findings.

3.1 Motivation

Utility functions have been successfully applied for self-assessment purposes in a DAS [15, 20, 87, 109]. At run time, a DAS can use utility functions to assess how well it satisfies requirements, as well as identify conditions conducive to a requirements violation [28, 29, 97, 98]. As Figure 3.1 illustrates, within the context of a DAS, a utility function maps monitoring data to a scalar value, typically within the inclusive ranges of zero and one, that is proportional to how well the DAS satisfies its requirements. Significant drops in utility values often suggest undesirable conditions that may require adaptation. For example, the value of a utility function such as $\frac{Last_{ts}}{Current_{ts}}$, where $Last_{ts}$ is the last time an environmental property was measured and $Current_{ts}$ is the current time, will decrease as the gap between $Last_{ts}$ and $Current_{ts}$ increases, thus signaling the DAS to refresh its monitoring values. In contrast to traditional requirements monitoring approaches that use state-based model checkers [28, 29, 97], utility functions provide a light-weight alternative for associating the actions taken by a decision-making process with the high-level goals, concerns, and requirements of a DAS [20, 109].

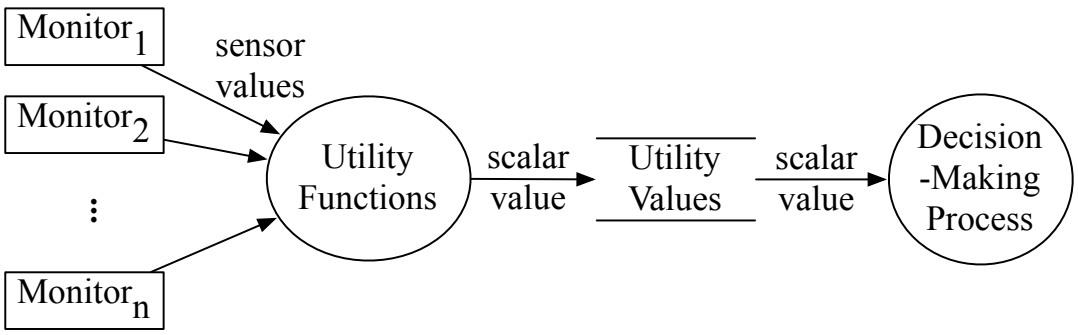


Figure 3.1: Utility functions within a DAS.

Typically, utility functions have been applied for monitoring both functional and non-functional requirements in DASs. To this end, a requirements engineer often elicits knowledge from domain experts to manually derive utility functions that can be used for monitoring functional requirements in a DAS [20]. Not only are these

ad hoc approaches for manually eliciting utility functions labor-intensive, but they may also fail to capture all functional requirements in a DAS. In contrast, statistical regression-based techniques can be used to indirectly infer, at run time, utility functions for monitoring non-functional requirements in a DAS [12, 20, 80, 116]. These *performance-based* utility functions often generate a single application-level utility value representative of the overall system's performance. While these statistics-based approaches facilitate the automatic derivation of performance-based utility functions, they tend to postpone their integration with the DAS until deployment, when real execution data becomes available to drive the regression process.

A goal model can be used during the requirements engineering phase to identify, elaborate, and refine the set of requirements and constraints that a DAS must satisfy. Perhaps with the exception of requirements-aware systems [5, 6, 103, 111], goal models are not commonly used after the requirements engineering and early design phases. Nevertheless, a goal model contains valuable information that can be leveraged at run time to determine when and how a DAS should self-reconfigure in response to system and environmental changes. As a result, new goal-oriented model-based techniques are needed to support the systematic monitoring of functional, non-functional, and RELAXed requirements.

3.2 Introduction to Athena

Athena is a component in our model-based framework that uses a goal-oriented requirements model to automatically derive utility functions for requirements monitoring. Derived utility functions directly support self-assessment in a DAS at the requirements level, using the abstracted goal model as its evaluation reference point. Moreover, since Athena operates at the requirements level, it facilitates the integration of derived utility functions throughout the entire development process of the DAS.

Lastly, by incorporating fuzzy logic functions, Athena also supports the automatic derivation of utility functions for monitoring the satisfaction of RELAXed goals. The resulting utility functions can be further augmented and refined with the aid of a domain expert, if necessary.

Athena supports the automatic derivation of state-, metric-, and fuzzy logic-based utility functions from a KAOS goal model [18, 105] that may also include RELAXed goals [13, 114]. State-based utility functions assess whether a DAS satisfies functional invariant goals. Metric-based utility functions, on the other hand, detect conditions that may be conducive to a requirements violation, ideally enabling a DAS to mitigate such conditions before an invariant goal becomes unsatisfied. Lastly, fuzzy logic-based utility functions compute the satisfaction of non-invariant goals that have been RELAXed to explicitly account for the effects of uncertainty. Collectively, the values produced by these utility functions enable a DAS to monitor the satisfaction of requirements, determine when an adaptation is warranted, and identify sets of goals and agents that may be responsible for a requirements violation.

Athena adopts a bottom-up approach to derive utility functions from a goal model of the DAS. In particular, Athena analyzes a goal's definitions and keywords to identify and map environmental conditions (ENV properties) to their corresponding monitoring elements (MON). Next, Athena extracts keywords for each goal's specification to select a corresponding utility function *template* to evaluate that particular goal. These utility function templates are conceptually similar to functional template constructs provided by programming languages and provide generic and reusable structures for evaluating the satisfaction of goals according to the semantics associated with keywords in the goal's definition.

Athena uses different utility function templates for evaluating different goal types. Namely, we defined state-based utility function templates that preserve the semantics of linear temporal logic to evaluate the satisfaction of invariant goals [105]. We

also defined metric-based utility function templates to measure the degree to which an observable condition in the goal's formulation is minimized or maximized with respect to a given threshold. Lastly, we defined fuzzy logic-based utility function templates to measure the satisfaction of RELAXed goals [114]. Once Athena selects the corresponding utility function template for a goal, it then uses the ENV and MON mappings to instantiate each of these utility function templates such that they refer to specific system and environmental properties.

3.3 Athena Process

This section states the expected inputs and outputs of Athena, as well as describes each step that Athena applies to generate utility functions.

3.3.1 Expected Inputs and Outputs

Athena requires two input elements to generate utility functions for requirements monitoring in a DAS: a KAOS goal model of the DAS that may include RELAXed goals, and a set of ENV properties, MON elements, and REL relationships. Next, we describe how Athena uses each of these inputs.

KAOS Goal Model. Athena uses a KAOS goal model [18, 105] that captures the objectives and constraints of the DAS to derive utility functions. For example, Figure 3.2 shows a KAOS goal model with RELAXed goals for the RDM application, where RELAX goals are shown in uppercase. In particular, this goal model captures a set of goals for establishing and maintaining a connected network of RDMs, as well as protecting critical data by replicating and distributing copies of data to data mirrors throughout the network. As this goal model shows, the data mirrors can distribute data across the RDM network by using either synchronous or asynchronous propagation methods. This goal model also captures the various adaptation objectives

of the RDM network, such as maximizing the number of active data mirrors while minimizing the number of passive and quiescent data mirrors.

In contrast to the goal model previously presented in Figure 2.4, this model includes several RELAXed goals to both explicitly address identified sources of uncertainty as well as demonstrate how Athena can derive utility functions for evaluating the satisficement of RELAXed goals. For instance, Goal (F) was RELAXed since the RDM network can temporarily tolerate network partitions without necessarily violating Goal (A). In contrast, Goals (A) and (B) denote invariants that were not RELAXed as temporary violations of these goals are not acceptable since they would either defeat the purpose of remote data mirroring or the constraints under which the system must protect data, respectively. Furthermore, each goal must also be classified either as an invariant or a non-invariant goal.

Athena requires that the textual definition of each KAOS goal conforms to the extended BNF grammar [3] presented in Figure 3.3. This grammar, based on the KAOS grammar introduced by van Lamsweerde *et al.* [18, 105], states that a KAOS goal comprises exactly one KAOS keyword followed by one ENV property and zero or one logical operator and constraint. Note that ENV properties and constraints are defined here as placeholder strings; both are application domain-dependent and must be specified by a requirements engineer. Lastly, a constraint can be defined either as a fixed threshold, such as a physical property, or in terms of an ENV property that may change after the DAS is deployed. For example, in Goal (B), `Maintain` is the KAOS keyword, `OperationalCosts` is the ENV property, <= is the logical operator, and `Budget` is the constraint.

Likewise, Athena also requires that the textual definition of each RELAX goal conforms to the extended BNF grammar presented in Figure 3.4. This grammar, defined based on the RELAX operators and their semantics [13, 114], states that RELAXed goals can be expressed either by exactly one KAOS keyword, an ENV property, a

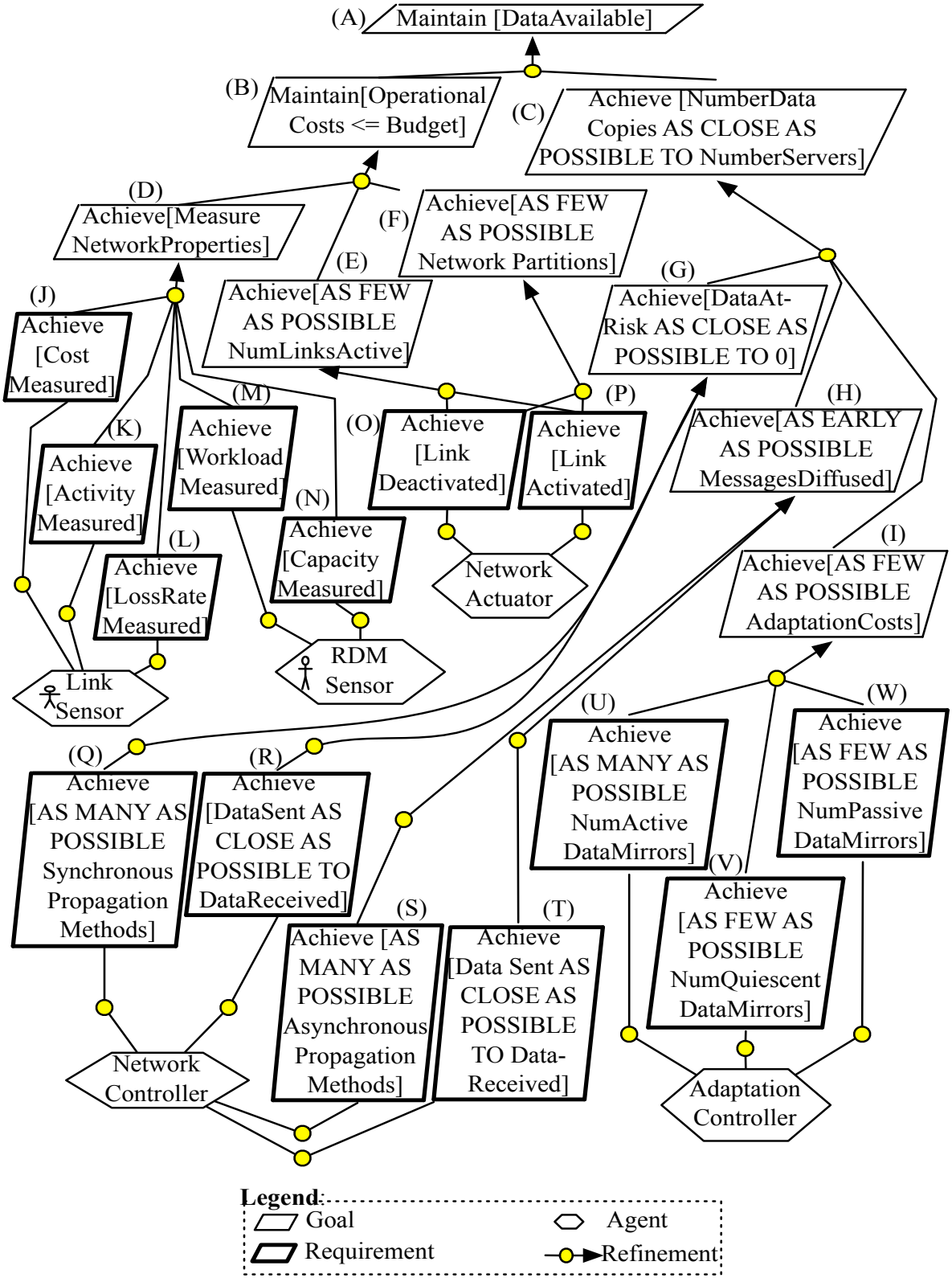


Figure 3.2: RELAXed goal model for RDM application.

```
KAOS Goal        :: KAOS Keyword,  (ENV Property,  Logical Operator?,  Constraint?)
KAOS Keyword     :: "Achieve" | "Avoid" | "Maintain"
ENV Property     :: String, { String }
Logical Operator :: > | <= | >  | >= | == | !=
Constraint       :: ENV Property | String, { String }
String           :: " Char, { Char } "
Char             :: Letter | Digit
Letter           :: a | b | … Y | Z
Digit            :: 0 | 1 | … | 9
```

Figure 3.3: Specification grammar for KAOS goals.

RELAX operator, and a constraint, or by one KAOS keyword, a RELAX operator, and an ENV property. For example, Goal (C) is expressed using the first pattern, where `Achieve` is the KAOS keyword, `NumberDataCopies` is the ENV property, `AS CLOSE AS POSSIBLE TO` is the RELAX operator, and `NumberServers` is the constraint. Goal (F), on the other hand, is expressed using the second pattern, where `Achieve` is the KAOS keyword, `AS FEW AS POSSIBLE` is the RELAX operator, and `NetworkPartitions` is the ENV property.

```
RELAX Goal       :: KAOS Keyword, ENV Property, RELAX Operator, Constraint |
                    KAOS Keyword, RELAX Operator, ENV Property
KAOS Keyword     :: "Achieve" | "Avoid" | "Maintain"
RELAX Keyword    :: "AS EARLY AS POSSIBLE" | "AS CLOSE AS POSSIBLE TO" |
                    "AS LATE AS POSSIBLE" | "AS FEW AS POSSIBLE" |
                    "AS MANY AS POSSIBLE"
ENV Property     :: String, { String }
Logical Operator :: > | <= | >  | >= | == | !=
Constraint       :: ENV Property | String, { String }
String           :: " Char, { Char } "
Char             :: Letter | Digit
Letter           :: a | b | … Y | Z
Digit            :: 0 | 1 | … | 9
```

Figure 3.4: Specification grammar for RELAX goals.

ENV Mappings. In addition to the KAOS goal model of the DAS, Athena also requires a set of ENV properties, MON elements, and REL relationships. In particular, ENV specifies environmental conditions observable by the DAS; MON elements specify the environmental agents, or sensors, that make up the DAS's monitoring infrastruc-

ture; and REL specifies how to compute the values of ENV properties from MON elements. Note that ENV properties can be either directly observed by MON elements, or it can be derived from an REL relationship consisting of other ENV properties or MON elements. To this end, each REL relationship must preserve the dependencies between MON elements and the types of data required by the corresponding computations.

Table 3.1 presents a subset of these ENV, MON, and REL elements that are relevant to Goal (B) and its subgoals in the RDM goal model. For instance, Rows (1) and (2) specify that in Goals (J) and (K), the ENV properties `OperationalCost` and `ActivityStatus` can be directly measured by the MON element `LinkSensor`, respectively. Note that not all ENV properties can be directly observed by a single MON element and instead require more complex calculations that either manipulate or aggregate the values of individual MON elements. For example, calculating the value of ENV property `NetworkPartitions` in Goal (F) requires applying a graph-based reachability algorithm to detect the total number of connected components in the RDM network [22]. Likewise, calculating the value of ENV property `OperationalCosts` in Goal (B) requires summing the cost of each active network link.

Table 3.1: Table with ENV, MON, and REL elements for RDM application.

Row	Goal	ENV	MON	REL
1	J	Cost	LinkSensor	LinkSensor.cost
2	K	Activity Status	LinkSensor	LinkSensor.is_active
3	L	Loss Rate	LinkSensor	LinkSensor.loss_rate
4	M	Workload	RDMSensor	RDMSensor.workload
5	N	OperationalCapacity	RDMSensor	RDMSensor.capacity
6	E	NumLinksActive	All Link Sensors	Sum(LinkSensor.is_active)
7	F	NetworkPartitions	LinkSensor, RDMSensor	|ConnectedComponents(RDMSensor, LinkSensor if LinkSensor.is_active)| - 1
8	C	NumberServers	All RDMSensors	Sum(RDMSensor.is_active)
9	B	OperationalCosts	LinkSensors	Sum(LinkSensor.cost if LinkSensor.is_active)

As output, Athena produces a set of state-, metric-, and fuzzy logic-based utility functions by instantiating different function templates based on the goal's type. As with traditional requirements monitoring approaches [28, 29], Athena generates

state-based functions to monitor functional invariant goals since their *satisfaction* can be determined in a crisp, true or false manner, such as "Has the RDM network ever exceeded the allocated operational budget?". In addition, Athena also generates metric- and fuzzy logic-based utility functions to monitor the *satisficement* of non-invariant goals. Although not commonly done by traditional requirements monitoring approaches, when possible, Athena also generates a metric-based utility function to evaluate the degree to which an invariant goal is satisficed; this satisficement information can enable a DAS to detect when the violation of an invariant goal is imminent.

3.4 Utility Function Derivation Process

The data flow diagram in Figure 3.5 shows the process that Athena applies in order to generate a *single* utility function for a given KAOS or RELAX goal. In this data flow diagram, we added boolean guards (denoted in bold font) to explicitly capture the alternate data flows within the derivation process based on the type of goal being processed. Athena begins the utility function derivation process at each leaf goal and, following a bottom-up approach, applies the process depicted in Figure 3.5 to each goal in the model until the root goal is reached. This bottom-up approach enables Athena to incorporate how the satisfaction of subgoals ultimately affect the satisfaction of their parent goals. Next, we describe each of the key steps of the Athena process.

(1) Map ENV property. An ENV property is a condition of the execution environment that can be observed or inferred by a DAS through its monitoring infrastructure [13, 114]. Athena uses the KAOS and RELAX grammars presented in Figures 3.3 and 3.4 to parse a goal's definition to determine if a textual element refers to an ENV property. Specifically, Athena matches textual elements in a goal's specification with the set of ENV properties previously identified by a requirements engineer

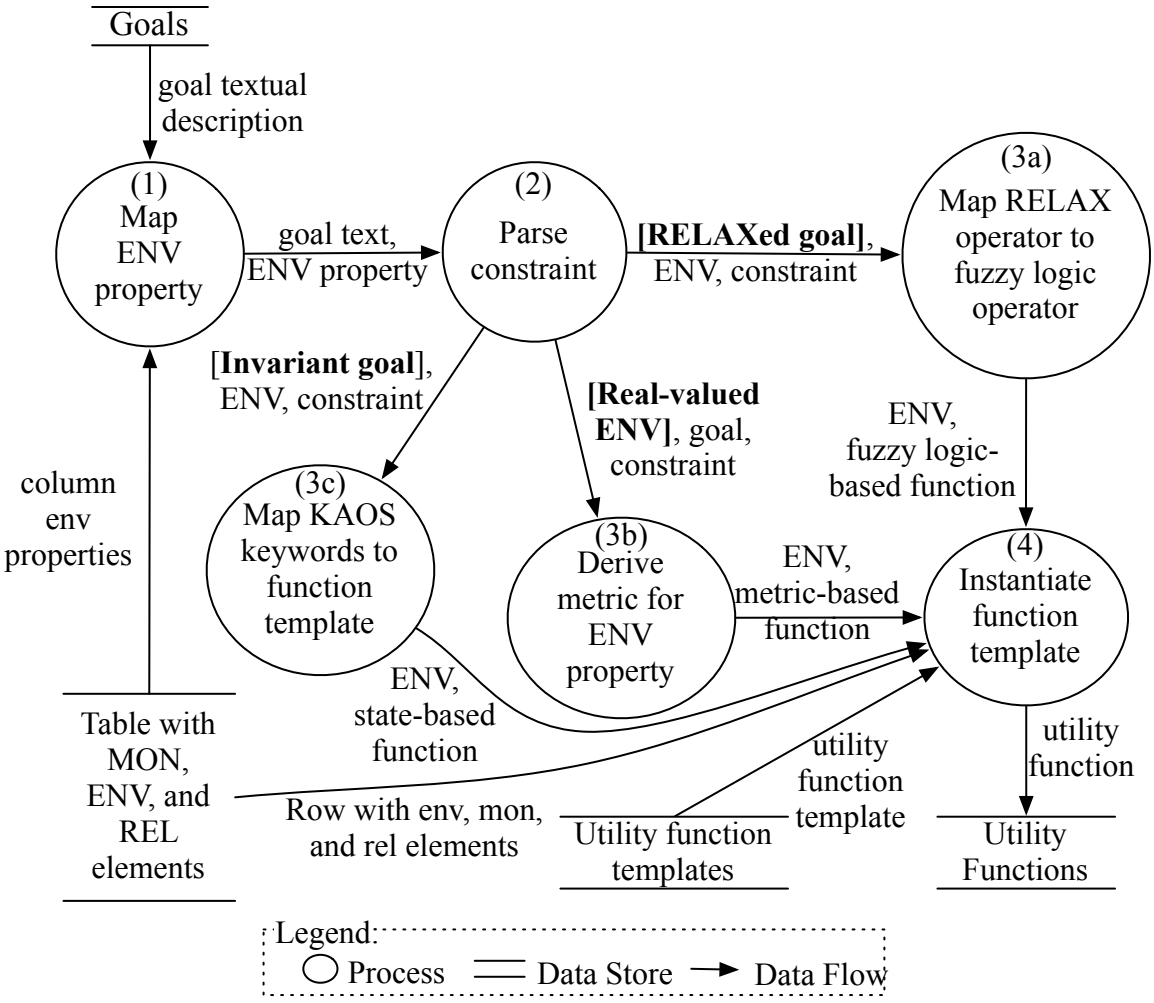


Figure 3.5: Data flow diagram illustrating how Athena generates a single utility function for a given KAOS or RELAXed goal.

while applying the RELAX specification process. For example, Goal (B) in Figure 3.2 contains the textual token `OperationalCosts` that is specified as an ENV property in Row 8 of Table 3.1. Note that not all goals refer to an ENV property. For instance, although Goal (D) specifies that `NetworkProperties` must be measured, this textual token is not specified as an ENV property in Table 3.1. If a goal does not refer to an ENV property, then Athena proceeds to step (5).

(2) Parse goal constraints. A constraint is often a logical condition or threshold that can be evaluated in a crisp fashion (i.e., true or false). A goal may specify

either an *absolute* constraint, such as a fixed threshold, or a *relative* constraint that states a relationship between properties whose values may change at run time, such as an ENV property. To this end, Athena parses the textual description of a goal to extract the specified constraint, if any. If the goal specifies a constraint, then Athena attempts to match the textual token with an ENV property in the table of ENV, MON, and REL elements. For example, Goal (C) specifies a relative constraint, `NumberServers`, that appears in Row 8 of Table 3.1. As the REL entry for this ENV property shows, the value of `NumberServers` depends upon the configuration of the RDM network and can be measured by aggregating the number of active RDMs in the network. Athena proceeds to step (5) if a goal does not specify a constraint or the constraint does not appear in the table of ENV, MON, and REL properties.

(3a) Map RELAX operator to fuzzy logic-based utility function. RELAX defines a set of operators to constrain how a non-invariant goal may become temporarily unsatisfied at run time due to environmental uncertainty [13, 114]. Athena uses the RELAX grammar previously presented in Figure 3.4 in order to determine if a requirements engineer RELAXed a KAOS goal. To achieve this objective, Athena matches textual elements in a goal's specification with the set of RELAX operators. Each RELAX operator, in turn, is associated with a fuzzy logic-based utility function template that evaluates the degree to which a non-invariant goal is *satisficed*. While Athena instantiates the corresponding utility function with parameters extracted from the goal's definition, in some cases the specific bounds that define the satisfaction criteria must be specified by the requirements engineer after the derivation process completes.

Currently, Athena leverages three utility function templates to measure the satisficement of RELAXed goals. Each utility function template models a different fuzzy logic function shape, such as triangle, left shoulder, and right shoulder shapes. Moreover, each utility function template can be instantiated either with ordinal or tem-

poral property types, thereby supporting all operators defined thus far in the RELAX specification language [114].

A triangle-shaped fuzzy logic function is often associated with the semantics of the ordinal and temporal "AS CLOSE AS POSSIBLE TO" RELAX operators [13, 114]. As such, Athena uses the triangle-shaped utility function shown in Figure 3.6(A) in order to evaluate the satisficement of KAOS goals that have been RELAXed with this RELAX operator. This utility function template maps a measured property, either ordinal or temporal, to a triangular-shaped function curve whose output values range from zero to one, inclusive. Once instantiated, as in Figure 3.6(B), this utility function produces values that approach one as the measured property approaches its constraint or desired value. In contrast, this utility function produces values that linearly approach zero as the measured property diverges from its constraint or desired value, eventually returning zero once the measured property exceeds the minimum or maximum values allowed.

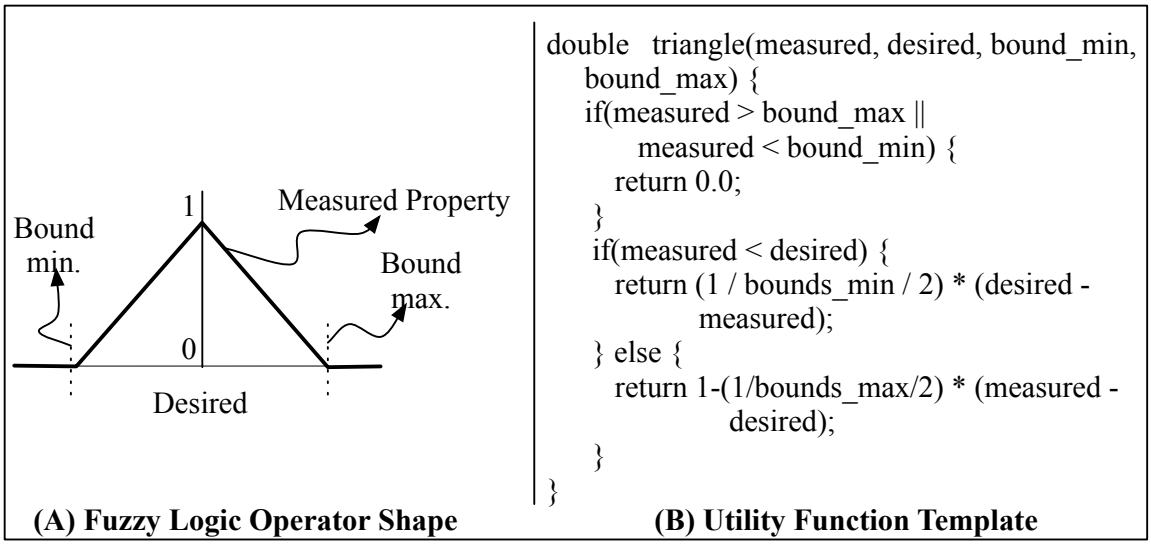


Figure 3.6: Triangle shape fuzzy logic operator and its corresponding utility function template.

A left shoulder-shaped fuzzy logic function is often associated with the semantics

of the "AS EARLY AS POSSIBLE" and "AS FEW AS POSSIBLE" RELAX operators [5, 114]. Therefore, Athena uses the left shoulder-shaped utility function shown in Figure 3.7(A) to evaluate the satisficement of KAOS goals that have been RELAXed with either of those two RELAX operators. This utility function template maps the value of a measured property, either ordinal or temporal, to a linearly decreasing function curve. Once instantiated, as in Figure 3.7(B), this utility function produces values that approach one as the measured property approaches a time or quantity equal to 0. In contrast, this utility function produces values that linearly approach zero as the measured property increases, eventually returning zero once the measured property exceeds the maximum value allowed.

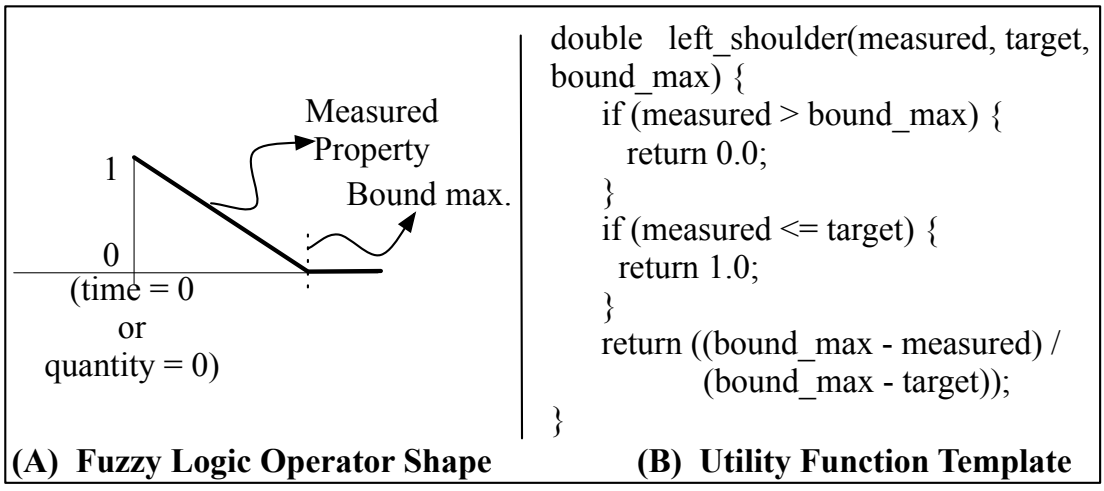


Figure 3.7: Left shoulder shape fuzzy logic operator and its corresponding utility function template.

Lastly, a right shoulder-shaped fuzzy logic function is often associated with the semantics of the "AS LATE AS POSSIBLE" and "AS MANY AS POSSIBLE" RELAX operators [5, 114]. Athena uses the right shoulder-shaped utility function shown in Figure 3.8(A) to evaluate the satisficement of KAOS goals that have been RELAXed with either of those two RELAX operators. This utility function template, shown in Figure 3.8(B), maps the value of a measured property, either ordinal or temporal,

to a linearly increasing function curve. Note that this utility function returns values that approach one as the measured property approaches a time or quantity that is greater than the minimum value allowed and zero for measured property values less than the minimum value allowed.

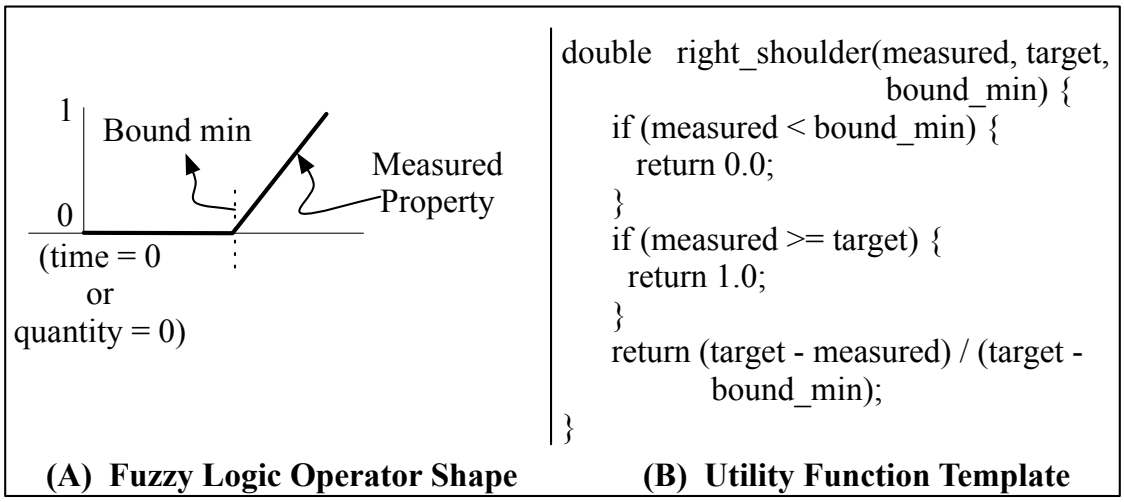


Figure 3.8: Right shoulder shape fuzzy logic operator and its corresponding utility function template.

As an example, consider Goal (F) from the RDM goal model in Figure 3.2. This goal has been RELAXed by applying the "AS FEW AS POSSIBLE" RELAX operator. As a result, to evaluate the satisficement of Goal (F) at run time, Athena applies the left shoulder-shaped utility function template presented in Figure 3.7(B). Athena instantiates this utility function template by assigning the ENV property `NetworkPartitions` to the `MeasuredQuantity` variable, and the absolute threshold of zero to the `DesiredQuantity` constraint. This utility function measures the satisficement of RELAXed Goal (F) by explicitly capturing the extent to which the RDM can tolerate temporary network partitions, as established by the specified `bound_max` constraint.

(3b) Derive a metric for a real-valued ENV property. Athena generates a metric-based utility function to measure the *satisficement* of both invariant and

non-invariant goals. Although it is possible to determine whether an invariant goal is *satisfied*, Athena *also* evaluates the degree to which an invariant goal is satisficed when the goal refers to a real-valued ENV property. Treating an invariant goal as a non-invariant goal enables a DAS to detect and mitigate conditions conducive to a requirements violation. To achieve this objective, Athena uses a goal's fitness criterion, which is an annotation often associated with soft goals to quantify the extent to which a goal should be met [105]. Athena maps keywords in this annotation (e.g., `Minimize` and `Maximize`) to a utility function template that either minimizes or maximizes the divergence between an ENV property and its constraint or threshold. In particular, the following utility function template measures the degree to which an ENV property equals a given constraint, $\text{Val}_{constraint}$:

$$\text{UT}_{\text{minimize}} = 1 - \min \left\{ \frac{|\text{Val}_{ENV} - \text{Val}_{Constraint}|}{\text{Val}_{ENV}}, 1 \right\} \tag{3.1}$$

and the following utility function template measures the degree to which an ENV property diverges from a given constraint, $\text{Val}_{constraint}$:

$$\text{UT}_{\text{maximize}} = \min \left\{ \frac{|\text{Val}_{ENV} - \text{Val}_{Constraint}|}{\text{Val}_{ENV}}, 1 \right\} \tag{3.2}$$

For example, consider Goal (C) from the RDM goal model in Figure 3.2 that specifies that each data item should be stored in each data mirror in the network. This goal is a non-invariant because it takes the RDM network a certain amount of time to diffuse data items to all data mirrors, thus it cannot always be satisfied as stated. As such, Athena measures the satisficement of this goal by deriving a metric-based utility function that measures the degree to which the RDM minimizes the difference between `NumberDataCopies` and `NumberServers`. In particular, function template (3.1) can be instantiated as follows:

$$\mathrm{UT_C} = 1 - \min \left\{ \frac{|\mathrm{NumberDataCopies} - \mathrm{NumberServers}|}{\mathrm{NumberDataCopies}}, 1 \right\} \qquad (3.3)$$

This utility function evaluates how close the RDM network is replicating each data item at each data mirror by producing values inversely proportional to the difference between `NumberDataCopies` and `NumberServers`.

(3c) Derive state-based utility function for an invariant goal. An invariant goal describes functionality that the system-to-be must *always* provide. As part of the RELAX approach, a requirements engineer classifies goals as an invariant or non-invariant. To derive a state-based utility function for an invariant goal, Athena first parses the goal's definition and identifies which KAOS keyword it contains. Each keyword (i.e., `Achieve`, `Avoid`, and `Maintain`) can be mapped to precise semantics in temporal state-based logic [105]. Athena maps these keywords to a state-based utility function template that returns true or false depending on whether the constraint is satisfied or not. Figure 3.9 presents the three state-based utility function templates that Athena uses to generate utility functions for invariant goals. For example, Athena instantiates the utility function template in Figure 3.9(A) to monitor the satisfaction of `Maintain` goals, where ENV refers to the environmental condition identified in step (1), `Op` refers to a logical operator, such as $<$ and $=$, and `Constraint` refers to the goal's constraint identified in step (2). This template uses a *satisfied* guard to preserve the semantics of a `Maintain` goal and thus returns true if and only if the constraint has always been satisfied.

(4) Instantiate utility function template. Each utility function accepts monitoring information from MON elements (i.e., sensors) in order to evaluate the satisfaction or satisficement of goals at run time. Athena uses the set of ENV properties, MON elements, and REL relationships to express each utility function solely in terms of MON elements. In particular, Athena replaces each ENV term with the set of MON elements responsible for computing its value. For example, Athena replaces the ENV

```
// (A) Maintain Utility Function Template:
boolean maintain_template(ENV, Op, Constraint) {
   if(satisfied) {
      return (satisfied = Op(ENV, Constraint));
   }
   return false;
}

// (B) Achieve Utility Function Template:
boolean achieve_template(ENV, Op, Constraint) {
   return Op(ENV, Constraint);
}

// (C) Avoid Utility Function Template:
boolean avoid_template(ENV, Op, Constraint) {
   if(!satisfied) {
      return !(satisfied = Op(ENV, Constraint));
   }
   return false;
}
```

Figure 3.9: State-based utility function templates for Achieve, Avoid, and Maintain goals.

property term `OperationalCosts` in the utility functions derived for Goal (B) with the following expression:

$$OperationalCosts = \sum_{i=0}^{n} LinkSensor.cost * LinkSensor.is\_active \qquad (3.4)$$

This expression, shown in Row (8) of Table 3.1, specifies that the value of `OperationalCosts` can be computed by summing the operational cost of each active network link. Here, `is_active` returns 1 (True) if a network link is active and 0 (False) otherwise.

(5) Propagate utility values to parent goal. Athena propagates the utility value associated with a goal, if any, to its parent goal. To a parent goal, this propagated utility value measures how well its subgoals are satisficed. The utility values of multiple subgoals are combined in different ways depending on the type of goal

refinement applied. For an AND-decomposition, Athena computes the product of each utility value reported by the subgoals in the refinement. For instance, if the utility value associated with Goals (D), (E), and (F) are 1.0, 0.8, and 1.0, respectively, then, from the perspective of Goal (B), its subgoals are satisfied to a degree of 0.8. In contrast, for an OR-decomposition, Athena selects the *maximum* value of each utility value produced by the subgoals in the OR-refinement. For example, if the utility value associated with Goals (Q) and (R) are 0.8 and 0.9, respectively, then from the perspective of Goal (G), its subgoals are satisficed to a degree of 0.9. These semantics capture the notion that to satisfy a goal that has been AND-decomposed all subgoals must be satisfied, whereas to satisfy a goal that has been OR-decomposed, at least one subgoal must be satisfied.

(6) Repeat steps (1) through (5) until the root goal is reached from every starting leaf goal.

3.5 Case Study

This section presents two experiments to illustrate how utility functions derived by Athena can be used for requirements monitoring in a DAS. First, we describe the experimental setup used throughout each RDM simulation. We then present results that show how derived utility functions capture the satisfaction of invariant, non-invariant, and RELAXed goals at run time in response to different system and environmental conditions.

The following two experiments simulate a network that comprises 25 RDMs and 300 network links that can be activated to distribute data. Each RDM simulation executes for 300 time steps during which 150 new data items must be diffused across the network, where each data item is randomly inserted at different time steps and RDMs. Note that new data is inserted only during the first 80% of the simulation

length to provide a sufficiently reasonable amount of time for the RDM network to diffuse messages before simulation completes.

Overall, the following experiments illustrate how derived utility functions capture requirements satisfaction in two completely different scenarios. Specifically, the first experiment does not subject the RDM to any adverse system or environmental condition, thereby facilitating the data replication and distribution process. In contrast, the second experiment introduces various forms of system and environmental uncertainty, such as repeated network link failures, dropped messages, and so forth. Since each simulation contains a random component, we performed 30 replicate simulation trials for each scenario and plot mean values with error bars where applicable to show statistical significance.

3.5.1 No Adverse Environmental Conditions

Experimental Objective. This experiment, Experiment 3.1, verifies that Athena derives utility functions that can detect how well the RDM network satisfies its requirements under ideal system and environmental conditions. By the end of each simulation, derived utility functions should report that all data messages have been diffused without exceeding the allocated operational budget.

Hypothesis. For this experiment, we define a null hypothesis, H_0, that states that *utility functions derived by* Athena *will not detect any unsatisfied requirements.* Here, an unsatisfied invariant requirement appears in the form of utility values that are not equal to 1. Likewise, an unsatisfied non-invariant requirement appears in the form of utility values equal to 0.

Configuration. Since this experiment does not subject the RDM network to adverse system and environmental conditions, the primary sources of uncertainty are *when* and *where* data messages are inserted into the RDM network. Table 3.2 specifies the base configuration used for this scenario, where "..." specifies a range of

inclusive values. As this table illustrates, the random number generator (RNG) uses different seed values for each simulation and produces values according to a normal distribution. Within the context of this experiment, a normal distribution tends to introduce new data messages within the mid-stages of the simulation length and within a particular subset of data mirrors.

Table 3.2: Configuration for simulation without uncertainty.

Property	Value
Seed	1...30
Distribution	Normal
Number Data Mirrors	25
Underlying Network Topology	Complete
Budget	$500000.00
Base Data Mirror Capacity	6.0 Gb
Data Mirror Capacity Variance	0.25
Base Network Link Bandwidth	7.0 Gb per time step
Network Link Bandwidth Variance	0.25
Base Data Message Size	2.0 Gb
Data Message Size Variance	0.25
Probability Data Mirror Failure	0.0
Probability Network Link Failure	0.0
Probability Data Message Drop	0.0
Probability Data Message Delayed	0.0
Probability Data Message Corrupted	0.0
Probability Data Mirror Sensor Failure	0.0
Probability Sensor Fuzz	0.0

Table 3.2 also shows that the processing capacity for each data mirror can vary by up to 25% of its base processing capacity. Likewise the bandwidth and cost of each network link can also vary by up to 25% of its base values. In addition to when and where a data message is inserted, a data message can also vary in size by up to 25% of its base size. Lastly, the likelihood of all sources of system and environmental uncertainty are explicitly set to 0% , thereby disallowing their occurrence throughout the simulation.

Results. Since both Goals (A) and (B) are invariant goals in Figure 3.2, Athena

derived state-based utility functions to evaluate their satisfaction during each RDM simulation. Together, these two goals specify that the RDM network should always maintain data available while never exceeding the allocated operational budget. As Figures 3.10 and 3.11 respectively show, both Invariant Goals (A) and (B) were always satisfied across all 30 simulation trials. In addition, Athena also derived a metric-based utility function for Invariant Goal(B) since both the ENV property and its corresponding constraint refer to floating-point values. As such, we also plot the satisficement, or degree of satisfaction, of Goal (B) in Figure 3.11 to capture the extent to which the RDM network satisfied this constraint. As the metric-based utility values show, the RDM network managed to satisfice Goal (B) to approximately a 55% of its maximum potential value, thereby implying that the RDM network activated redundant network links to improve data diffusion performance and reliability.

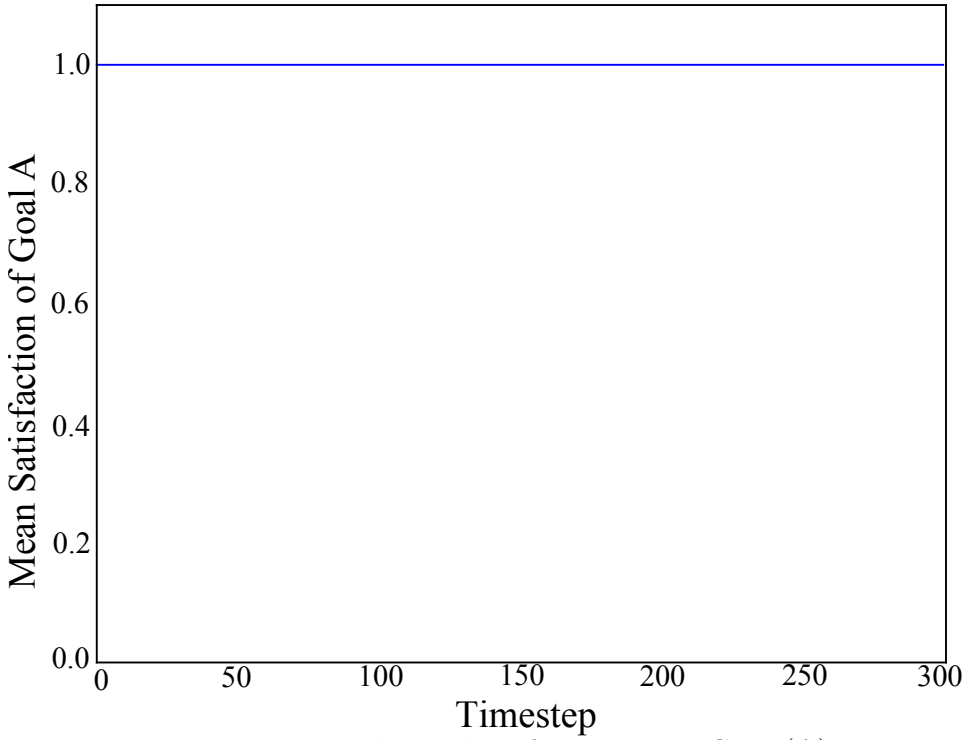


Figure 3.10: Utility values for Invariant Goal (A)

Figure 3.12 provides insights on how the RDM network satisfied its data diffusion requirements. Specifically, this figure plots the mean utility values for Goal (C) that

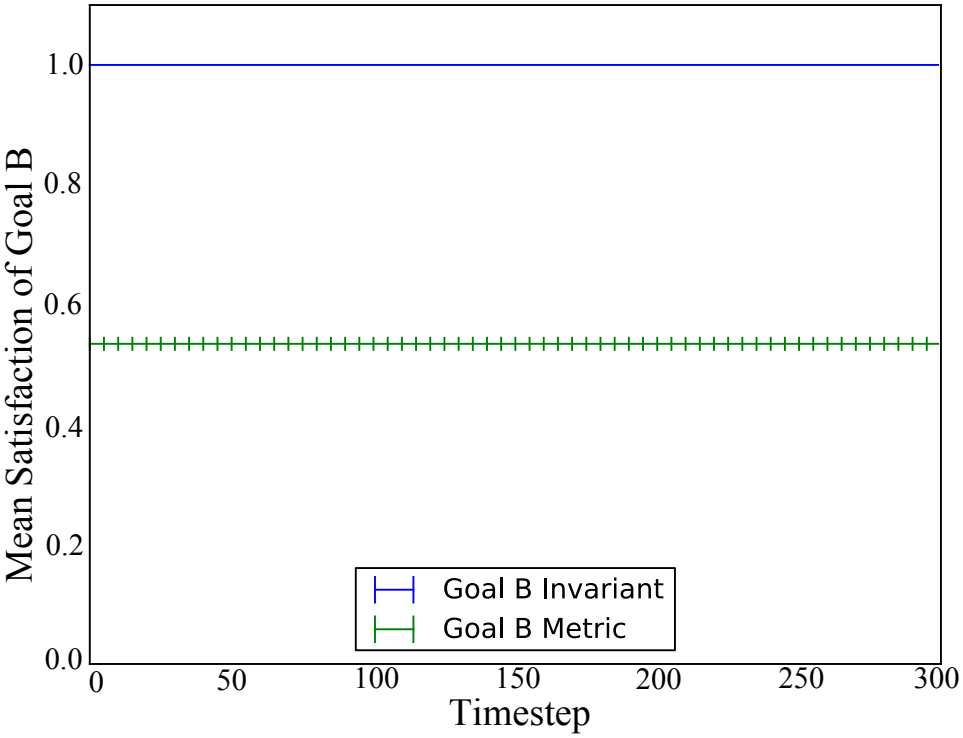


Figure 3.11: Utility values for Invariant Goal (B)

states that each data item should eventually be distributed to every RDM. As this plot shows, the RDM gradually began satisficing this goal at approximately time step 20 until *all* messages were replicated by time step 250.

Figure 3.13 plots the ratio of data messages replicated across the RDM network. As this plot illustrates, the utility values for Goal (C) roughly correlate with the ratio of data messages diffused. Collectively, Figures 3.12 and 3.13 support the utility values depicted in Figure 3.12 since they measure the relative availability and protection of data items in the network during each simulation trial.

As specified by Goal (F), the RDM network must maintain connectivity in order to completely diffuse all data items. Figure 3.14, which plots utility values for Goal (F), shows that the RDM network never became partitioned throughout simulations. This plot concurs with the experimental setup described in Table 3.2 since network link failures are not possible, thereby preventing the RDM network from becoming partitioned once it establishes connectivity.

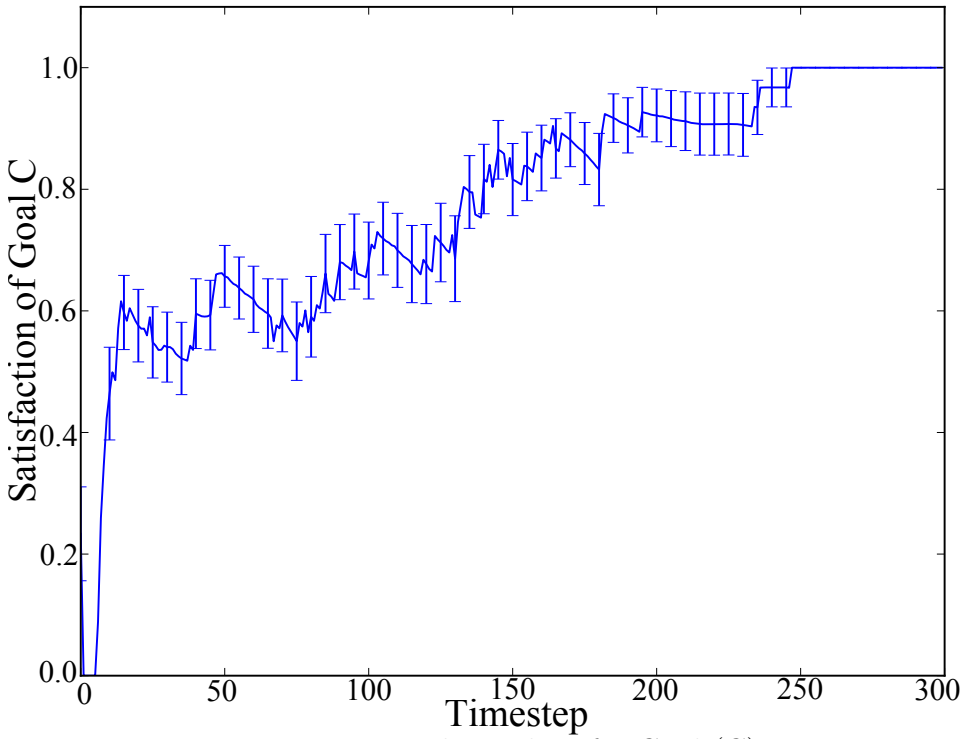


Figure 3.12: Utility values for Goal (C).

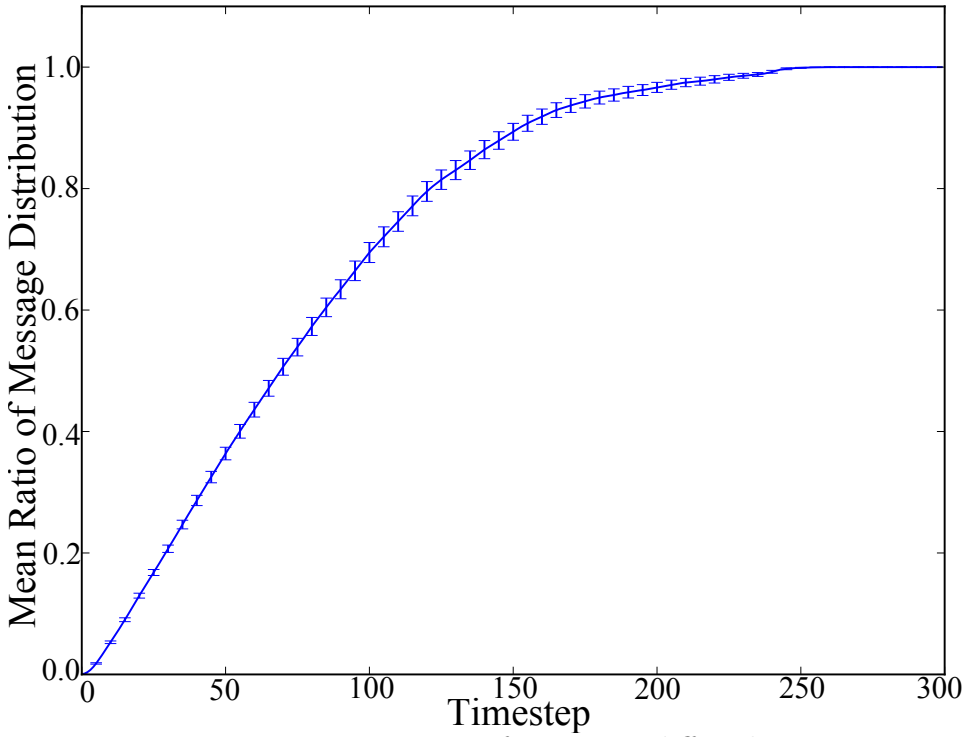


Figure 3.13: Ratio of messages diffused.

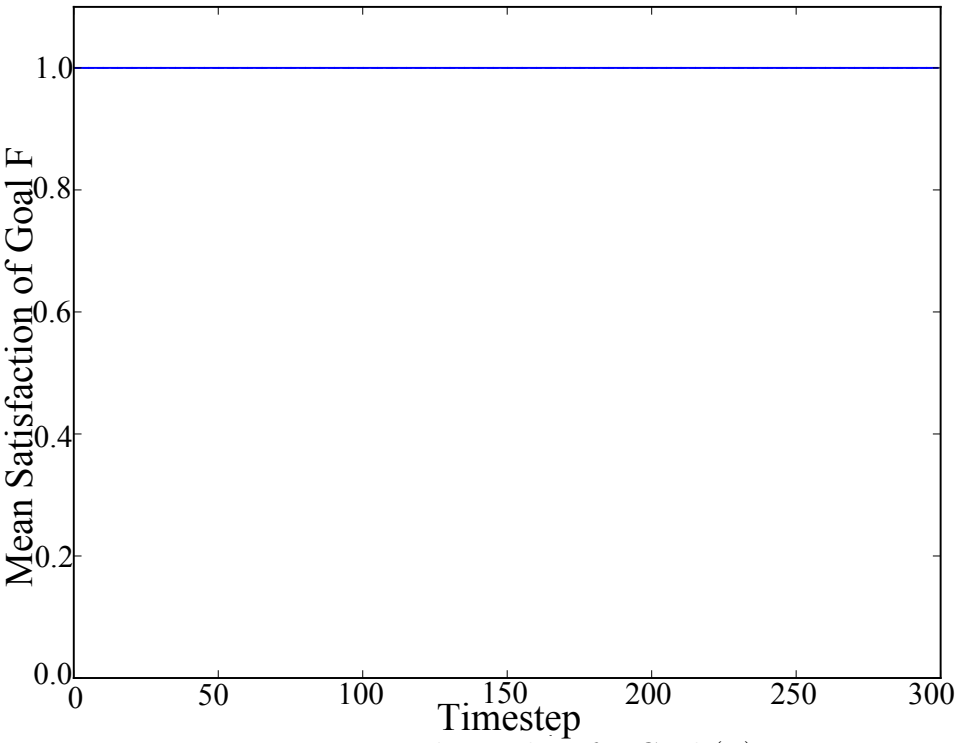


Figure 3.14: Utility Values for Goal (F).

Figure 3.15 provides additional insights regarding the satisfaction of Goals (F) and (B) by plotting the number of active network links throughout each time step. Combined with the utility values in Figure 3.14 that show the RDM network was connected, this additional plot shows how the RDM network minimized the number of active links in order to reduce operational costs. Specifically, with 25 data mirrors, at least 24 active network links are required to establish a spanning tree between all data mirrors. In this particular case, the RDM network activated approximately three to five redundant network links.

As Goal (H) in Figure 3.2 specifies, in addition to replicating each data item at each data mirror while minimizing operational costs, it is important for the RDM network to diffuse data messages *efficiently*. Figure 3.16 plots utility values for Goal (H); it depicts how the satisfaction of this goal gradually decreases from 1.0 to approximately 0.38 as the RDM network becomes congested with large quantities of data messages to replicate and diffuse. In particular, the operational capacity and

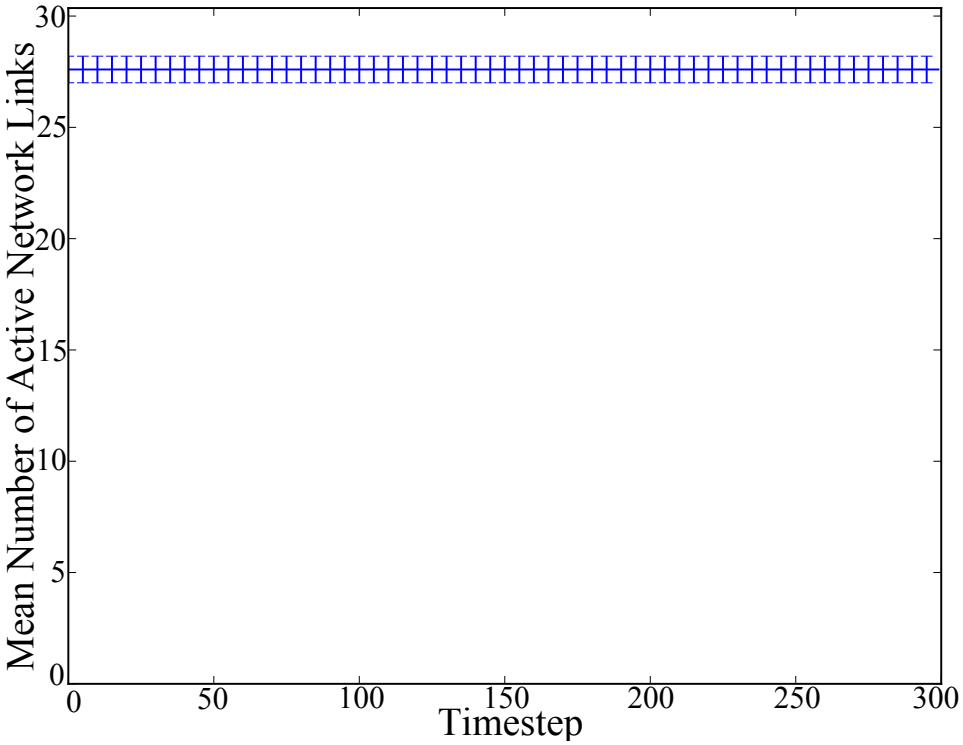


Figure 3.15: Number of active network links.

bandwidth of data mirrors and network links is insufficient to replicate, archive, and distribute all new data messages in the network between time steps 54 and 152.

As can be seen in Figure 3.17, this drop in utility values inversely correlates with an increase in the mean diffusion time across the RDM network. Specifically, the mean distribution time during the first 50 time steps is approximately 12 simulation time steps. However, the mean distribution time gradually increases as the network becomes congested, eventually requiring approximately 54 time steps to diffuse all data by the end of the simulation.

Lastly, the RDM did not execute any adaptations since this experimental setup disallowed network link failures and dropped, corrupted, or delayed data messages at run time. Without triggering self-adaptations, all data mirrors remained active throughout the simulation and none reached passive or quiescent states. As a result, Goals (I), (U), (V), and (W) were maximally satisfied with values of 1.0 at every time step.

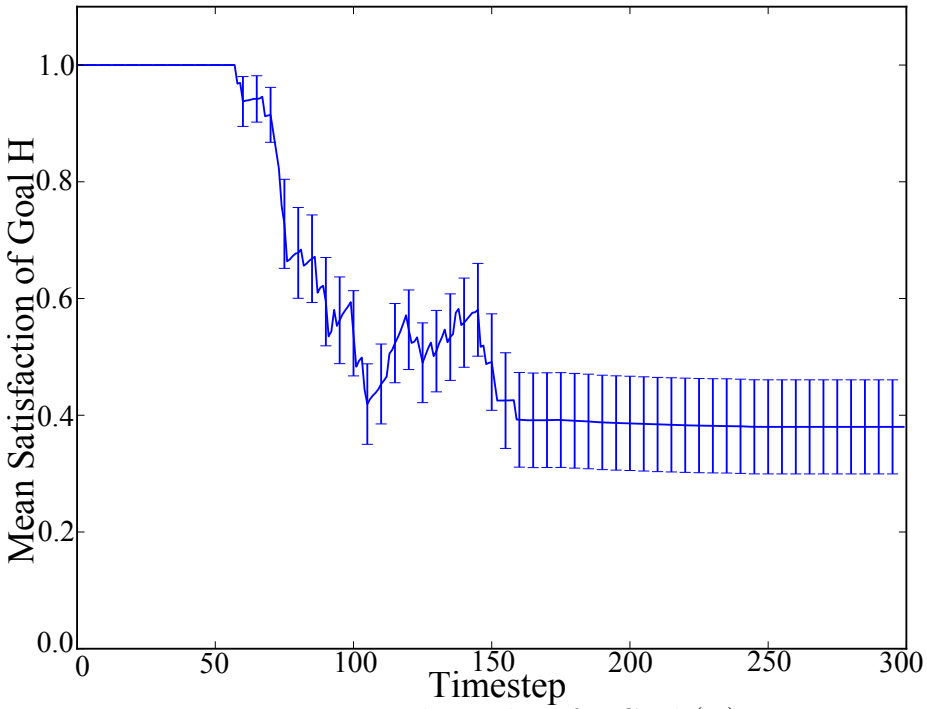


Figure 3.16: Utility values for Goal (H).

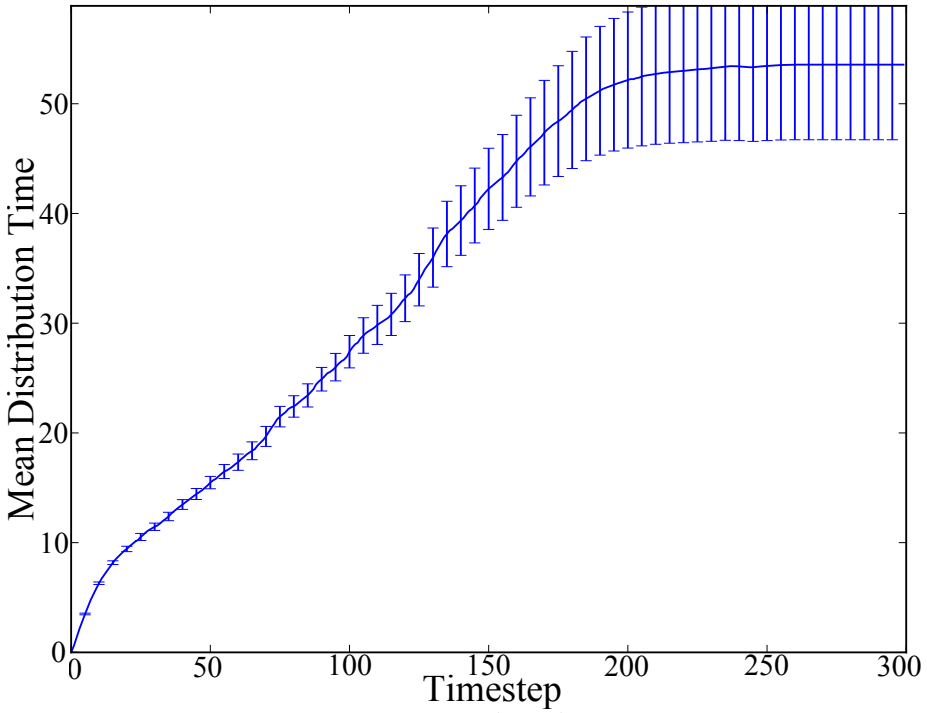


Figure 3.17: Mean distribution time.

As this collection of plots illustrate, the utility functions derived by Athena did not detect any unsatisfied requirements in any of the 30 replicate runs. These plots

also provided insights as to how the RDM network was satisfying its requirements at run time. Specifically, Invariant Goals (A) and (B) always obtained values equal to 1, thereby implying these goals were always *satisfied*. Likewise, all non-invariant goals achieved values greater than 0 by the end of the RDM simulation, thereby implying these goals were *satisficed* to some extent. Combined these results enable us to accept our null hypothesis H_0 ($p < 0.01$, t-test).

3.5.2 Requirements Violation Produced by Environmental Uncertainty

Experimental Objective. This experiment, Experiment 3.2, verifies that Athena derives utility functions that can detect unsatisfied requirements under adverse system and environmental conditions. Specifically, derived utility functions should detect that not all data messages were successfully diffused and provide contextual information about why not all requirements were satisfied.

Hypothesis. As with the previous experiment (Experiment 3.1), we define a null hypothesis, H_0, that states that *utility functions derived by* Athena *will not detect any requirements violations*. Furthermore, for this experiment we also define an alternate hypothesis, H_1, that states that *utility functions derived by* Athena *will detect unsatisfied requirements*. Here, an unsatisfied invariant requirement appears as a utility value that is not equal to one, and an unsatisfied non-invariant requirements violation appears as a utility value equal to zero.

Configuration. This experiment introduces various forms of system and environmental uncertainty. As in the previous experiment (Experiment 3.1), there is uncertainty regarding *when* and *where* data messages are inserted into the RDM network. In addition, this experiment configuration also introduces adverse system and environmental conditions such as noisy monitoring data, data mirror and network link failures, and dropped, delayed, and corrupted data messages. Table 3.3 specifies the

base configuration used for this scenario, where "|" denotes alternate configuration values and "..." specifies an inclusive range of values.

Table 3.3: Configuration for simulation with uncertainty.

Property	Value
Seed	1...25
Distribution	Binomial|ChiSquare|Exponential|Gamma| Normal|Poisson|Uniform
Number Data Messages	100 ... 200
Number Data Mirrors	15...30
Underlying Network Topology	Complete|Grid|Random|Social| Torus|Tree
Budget	$500000.00
Base Data Mirror Capacity	6.0 Gb
Data Mirror Capacity Variance	0.25
Base Network Link Bandwidth	7.0 Gb per time step
Network Link Bandwidth Variance	0.25
Base Data Message Size	2.0 Gb
Data Message Size Variance	0.25
Prob. Data Mirror Failure	0.0 ... 0.05
Prob. Network Link Failure	0.0 ... 0.15
Prob. Data Message Drop	0.0 ... 0.15
Prob. Data Message Delayed	0.0 ... 0.1
Prob. Data Message Corrupted	0.0 ... 0.1
Prob. Data Mirror Sensor Failure	0.0 ... 0.1
Prob. Sensor Fuzz	0.0 ... 0.25

As Table 3.3 shows, the random number generator (RNG) uses different seed values for each simulation trial and produces values according to different possible distributions. In particular, the RNG can use binomial, exponential, gamma, normal, poisson, and uniform distributions when drawing pseudo-random values. These distributions directly affect the likelihood and frequency of different system and environmental events, such as a new data message insertion or a network link failure. For example, while a uniform distribution tends to spread new data message insertions across all valid time steps and data mirrors, other supported distributions will skew new data message insertions toward a narrower range of valid time steps and subset

of data mirrors. In this manner, these additional distributions help evaluate how the RDM network responds to different likelihoods, sources, and impacts of adverse environmental conditions.

This table also shows that the performance characteristics of data mirrors and network links are the same as in the previous experiment. In contrast to the previous experimental configuration (see Experiment 3.1), data mirrors can now fail at run time. When a data mirror fails, its state is lost, including any queued and archived data messages. In addition, network links can also fail at run time. When a network link fails, the RDM network may become partitioned. Besides network link failures, data messages can also be delayed, dropped, or corrupted. Lastly, sensor information can also be inaccurate and imprecise due to various forms of noise.

Results. In contrast to Experiment 3.1, the RDM network was now unable to always satisfy its invariant requirements under adverse combinations of system and environmental conditions. Figures 3.18 and 3.19 show that although the RDM network was almost always able to satisfy Invariant Goal (B), only 19 out of 30 simulation trials satisfied Invariant Goal (A) throughout the entire simulation. Furthermore, the corresponding utility value dips plotted in Figure 3.18 suggest that Goal (A) became unsatisfied at different times in the simulation for various reasons. For instance, in one scenario, Goal (A) became unsatisfied at the beginning of the simulation when a newly inserted data message became corrupted. In another scenario, Goal (A) became unsatisfied near the end of the simulation when the congested RDM network was unable to completely diffuse a data item before the simulation completed.

Figure 3.19 shows that not all simulations satisfied Invariant Goal (B). In the one case where this invariant became unsatisfied, the initial set of active network links failed at run time, thereby causing the RDM network to self-reconfigure by activating less optimal network links that incurred additional costs and ultimately exceeded the allocated budget, thus violating Goal (B). Furthermore, this plot shows that the

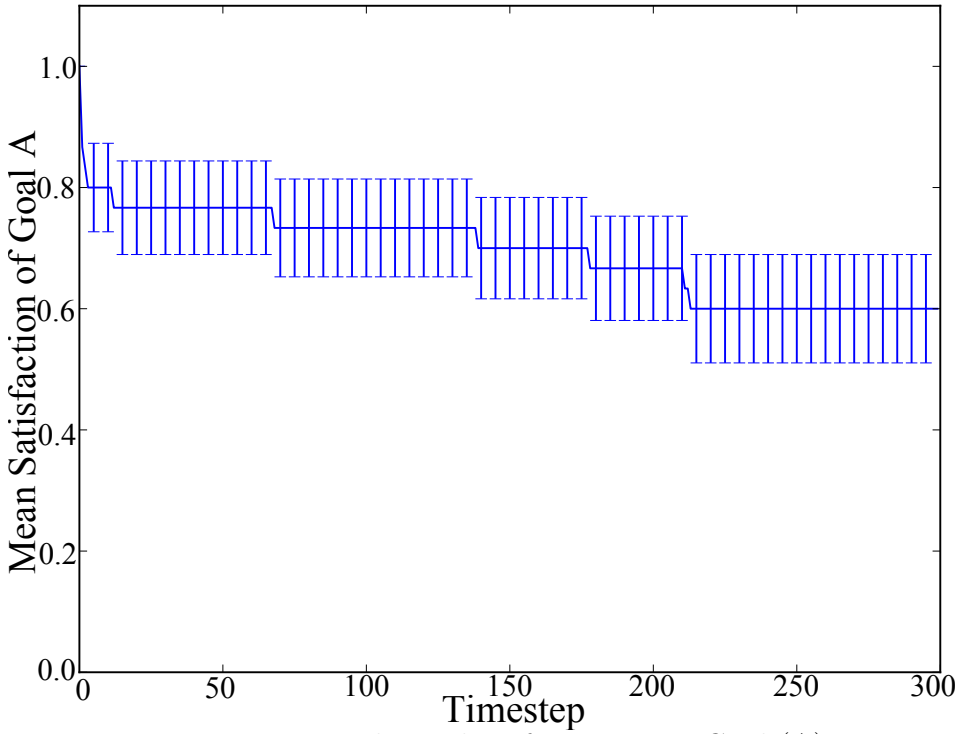


Figure 3.18: Utility values for Invariant Goal (A).

metric-based utility function for Goal (B) was satisficed at approximately 40% of its maximum potential value. In comparison with the plot in Figure 3.11, this lower utility value suggests the RDM network activated a greater number of redundant network links to improve data diffusion performance *and* reliability in response to adverse environmental conditions.

Figure 3.20 plots the utility values for Goal (C) that states that all data messages should be replicated across all RDMs. As this figure shows, the RDM network began to gradually satisfy Goal (C) by replicating and distributing data messages around time step 12. In contrast to the previous experimental scenario, which successfully diffused all data messages by time step 250, in this experiment the RDM network was unable to fully replicate all data messages as captured by the lower than zero utility values by the end of the simulations. Adverse environmental conditions, such as repeatedly failed network links and dropped data messages prevented the eventual satisfaction of Goal (C).

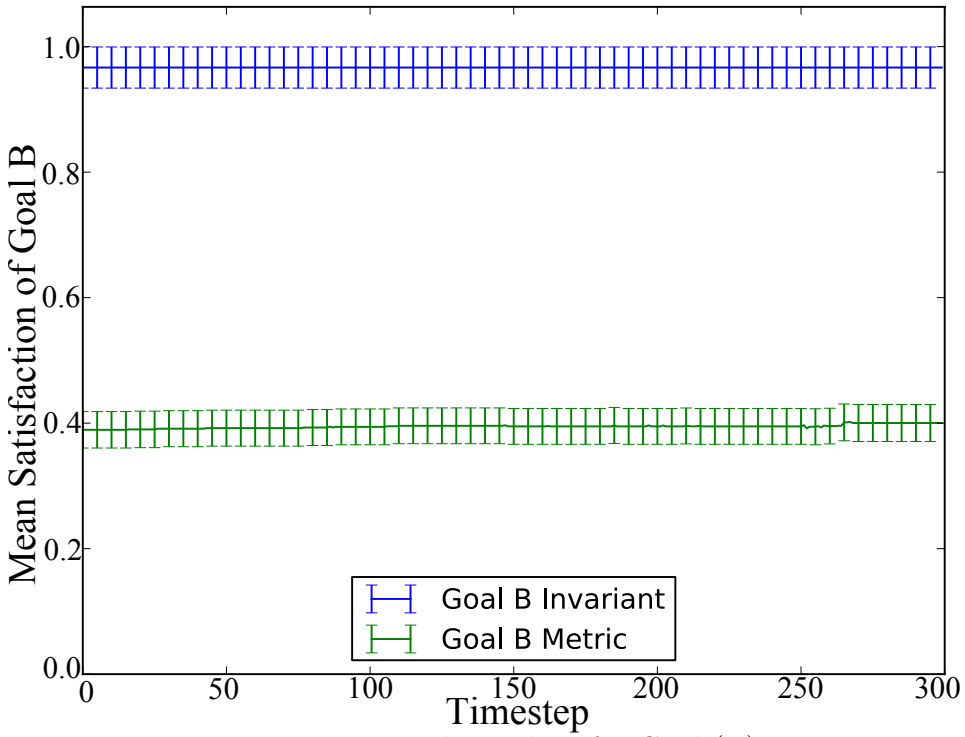


Figure 3.19: Utility values for Goal (B).

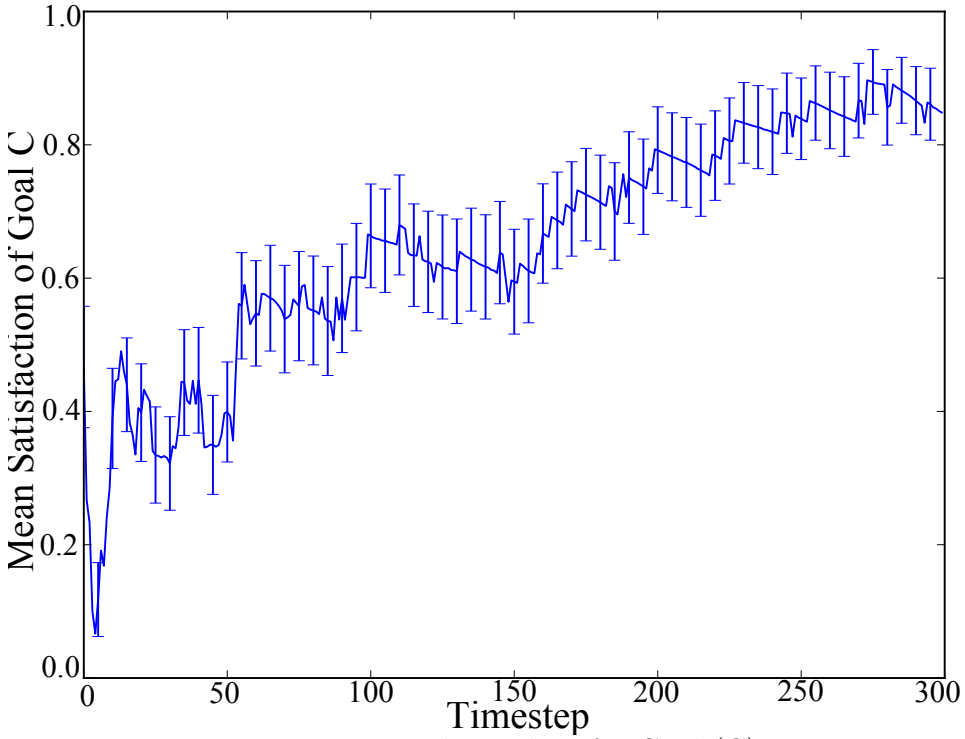


Figure 3.20: Utility values for Goal (C).

Figure 3.21 plots the ratio of data messages replicated and distributed across the RDM network. This diffusion ratio positively correlates with the trend in the utility values for Goal (C) (see Figure 3.20. Specifically, the message diffusion ratio increases as more data messages are replicated across the network yet also depicts how not all messages were diffused by the end of the simulation. As in the previous experiment, these two plots support the utility values captured in Figures 3.18 and 3.19 for Goals (A) and (B), respectively, since they show that not all data messages were protected against data mirror failures.

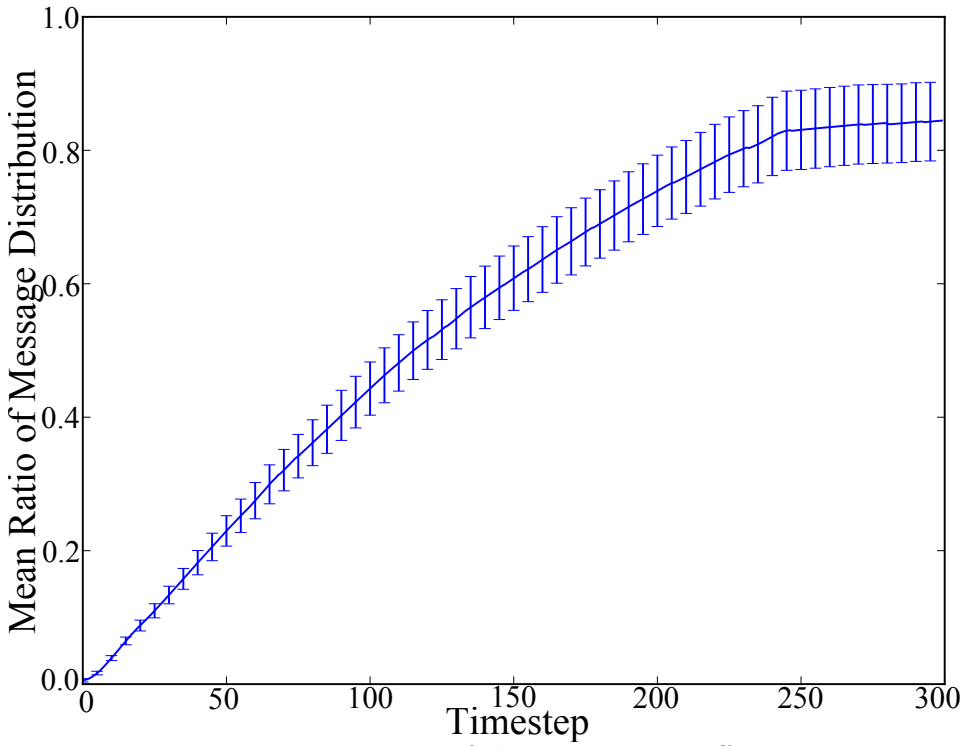


Figure 3.21: Ratio of data messages diffused.

Given that this experiment introduced network link failures, it is quite possible for the RDM network to become partitioned at run time. Figure 3.22 plots the utility values for Goal (F) that states that the RDM network should minimize the number of network partitions in order to distribute data messages to all RDMs. As this plot shows, mean utility values for Goal (F) were within the ranges of 0.84 and 1.0. This drop in utility values suggests that the RDM network became partitioned at run time

due to recurrent network link failures. Note, however, that the RDM network did not suffer from more than one concurrent network partition as the utility values would have dropped below 0.8 in that case.

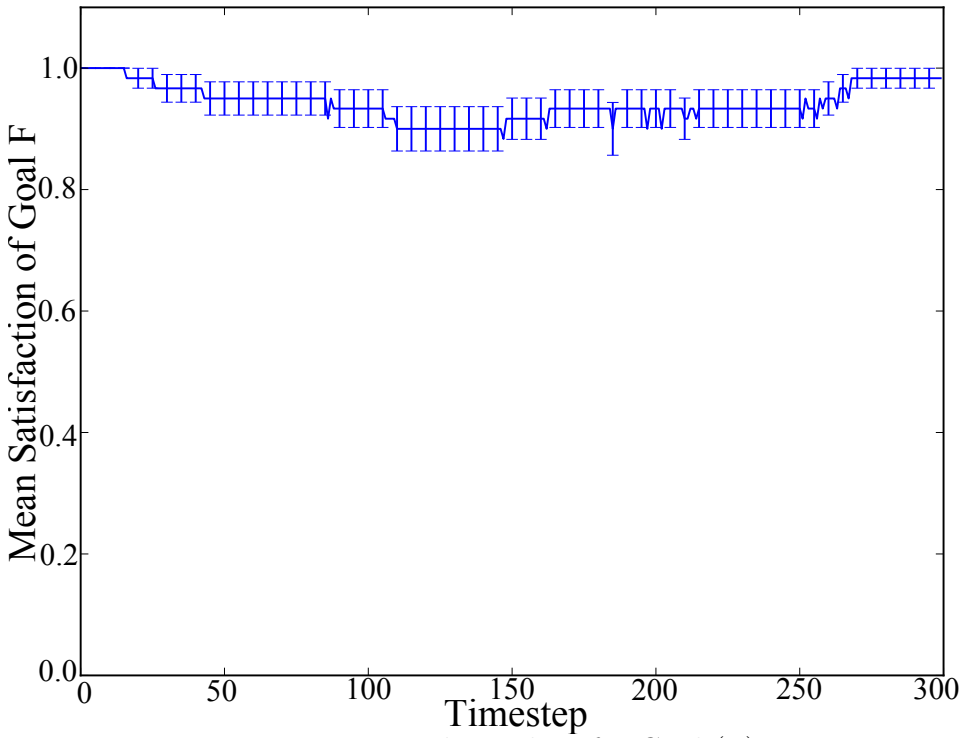


Figure 3.22: Utility values for Goal (F).

Similarly, Figure 3.23 plots the mean number of active network links in the RDM network throughout each time step. As this plot shows, the RDM network activated approximately 28 to 29 network links. As such, four to five of these network links were redundant and improved data diffusion performance and reliability. Specifically, this redundancy protected the RDM network even when multiple concurrent link failures occurred such that the network did not end up with more than one partition.

Figure 3.24 plots utility values for Goal (H) in Figure 3.2, which states that the RDM network must minimize the amount of time taken to diffuse data messages. As with the previous experiment (see Figure 3.16), this figure depicts how the satisfaction of this goal gradually decreases from 1.0 to approximately 0.4 as the simulation progresses.

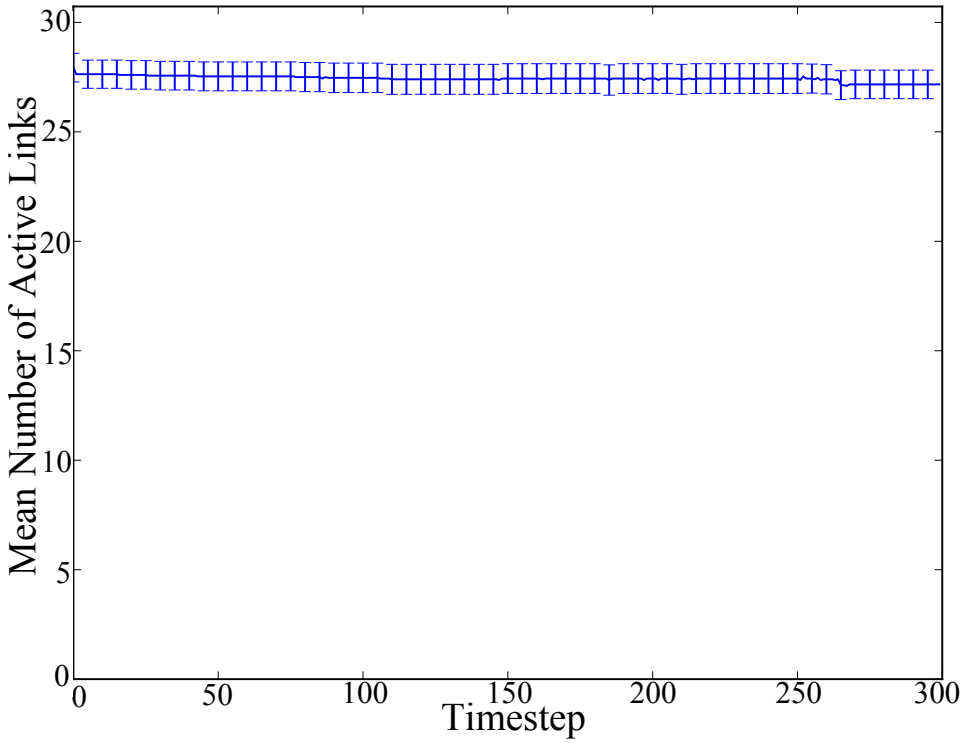


Figure 3.23: Number of network links.

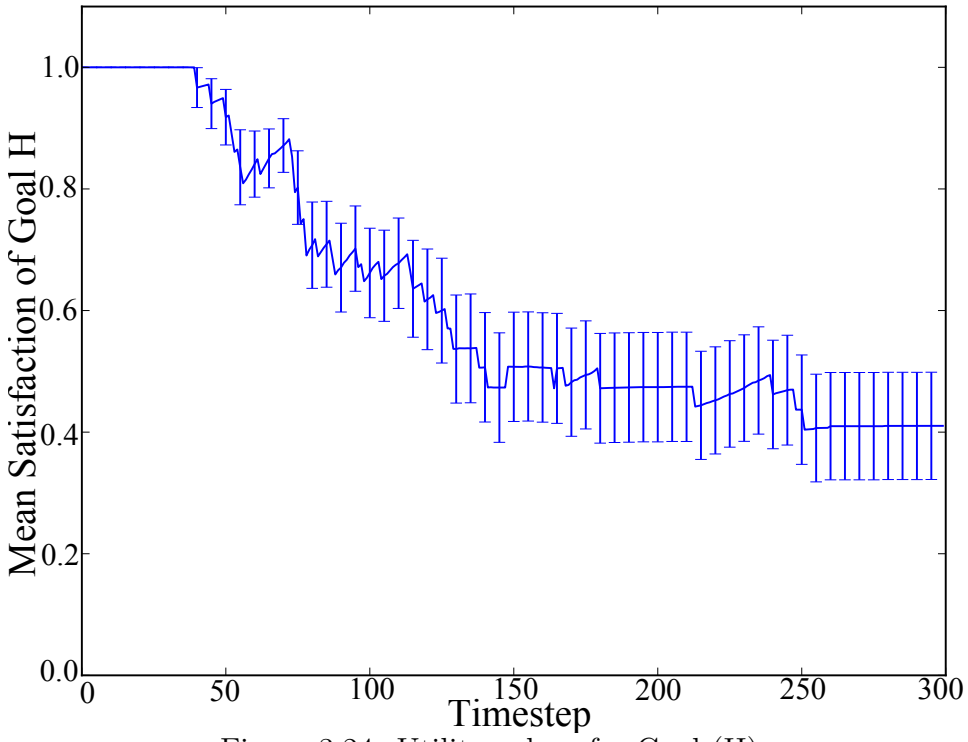


Figure 3.24: Utility values for Goal (H).

Part of the reason for this drop in the utility values for Goal (H) can be seen in Figure 3.25, which plots the mean distribution time throughout each time step in the simulation. Compared with results obtained in Experiment 3.1, the mean diffusion time for this scenario increased from 54 to 83 time steps by the end of the simulation. This increase in diffusion time was caused not only by a congested RDM network, but also by repeatedly dropped, delayed, and corrupted data messages.

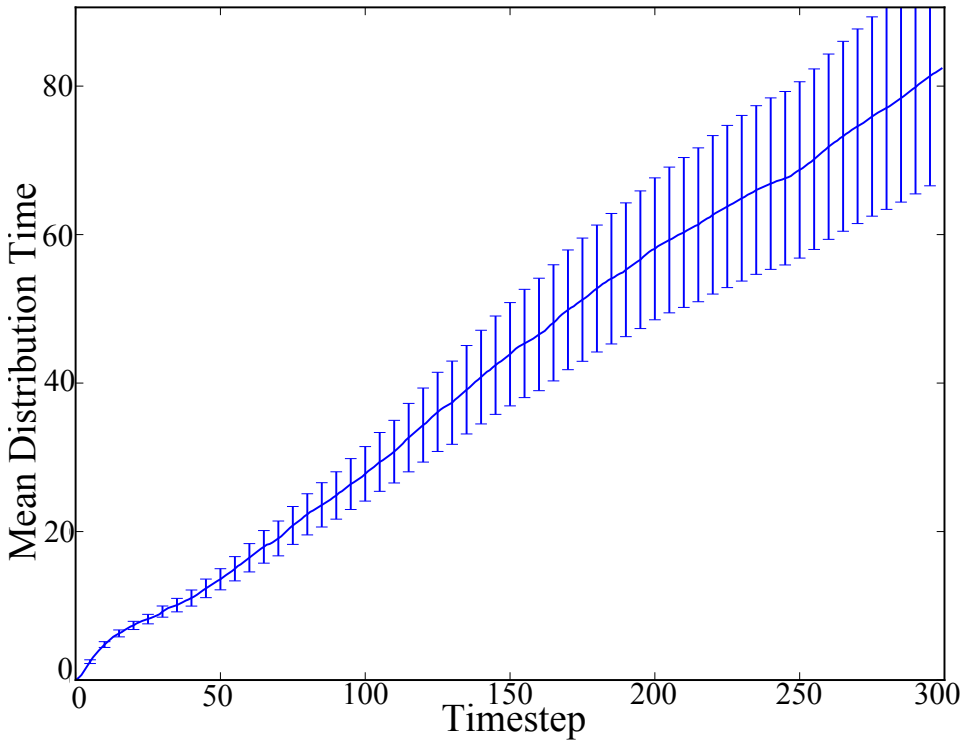


Figure 3.25: Mean distribution time.

Figure 3.26 plots utility values for Goal (I) in Figure 3.2 that specifies that the RDM network should minimize adaptation costs. In contrast to the previous experiment, this scenario introduces system and environmental uncertainty such that the RDM network has to self-reconfigure in order to continue satisfying its requirements. As this figure illustrates, utility values for Goal (I) were within the ranges of 0.86 and 1.0, thus implying that the RDM network was, for the most part, able to continue replicating and distributing data messages even while adaptations were performed.

Lastly, Figure 3.27 plots the mean cumulative number of adaptations triggered

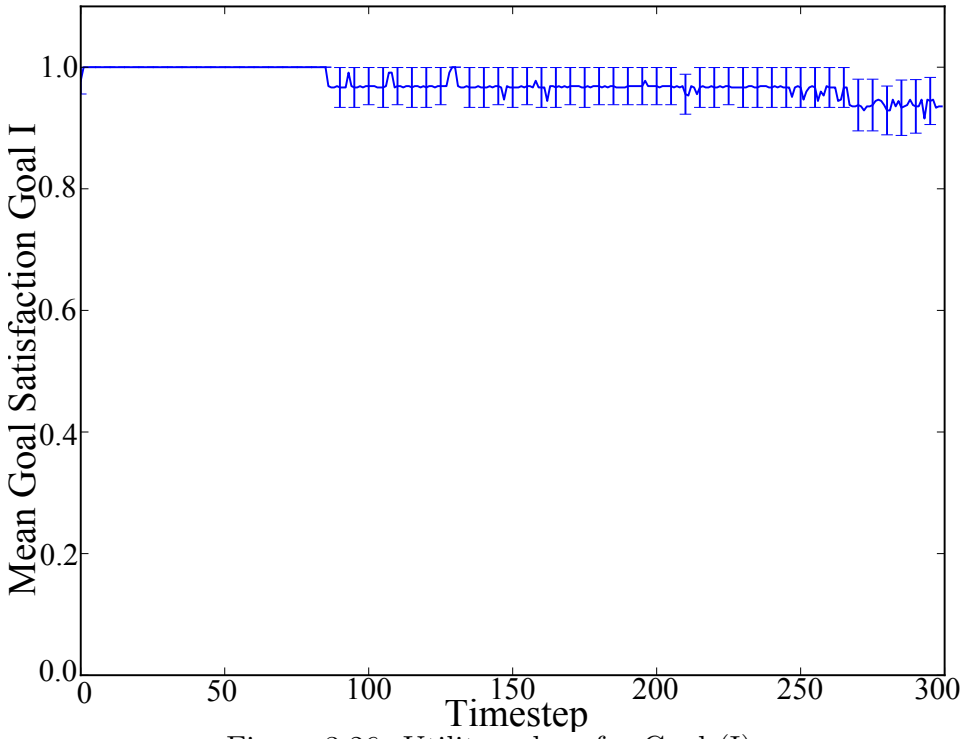


Figure 3.26: Utility values for Goal (I).

throughout each simulation. In comparison with Experiment 3.1, this plot depicts how the RDM network had to self-reconfigure anywhere from 2 to 15 times per simulation in order to re-establish connectivity and continue diffusing data due primarily to network link failures.

As this collection of plots illustrate, the utility functions derived by Athena managed to detect unsatisfied requirements during all 30 replicate runs. In addition, these plots also provided insights as to why requirements became unsatisfied at run time. In particular, Invariant Goals (A) and (B) were not always equal to 1, thus implying these goals became *unsatisfied* under certain combinations of adverse system and environmental conditions. Furthermore, although non-invariant goals were still *satisficed* in this experiment, they were satisficed to a lesser degree when compared with the results obtained in Experiment 3.1. Combined, these results enable us to reject our null hypothesis H_0 ($p < 0.05$, t-test) and accept our alternate hypothesis H_1 ($p < 0.05$, t-test).

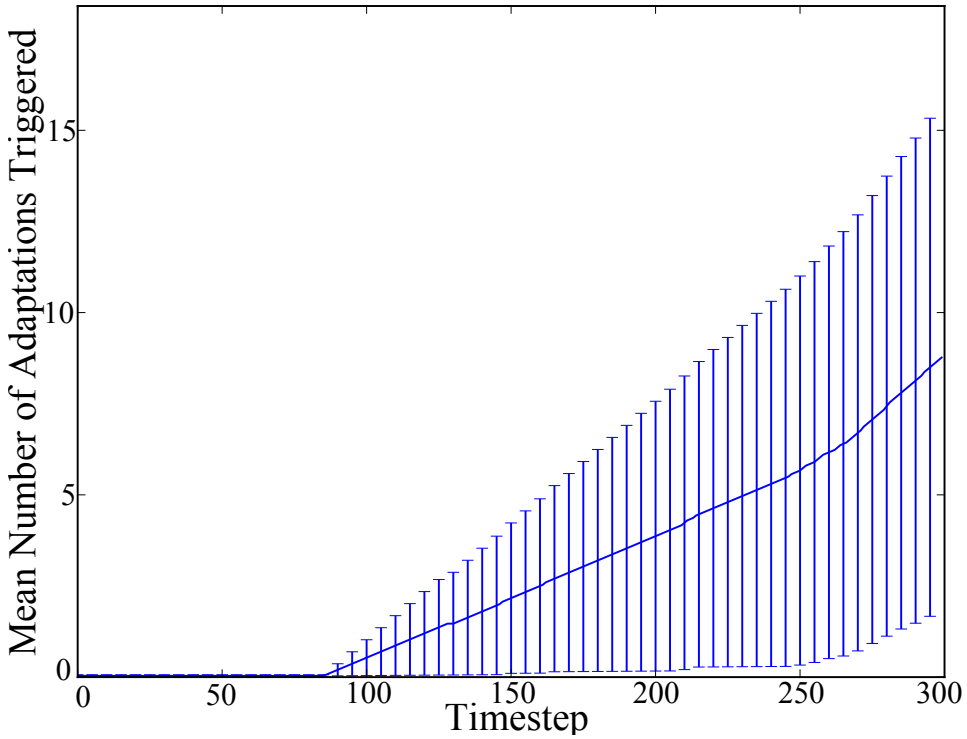


Figure 3.27: Number of active data mirrors.

3.6 Discussion

Compared to traditional requirements monitoring frameworks [28, 29, 97, 98], Athena provides a systematic approach for automatically deriving utility functions that does not rely on ad-hoc or statistics-based approaches. Instead, Athena generates utility functions for requirements monitoring as specified by the objectives and constraints captured in a goal model of the DAS and the set of ENV, MON, and REL elements identified in the RELAX process. By reusing artifacts developed by a requirements engineer as part of the RELAX process, Athena reduces the manual efforts that a requirements engineer must perform in order to monitor requirements satisfaction. Furthermore, by using goal type-specific templates, Athena supports the monitoring of functional, non-functional, and RELAXed goals via state-, metric-, and fuzzy logic-based utility functions.

Valetto *et al.* [20] suggested that utility functions provide a light-weight alter-

native for self-assessment purposes when compared to state-based model checking requirements monitoring approaches [28, 29, 97, 98]. From this perspective, Athena produces utility functions that comprise simple mathematical computations for assessing a goal's satisfaction or satisficement. Requirements monitoring approaches, on the other hand, rely on tracing through state-based model representations of the system in order to detect a requirements violation that, in comparison with utility functions, can be computationally expensive depending on the size of the model and the number of transitions it has. Invoking a utility function either too often or not often enough, however, could potentially cause problems for a DAS as well. If the utility function is invoked too often, then it may interfere with the behavior of the DAS by consuming limited computing resources. Similarly, if the utility function is not invoked frequently enough, then the DAS may perform decision-making tasks based on outdated data, potentially leading to inadequate or unsafe adaptations at run time. Note that Athena does not prescribe how often a utility value should be computed for a given goal as this is an application-specific parameter.

Athena leverages a set of utility function templates in order to generate and instantiate a utility function for each goal type. Once instantiated and implemented within the context of a DAS, these utility function templates are capable of detecting requirements violations and conditions leading to these. Since these utility function templates are generic in nature, they could be augmented or refined by a requirements engineer as necessary in order to better capture the semantics of goal satisfaction in specific application domains. For instance, the current suite of utility function templates used by Athena could be extended to incorporate sigmoidal function shapes that are commonly found in robotic application domains. Essentially, these new utility function templates provide different levels of sensitivity to system and environmental conditions when monitoring the satisfaction or satisficement of a goal.

3.7 Summary

This chapter presented Athena, a technique for automatically deriving a set of utility functions from a KAOS goal model that may include RELAXed goals. The derived set of utility functions can be implemented within the context of a DAS to assess the satisfaction of invariant goals as well as the satisficement of non-invariant and RELAXed goals. State- and fuzzy logic-based utility functions enable a DAS to evaluate whether an invariant or RELAXed goal has been violated, respectively. Metric-based utility functions, on the other hand, enable a DAS to detect conditions conducive to a requirements violation, thereby enabling a DAS to mitigate such conditions before a goal becomes unsatisfied. Experimental results demonstrate that Athena can generate utility functions for quantifying how different system and environmental conditions affect how well a DAS satisfies its requirements at run time.

Chapter 4

Exploring Environmental Uncertainty

This chapter describes how our model-based framework supports the automatic discovery of operational contexts that produce requirements violations and latent behaviors in a DAS. First, we motivate the importance of automatically exploring the space of possible operational contexts a DAS may encounter at run time. We then introduce Loki, [1] the component in our framework that searches for *interesting* combinations of system and environmental conditions. Next, we present the Loki process for exploring the execution environment of a DAS. Subsequently, we apply Loki to the RDM network application and present and discuss experimental results. Lastly, we discuss different ways to apply Loki and summarize main findings.

[1] Loki is the god of trickery and mischief in Norse mythology. In our model-based framework, Loki attempts to trick and deceive a DAS by introducing various sources of uncertainty.

4.1 Motivation

A key objective in requirements engineering is to progressively identify, analyze, and refine system requirements and domain assumptions [105]. To augment goal-oriented models with more comprehensive and realistic requirements, Letier and van Lamsweerde proposed heuristics, refinement patterns, and formal techniques for reasoning about obstacles [105, 106, 107] and partial goal satisfaction [69]. Nevertheless, early system requirements and domain assumptions are often ambiguous and idealized, and can lead to inconsistencies between the specification and run-time behavior of a system [106, 107, 111, 112]. For instance, during the testing phase of an industrial-sized case study, Lutz and Mikulski [71] discovered incomplete requirements specifications and unexpected requirements interactions. As such, tools and techniques are needed to rigorously explore possible inconsistencies between the requirements and run-time behavior of the system-to-be, preferably during the earlier stages of the software development life cycle.

Uncertainty may lead to inconsistencies between the requirements and run-time behavior of a DAS. Within a DAS, uncertainty arises primarily from two sources: inaccurate monitoring data and unanticipated or poorly understood environmental conditions. Specifically, a DAS relies on its monitoring infrastructure to measure properties about its execution environment to identify when it should self-reconfigure. Unfortunately, the monitoring infrastructure of a DAS is potentially unreliable and can produce inaccurate, imprecise, and unanticipated sensor values. Likewise, as DASs increase in complexity and become intertwined with the physical elements, including the environment, it becomes increasingly impractical for a human to exhaustively explore or even understand the space of operational contexts that the DAS will encounter throughout its lifetime [13, 111, 112, 114].

Recently, evolutionary computation techniques have been applied to generate suites of test cases that cause a failure in the system under test [2, 66, 81]. These

approaches test a system by providing different sets of inputs and verifying the correctness of its output. Although these techniques facilitate the identification of obstacles, they tend to converge upon specific types of failures that explicitly satisfy the user-defined fitness function that guides the evolutionary algorithm. Moreover, while these approaches are intended to test systems during the implementation phase, it would be ideal to explore how the environment affects the behavior of a DAS during the requirements engineering phase, where there is greater flexibility for resolving obstacles that prevent the satisfaction of goals [106, 107].

4.2 Introduction to Loki

Loki is an evolutionary computation-based approach that automatically explores and evaluates how the operational context of a DAS affects its ability to satisfy its requirements. Instead of searching for specific operational contexts that cause goals to become unsatisfied, Loki searches for system and environmental conditions that produce diverse and diverse behaviors in a DAS, including requirements violations and latent behaviors. Loki achieves this objective by applying novelty search [68, 96] to generalize, or collapse, vast collections of DAS behaviors into fewer sets of *representative* behaviors in response to adverse system and environmental conditions.

Loki harnesses the concept of evolutionary computation to automatically generate interesting operational conditions that produce undesirable behaviors in a DAS, such as requirements violations and latent behaviors. While a requirements violation clearly obstructs a specific set of system goals, latent behaviors are unexpected and potentially undesirable behaviors that manage to satisfy requirements. Those unwanted behaviors might mean that requirements need to be modified to explicitly disallow the unwanted behavior. A requirements engineer can analyze the set of agents and goals involved in both types of behaviors to refine the goal model.

Furthermore, in addition to identifying new obstacles, a requirements engineer can also augment obstacle mitigations to resolve requirements violations as well as further constrain goal specifications to prevent undesirable latent behaviors. The set of system and environmental conditions that cause such behaviors can also be reused as test cases to guide the testing process of implemented systems.

4.3 Novelty Search

Novelty search is a search heuristic developed by Lehman and Stanley [68, 96] for preventing evolutionary algorithms from becoming "trapped" in deceptive and suboptimal areas of the solution space. To achieve this objective, novelty search *replaces* the fitness function of an evolutionary algorithm with a novelty function or metric. Specifically, the novelty of a newly generated solution is computed by comparing it to other solutions in the population *and* solutions stored in an *archive* of past solutions whose behaviors were once novel when they were discovered. By rewarding solutions for being different, novelty search creates a constant selective pressure for the evolutionary algorithm to discover new behaviors and distinguishable solutions and behaviors.

A good novelty metric should compute the *sparseness* of a point in the solution space. Although there are many different ways to compute the novelty value of a solution, commonly applied distance metrics include Euclidean and Manhattan distances [8]. Novelty metrics like these consider areas with dense clusters of solutions to be less novel, and therefore reward those solutions with lover novelty values. Using these distance metrics, the novelty search algorithm applies a simple k-nearest neighbors measure to compute the sparseness of a solution:

$$\rho(x) = \frac{1}{k} \sum_{i=1}^{k} \text{dist}(x, \mu_i) \tag{4.1}$$

where μ_i is the i^{th} nearest neighbor of solution x with respect to the distance metric, and k is a fixed parameter usually determined empirically. If the novelty value of a solution is above a certain user-defined threshold, then that solution is entered into the permanent archive of previously explored solutions. The current population, and the archive of novel solutions, provide a comprehensive history of the area the algorithm has explored. Note that the novelty search algorithm does not degenerate into a random walk as it cannot backtrack into areas of the solution space that is has already explored and archived.

4.4 Loki Process

This section states the expected inputs and outputs of Loki, as well as describes how Loki discovers combinations of system and environmental uncertainty that produce interesting DAS behaviors.

4.4.1 Expected Inputs and Outputs

Loki requires two input elements to generate operational contexts that produce diverse DAS behaviors: a set of utility functions for monitoring requirements satisfaction in a DAS, and an executable specification of the DAS. Next, we describe how Loki uses each of these input elements:

Utility Functions for Requirements Monitoring. Loki requires a set of utility functions in order to monitor and record how well the DAS satisfied its requirements in response to specific operational contexts, such as those generated by Athena. In particular, these utility functions map utility values to specific goals and requirements in the KAOS goal model of the DAS. Loki then uses this numerical information as part of the novelty search distance metric calculation when clustering groups of DAS behaviors.

Executable DAS Specification. Loki also requires an executable specification of the DAS to simulate and evaluate operational contexts. Note that the choice of the simulation platform may restrict the sources of uncertainty that Loki can control. For this chapter we reuse the RELAXed KAOS goal model previously presented in Figure 3.2, as well as the RDM network simulation previously used in Chapter 3.

As output, Loki produces an archive of operational contexts, each of which specifies sources of system and environmental uncertainty, their likelihood of occurring, and their impact or severity. Each of these contexts produces a specific DAS behavior. In addition, Loki also records a history or trace of how well the DAS satisfied its requirements when subjected to each operational context. This information can be analyzed to identify unsatisfied goals, agents involved in undesirable behaviors, and the specific conditions that lead to these events and behaviors.

4.4.2 Environmental Assessment Process

The data flow diagram in Figure 4.1 overviews how Loki iteratively generates, simulates, and evaluates operational contexts, measures the distances between discovered behaviors, and facilitates the identification of missing and inadequate obstacle mitigations. As this figure illustrates, a requirements engineer (1) identifies possible sources of uncertainty, their possible likelihoods, and severities. Next, (2) Loki generates operational contexts by instantiating different combinations of system and environmental conditions based on the identified sources of uncertainty, their likelihood, and severity. These operational contexts expose the DAS to different scenarios that may cause it to self-adapt in order to continue satisfying its requirements. Loki then (3) executes the simulation with the specified operational context and uses utility functions to monitor and record how the DAS satisfied its requirements. Using this information, Loki (4) computes the novelty between discovered solutions. Lastly, (5) a requirements engineer analyzes discovered behaviors, and the operational contexts

that caused them, to revise the goal model.

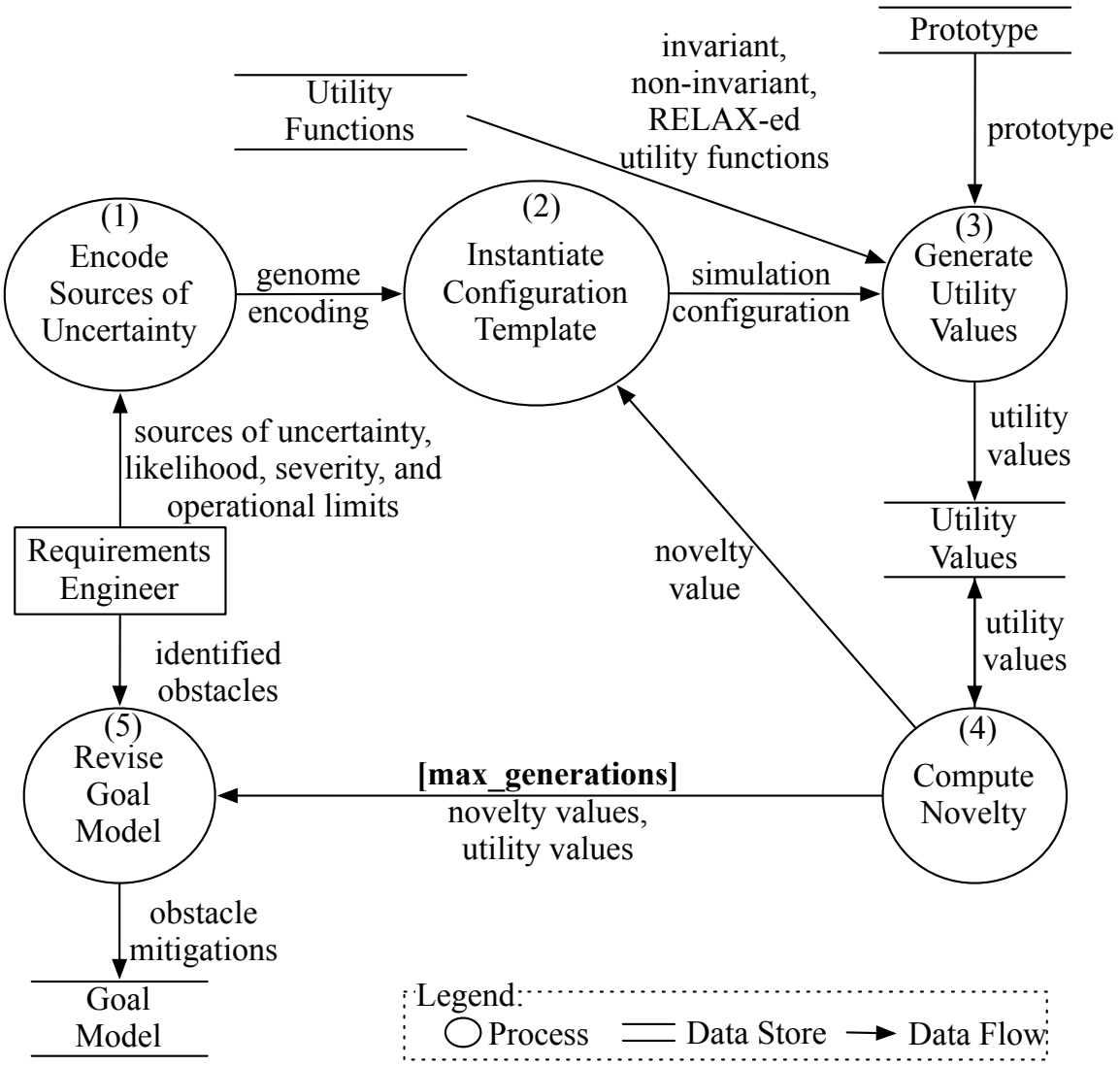


Figure 4.1: Data flow diagram describing how Loki explores effects of system and environmental uncertainty.

Next, we provide details on each step shown in the data flow diagram.

(1) Encode sources of uncertainty. A requirements engineer must define a genome representation to capture possible sources of uncertainty at the shared boundary between the DAS and its execution environment. Each genome comprises a vector of length n, where n is the number of sources of system and environmental uncertainty that can be introduced during the simulation. For each source of uncer-

tainty, the genome must also specify its likelihood of occurring as well as its impact or severity upon the DAS. For example, Figure 4.2 highlights two different sources of uncertainty in a sample genome for the RDM application. Namely, uncertainty in the form of sensor noise will be introduced into the RDM's monitoring infrastructure with a likelihood of 5% and will alter the RDM's operational capacity value by approximately 4%. In addition, uncertainty in the form of adverse environmental conditions will be introduced by injecting network link failures with a likelihood of 8% and a maximum of 3 consecutive failures at any given time. Lastly, the genome must also specify the underlying distribution to use for generating random numbers and events such that Loki can change the frequency and timing at which different events occur throughout the simulation.

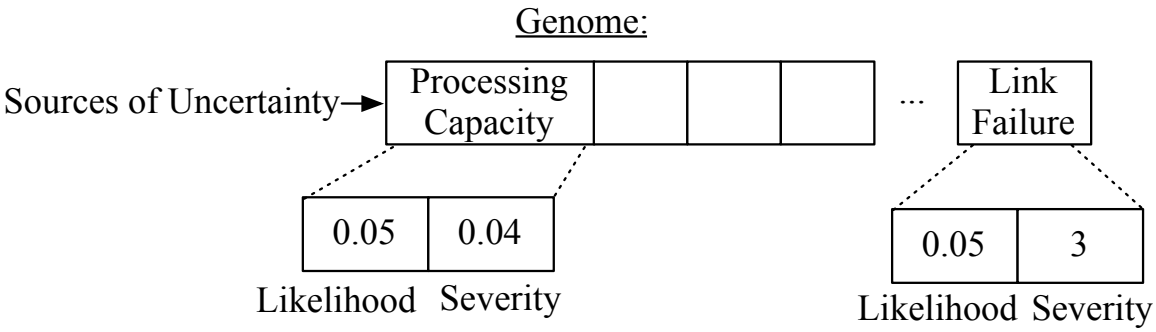
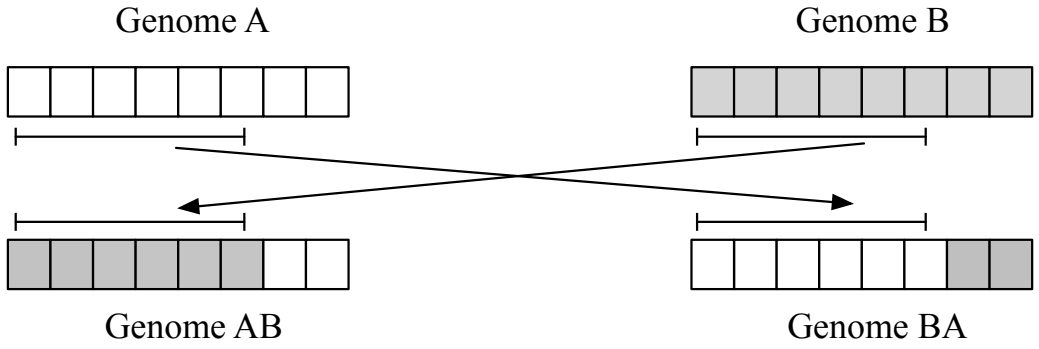


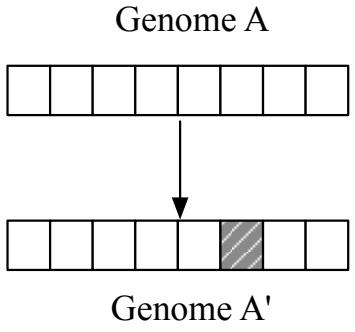
Figure 4.2: Example genome that specifies sensor noise configuration.

(2) Instantiate Configuration Template. The genome structure defined in step (1) serves as a *template* that Loki uses to generate specific operational contexts with system and environmental uncertainty. Using this template, Loki applies crossover and mutation operators to generate new genomes, each of which specifies the likelihood and severity of each uncertainty source. As Figure 4.3(a) illustrates, the crossover operator randomly selects two indices from an existing genome, such as A, and exchanges the uncertainty configurations between those two indices with those from another genome in the population, such as B. As the shading in Figure 4.3(a) shows, the crossover operator creates two new uncertainty configurations, AB and BA,

each comprising uncertainty configurations from both original genomes. In contrast, as Figure 4.3(b) illustrates, the mutation operator selects an existing operational context from the population and randomly modifies its uncertainty specification.



(A) Crossover operator creates new uncertainty configurations by recombining parts of existing uncertainty configurations.



(B) Mutation operator creates new uncertainty configuration by randomly modifying elements of existing uncertainty configurations.

Figure 4.3: Generating new configurations via crossover and mutation operators.

(3) Generate Utility Values. Loki introduces system and environmental uncertainty during each simulation. To this end, Loki first directly injects faults into the system and execution environment of the DAS. For example, environmental uncertainty in the RDM application might include network link failures and dropped data messages. Although such events are usually beyond the scope and control of the DAS, especially when it involves an environmental agent, their occurrence can and often does adversely affect the extent to which a DAS satisfies its requirements at

run time. In the case of the RDM network application, these adverse environmental conditions hinder the process of data diffusion.

In addition, Loki can also introduce system uncertainty into the monitoring infrastructure of the DAS by injecting sensor noise and failures. Specifically, Loki can replace raw sensor values with fuzzed or invalid monitoring values. For example, Loki might alter gathered monitoring data to falsely suggest that an operational and actively used network link has failed. The DAS then computes utility values based on this potentially unreliable monitoring data. Continuing with the RDM example, the incorrect perception of a faulty network link might cause the DAS to unnecessarily self-reconfigure its network topology. Introducing this combination of system and environmental uncertainty enables Loki to evaluate how the DAS reacts and self-reconfigures in response to an altered perception about itself and its execution environment.

(4) Compute novelty. Next, Loki extracts the recorded collection of utility values that measure how well the DAS satisfied its requirements under specific operational contexts. Loki then computes the novelty score for each operational context by using a pair-wise distance metric between each DAS behavior in the population and the novelty archive. The novelty score itself is calculated by applying a Manhattan distance metric [8] to measure differences between utility values that characterize each DAS behavior. These novelty scores or distances are then ranked in increasing order and used to cluster or compute the k-nearest neighboring values as follows:

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

where μ_i is the i^{th} nearest neighbor of solution x with respect to the distance metric, and k is a fixed parameter that limits the cluster size. Operational contexts that produce DAS behaviors whose novelty exceeds the archive threshold or are within

the top 10% of novelty scores in the population are then added to the novelty archive at each iteration.

Repeat steps (2) through (4) until maximum number of generations are executed.

(5) Revise goal model. Loki returns the archive of operational contexts once the search process completes. These operational contexts and their recorded effects can be analyzed to revise the current goal model of the DAS. Possible goal revisions include goal mitigation, prevention, strengthening [107], or RELAXation [13, 114].

4.5 Experimental Results

This section applies Loki to the RDM application to explore how sources of system and environmental uncertainty affect its ability to satisfy requirements. First, we describe the experimental setup and configuration used for these experiments. We then present and analyze experimental results, including a comparison between Loki and a randomized search algorithm used as a baseline.

4.5.1 Simulation and Experimental Setup

Many factors can influence the resulting behavior of a DAS. For example, in the RDM application, the rate at which network links fail concurrently can determine the severity of each triggered self-reconfiguration. Likewise, the amount of flexibility introduced into the satisfaction criteria of a RELAXed goal can also affect the overall number of self-reconfigurations triggered in response to adverse conditions. These experiments leverage the concept of evolutionary computation, in the form of Loki, to address this sheer complexity that arises in terms of possible operational contexts and how they alter the behavior of a DAS at run time.

To control the search process for new DAS behaviors, a requirements engineer

must specify parameters such as population size, crossover and mutation rates, a novelty threshold, and a termination criterium to constrain how the search process generates new solutions, as well as when the search process should terminate. The *population size* limits the number of operational contexts that Loki generates and evaluates each generation. The *crossover* and *mutation* rates define how new solutions are produced by recombining and randomly modifying existing operational contexts in the population, respectively. The *novelty threshold* value imposes a criterion for filtering which behaviors get stored into the archive. Lastly, the *termination* criterion specifies when the search process should terminate, either by executing a specified number of iterations or generations, or discovering a target number of different behaviors.

The specific configuration values are often domain-specific and determined empirically. For example, although a larger population size may increase the number of operational contexts explored in parallel, the overall search process may consume a greater amount of time to complete. The crossover and mutation rates should also balance exploitation versus exploration such that operational contexts neither converge within the population nor the search process degenerates into a randomized search either. The novelty threshold value will also affect the generality of discovered behaviors. In particular, a smaller threshold value will tend to produce a greater number of smaller and denser behavioral clusters and vice-versa. Lastly, the termination criteria must maximize the number of behaviors discovered while simultaneously minimizing the amount of time required for the search process to complete.

Table 4.1 specifies the configuration of the genetic algorithm and novelty search for these experiments. With a population size of 20 genomes and a maximum number of 15 generations, this particular configuration evaluates exactly 300 different operational contexts. A Manhattan distance metric is used to compute the difference between the utility vectors associated with genomes in the population and novelty

archive. After ranking these distances, the novelty of a genome is assigned by computing the mean distance to the ten nearest genomes in the solution space. At the end of each generation, genomes with a novelty value in the top 20% are added to the novelty archive. Lastly, we conducted 25 trials for each subsequent experiment, each with a different seed value that is stored to ensure results are reproducible.

Table 4.1: Genetic algorithm and novelty search configurations.

Parameter Description	Value
Maximum number of generations	15
Population size	20
Mutation rate	0.3
Crossover rate	0.7
Distance metric	Manhattan Distance
k-nearest	10
Archive threshold	Top 20%

Lastly, we reuse the same simulation platform and derived utility functions for requirements monitoring from Chapter 3 to evaluate and record how different operational contexts affect the RDM application. Table 4.2 lists the possible range of values that Loki can specify for each source of uncertainty in the RDM application. In general, none of these sources of uncertainty act as single points of failure in the RDM application unless the likelihood and severity exceed the predetermined upper bound values. In contrast to previous experiments (see Experiment 3.1 and Experiment 3.2), note that Loki is now responsible for specifying the actual combinations, likelihoods, and severities of system and environmental conditions.

As this table shows, Loki can also change the underlying RDM network topology by specifying both the number of data mirrors as well as how they can be interconnected via an underlying topology. Each possible underlying topology results in a different RDM network structure. For example, in a *social* topology, certain data mirrors will be densely connected to other data mirrors. In a *tree* topology, however,

Table 4.2: Possible ranges of uncertainty values.

Property	Value
Seed	1...25
Distribution	Binomial|ChiSquare|Exponential|Gamma| Normal|Poisson|Uniform
Number Data Messages	100 ... 200
Number Data Mirrors	15...30
Underlying Network Topology	Complete|Grid|Random|Social| Torus|Tree
Budget	$500000.00
Base Data Mirror Capacity	6.0 Gb
Data Mirror Capacity Variance	0.25
Base Network Link Bandwidth	7.0 Gb per time step
Network Link Bandwidth Variance	0.25
Base Data Message Size	2.0 Gb
Data Message Size Variance	0.25
Prob. Data Mirror Failure	0.0 ... 0.05
Prob. Network Link Failure	0.0 ... 0.15
Prob. Data Message Drop	0.0 ... 0.15
Prob. Data Message Delayed	0.0 ... 0.1
Prob. Data Message Corrupted	0.0 ... 0.1
Prob. Data Mirror Sensor Failure	0.0 ... 0.1
Prob. Sensor Fuzz	0.0 ... 0.25

exactly $n - 1$ network links can be activated to connect the RDM network. This additional flexibility is intended to explore how the goal model satisfies different RDM network instances.

4.5.2 Discovering Behaviors

Experimental Objective. This experiment, Experiment 4.1, explores the range of RDM behaviors that Loki can discover by harnessing evolutionary computation when generating combinations of system and environmental uncertainty. Specifically, this experiment evaluates whether Loki can discover a range of RDM network behaviors that include desirable behaviors that satisfy requirements, as well as undesirable behaviors such as latent behaviors and requirements violations.

Hypothesis. For this experiment, we defined a null hypothesis, H_0, that states

that *the set of operational contexts generated by* Loki *will not produce significantly different RDM network behaviors.* In addition, we also defined an alternate hypothesis, H_1, that states that *the operational contexts generated by* Loki *will produce different RDM network behaviors, some that satisfy requirements and others that violate requirements.* Here, we use two metrics to evaluate the resulting RDM network behaviors and test these hypotheses. First, we use the novelty value associated with each operational context to estimate the similarity between each pair of RDM network behaviors. Second, we use the utility values recorded during each RDM network simulation to determine the extent to which requirements were satisfied or unsatisfied under each operational context.

Results. Loki consistently discovered different RDM network behaviors. Specifically, out of 300 evaluations performed in each trial, Loki discovered RDM behaviors with a mean novelty value of 98.27 in response to different operational contexts. In addition, for each trial, Loki discovered a minimum and maximum of 32 and 178 novelty distance values, respectively. Furthermore, upon inspection, the archived operational contexts storing operational contexts for the 10% to 15% highest novelty values provided sufficient coverage of *all* explored operational contexts. As Figure 4.4 shows, within this set of operational contexts, approximately 61.54% of them violated Goal (A) at some point during the simulation. The other 38.46 operational contexts enabled the RDM network to satisfy its requirements at run time, though in some cases by exhibiting latent behaviors. Overall, these results confirm the crucial role that system and environmental uncertainty plays in determining whether a DAS is able to satisfy its requirements.

Next, we present some examples of undesirable behaviors discovered by Loki:

Requirements Violations. We analyzed the set of behaviors that involved a requirements violation to determine which operational contexts had the most impact upon goal satisfaction. In general, the most detrimental sources of uncertainty

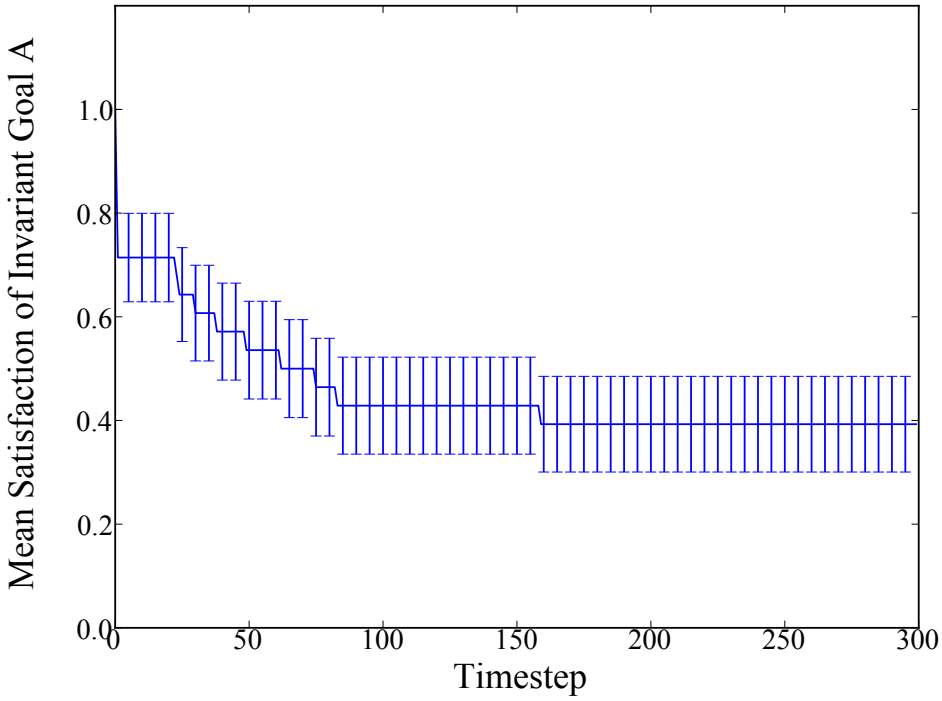


Figure 4.4: Mean satisfaction of Goal (A) under operational contexts in novelty archive.

involved repeated network link, data mirror, and sensor failures. Network link failures usually disconnected data mirrors from the network and thus prevented them from replicating and distributing data messages within the allocated simulation time. Likewise, data mirror failures wiped their state, including queued and archived data messages that might not yet be replicated elsewhere in the network. Lastly, sensor failures frequently triggered network self-reconfigurations such that data mirrors spent a considerable amount of time in passive and quiescent states instead of replicating and distributing data to other data mirrors.

Figure 4.5 presents the topology of an RDM network after it was exposed to an operational context with adverse system and environmental conditions. As this figure illustrates, at one point during the simulation, the RDM network comprises *three* partitions as a result of repeated network link failures. In this case, these partitions

occurred because the network links connecting data mirrors 5-9 and 18-23 failed at run time. These network partitions affected the data diffusion process in two key ways. First, *Partition 1* captures an isolated data mirror that cannot receive, replicate, or distribute data messages with any other data mirror. This isolated data mirror partition is particularly problematic as a new data message was subsequently introduced at data mirror 9 and shortly thereafter that data mirror failed, permanently losing data that was not replicated elsewhere. Second, *Partition 2* and *Partition 3* capture a component of RDMs that can distribute data amongst themselves, but not to data mirrors in other partitions. This network partition prevented the replication and distribution of data messages between the two other RDM components, thus increasing the amount of time required to diffuse data. In both cases, Invariant Goal (A) was unsatisfied either due to data loss or partially diffused messages.

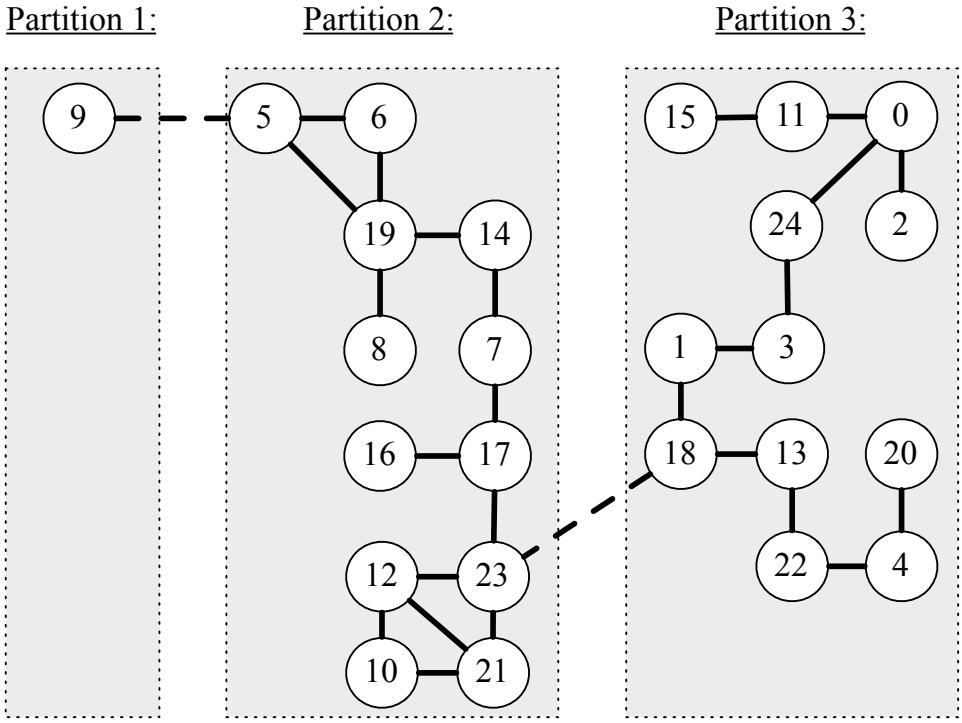


Figure 4.5: Sample partitioned RDM network that leads to a requirements violation.

Figure 4.6 plots the mean satisfaction of Goal (F) under the operational contexts stored in the novelty archive. As this plot shows, Loki introduced adverse environ-

mental conditions such that the RDM network gradually became more partitioned as the simulation progressed. Moreover, at approximately time step 220, the RDM network was operating with two to three partitions, which is likely a root cause for Goal (A) to be unsatisfied in most trials.

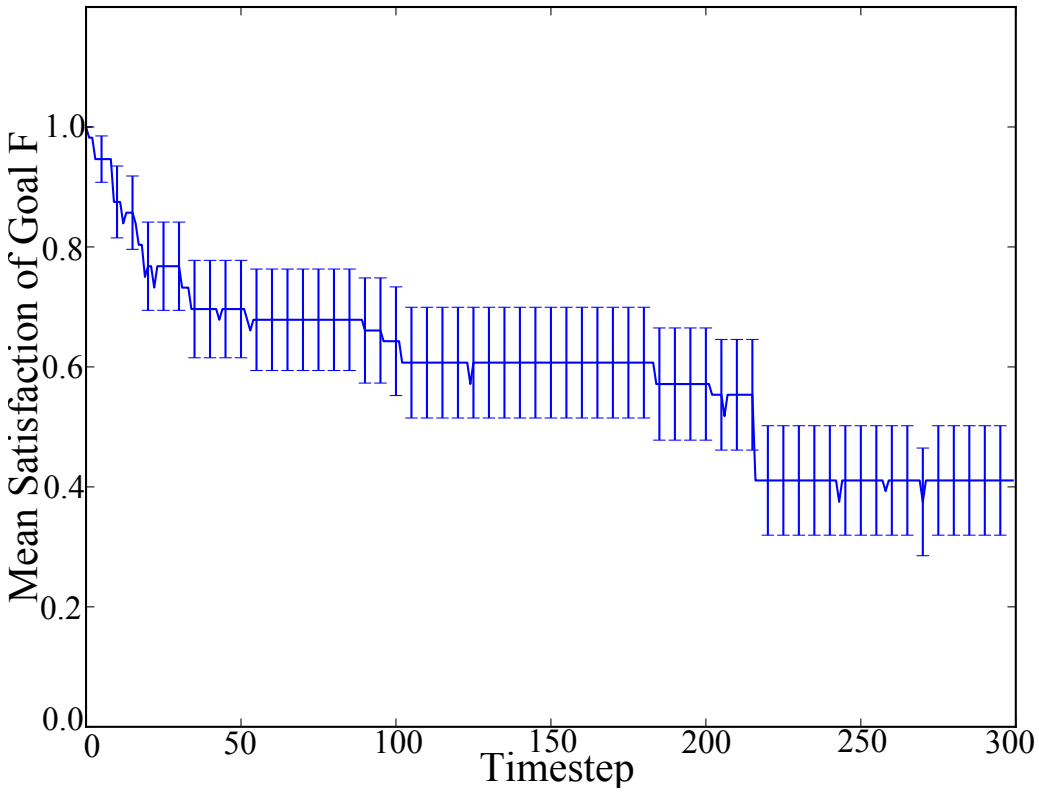


Figure 4.6: Utility values for Goal (F).

Likewise, Figure 4.7 plots the mean satisfaction of Goal (H) under the operational contexts stored in the novelty archive. This plot shows how Goal (H) became gradually unsatisfied as the RDM network became congested with duplicate, dropped, and delayed data messages, thereby increasing the amount of time required to diffuse data.

Other forms of system and environmental uncertainty were not as detrimental to requirements satisfaction in the RDM application. For example, sensor noise caused the RDM network to either select a suboptimal network configuration or to distribute data messages to other data mirrors that already contained such data. While these

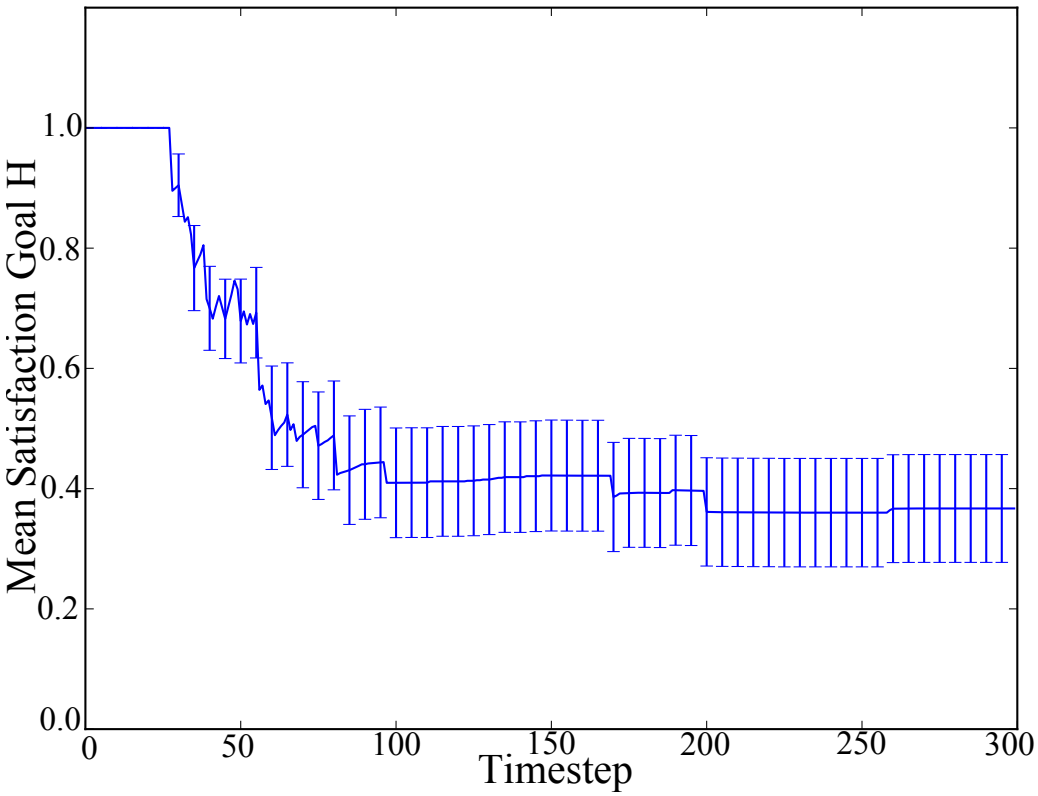


Figure 4.7: Utility values for Goal (H).

conditions did not directly impede the RDM from satisfying its invariant goals, as captured in Figure 4.7, the network became congested with redundant data messages that consumed limited network link bandwidth. In this manner, the effects of system and environmental uncertainty were, to some extent, cumulative.

Latent Behaviors. Several of the 38.46% behaviors discovered by Loki managed to satisfy requirements while also exhibiting latent behaviors. Specifically, most of the latent behaviors exhibited in the RDM network were caused by interactions between Goals (F) and (I) and its Subgoals (U), (V), and (W). Collectively, these goals state that the RDM network should minimize network partitions while also reducing the impact of adaptation upon the data diffusion process. To some extent, Goal (F) competes with Goal (I) and vice-versa since adaptations might be required to minimize the number of network partitions.

This contention between Goals (F) and (I) can be observed in Figures 4.6, 4.8,

and 4.9. In particular, Figure 4.8 shows how the satisfaction of Goal (I) gradually decreased as the simulation progressed, eventually finishing at approximately 80% of its maximum possible satisficement. This trend is also shown in Figure 4.9 as the cumulative number of adaptations increases at an exponential rate during the simulation. Overall, these three plots collectively show how the number of adaptations and network partitions are positively correlated and the number of network partitions and satisfaction of Goal (I) are negatively correlated.

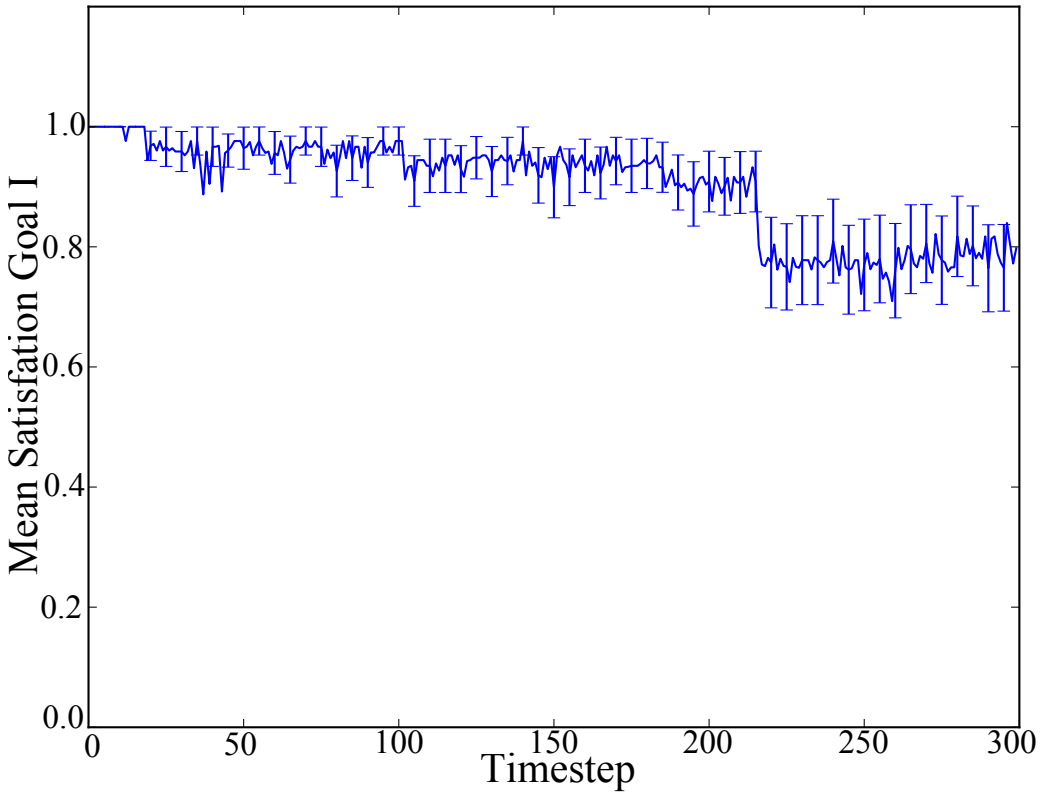


Figure 4.8: Utility values for Goal (I).

These interactions between Goals (F) and (I) basically prevented the RDM network to receive, replicate, and distribute data messages during the data distribution phases. Specifically, to improve the satisfaction of Goal (F), the RDM network triggered self-reconfigurations that placed active data mirrors into passive and quiescent states where data messages could not be received, replicated, or distributed. Under these scenarios, the RDM network triggered too many self-reconfigurations in an

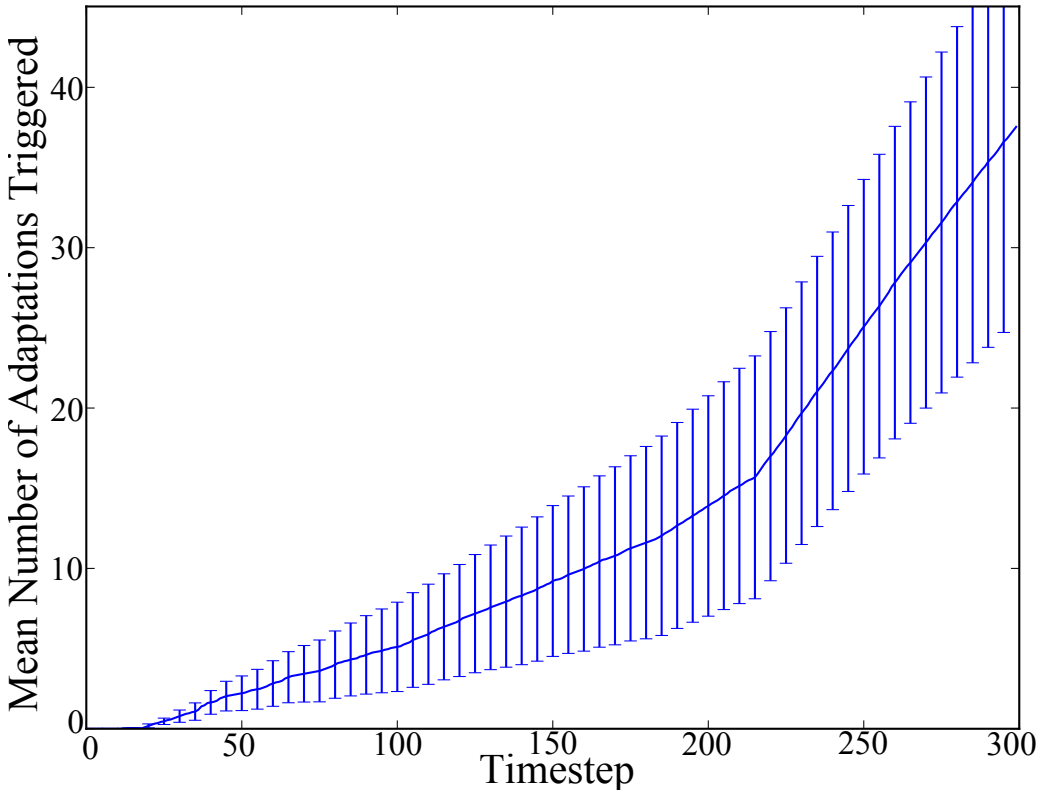


Figure 4.9: Mean cumulative number of adaptations triggered by different operational contexts.

attempt to satisfy Goal (F) at the expense of Goal (I).

While this tradeoff between satisfying Goal (F) and Goal (I) managed to satisfy invariant goals, the RDM network took approximately 23.6 more time steps to accomplish the data diffusion process. For instance, Figure 4.10 shows the topology of an RDM network near the end of a simulation. As this figure illustrates, the RDM network became partitioned after the network link connecting data mirrors 3 and 17 failed. Although this network partition did not isolate any data mirror, new messages inserted at data mirror 4 were not propagated across the rest of the RDM network until connectivity was restored at a subsequent time step. These types of network partitions often increased the time required to diffuse data. Moreover, if the time required to diffuse data continues increasing, as the trend in Figure 4.7 suggests, then it is likely that the allocated simulation time will become insufficient to diffuse all

data at some point, thereby violating Goal (A).

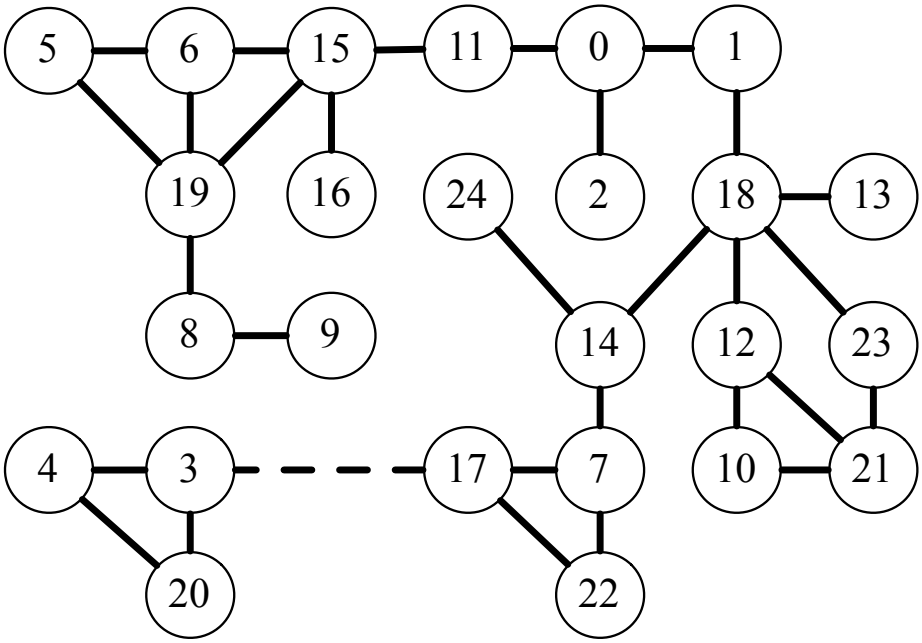


Figure 4.10: Sample RDM network partition that hinders data diffusion.

These experimental results enable us to reject our null hypothesis H_0 ($p < 0.01$ t-test). That is, results demonstrate that Loki was able to discover a wide range of RDM network behaviors as measured by the novelty and requirements satisfaction utility values. Specifically, the RDM network behaviors discovered by Loki included desirable behaviors that satisfied requirements in expected manners, as well as undesirable latent behaviors and requirements violations. Moreover, the resulting novelty archives effectively collapsed each explored operational context into a representative set of operational contexts. Collectively, these results also enable us to accept our alternate hypothesis H_1 ($p < 0.01$ t-test).

4.5.3 Randomized Search Comparison

Experimental Objective. It is often challenging to test software systems early in the development life-cycle as new information about the system-to-be, its requirements, and expected operational contexts is discovered. Randomized search-based

testing is an effective method for testing software systems when no additional information is available to guide the testing process [2]. As such, this experiment, Experiment 4.2, compares Loki with a randomized search algorithm that serves as a comparison baseline.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that *the novelty values achieved by* Loki *will not be different from the novelty values achieved by randomized search.* In addition, we also defined an alternate hypothesis, H_1, that states that *the novelty values achieved by* Loki *will be significantly larger than the novelty values achieved by randomized search.* As in Experiment 4.1, in this experiment we evaluate the range of discovered RDM network behaviors by examining the novelty values associated with each generated operational context. We evaluate these hypothesis under the assumption that whichever algorithm consistently achieves larger novelty values will have explored a greater range of possible DAS behaviors within the same number of operational context evaluations.

Configuration. This experiment *reuses* the novelty archives produced in Experiment 4.1. In particular, each novelty archive stores operational contexts that lead to sufficiently interesting and new RDM network behaviors as discovered by Loki. Moreover, for this experiment we also reuse the same configuration and RDM network executable specification as in Table 4.2 and Experiment 4.1, respectively. In contrast to Experiment 4.1, the novelty search algorithm used by Loki is now replaced with a randomized search algorithm.

We randomly generated 300 different operational contexts for each replicate trial in this experiment. Each operational context specifies different sources of uncertainty, their likelihood, and severity. We then executed one simulation of the RDM network for each operational context in order to produce corresponding utility value vectors that capture how well the RDM network satisfied requirements. Since each simulation executed sequentially and independently, the randomized search algorithm did not

have access to knowledge about which areas of the solution space had been explored already. Next, we computed novelty values for each RDM network behavior produced by using the same k-nearest neighbors novelty metric used by Loki. Lastly, we compared novelty values between the two approaches. Intuitively, whichever approach produced larger novelty values managed to cover a larger portion of the solution space.

Results. In general, the randomized search algorithm also discovered adverse environmental conditions that produced requirements violations. Specifically, out of 300 evaluations per trial, randomized search discovered a mean of 143.07 behaviors that involved requirements violations. These results confirm that randomized search algorithms are valuable tools for discovering test cases that trigger failures in the system-to-be.

Although it seemed that randomized search discovered a greater number of requirements violations than Loki, upon closer inspection most of the RDM network behaviors discovered by the randomized search algorithm were similar to each other. As the box plot in Figure 4.11 shows, the novelty values between Loki and randomized search were significantly different (Wilcoxon ranksum test, $p < 0.001$). Specifically, in this experiment, Loki achieved a median value of 96.4 while the randomized search algorithm achieved a median value of 28.1. As this box plot also captures, Loki also consistently found behaviors with larger novelty values than the randomized search algorithm. That is, in every trial, Loki discovered several behaviors with novelty values greater than 120.0, which is considerably larger than the mean and median values obtained by both approaches. Often, the magnitude of the novelty value correlated with the severity of the requirements violation or latent behavior. As such, behaviors with larger novelty values tended to be more interesting to requirements engineers.

This difference in novelty values can be attributed to how Loki and the randomized search algorithm conducted their search process. In particular, the randomized

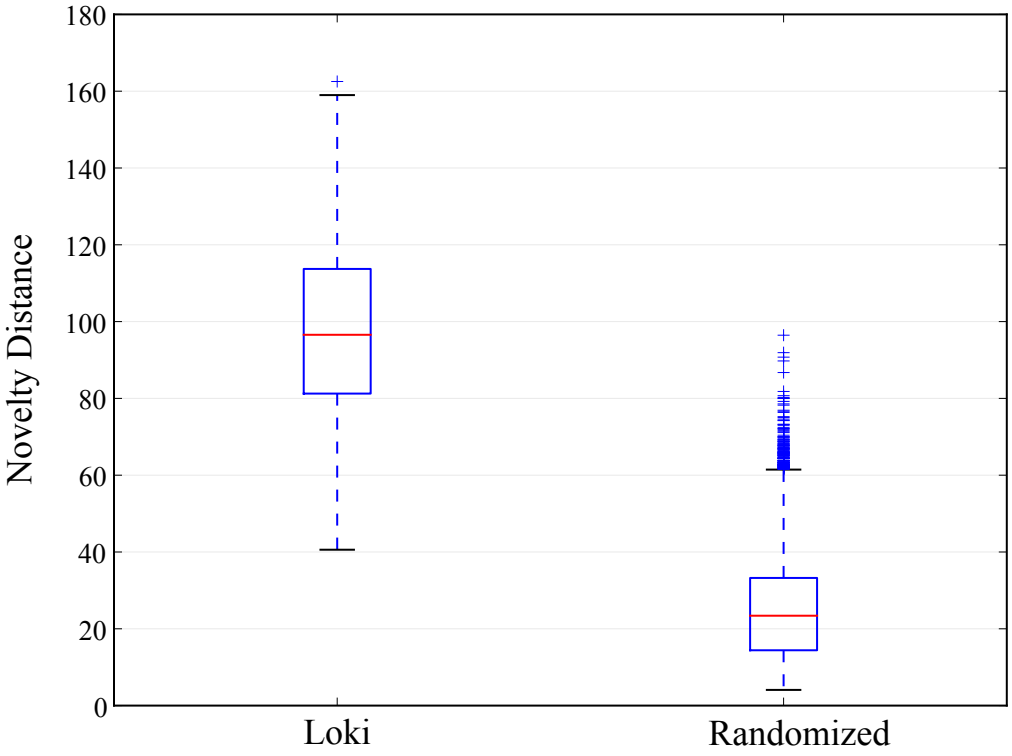


Figure 4.11: Box plot of discovered behaviors by novelty search and randomized search.

search algorithm was unable to guide its search process by exploiting knowledge about how a given operational context affects the behavior of a DAS. WIthout reusing this information about how operational contexts affected the RDM network's behavior, the randomized search algorithm effectively explored many operational contexts that produced relatively similar behaviors. In contrast, Loki gradually leveraged this behavioral information to gradually refine the combination of system and environmental conditions it exposed the RDM network too. Moreover, these results also confirm how the novelty archive of operational contexts enables Loki to avoid backtracking into areas of the solution space that have already been explored.

These experimental results enable us to reject our null hypothesis H_0 ($p < 0.001$ Wilcoxon ranksum test). That is, results demonstrate that Loki was able to consis-

tently discover RDM network behaviors with larger novelty values than randomized search. The obtained novelty values and utility values that captured how well the RDM network satisfied its requirements also demonstrate that Loki was able to explore a larger portion of the solution space within the same number of evaluated operational contexts. Collectively, these results also enable us to accept our alternate hypothesis H_1 ($p < 0.001$ Wilcoxon ranksum test).

4.6 Discussion

A requirements engineer may apply Loki in several ways depending on the sorts of behaviors being sought. Specifically, the k-nearest parameter in the novelty search algorithm directly affects the fitness value computation by controlling how many other behaviors are considered when establishing clusters of representative behaviors. Setting low values for the k-nearest parameter in the novelty search algorithm causes Loki to discover a larger number of representative behaviors and vice-versa. As such, a requirements engineer who wishes to explore an initially small set of representative behaviors might prefer a larger k-nearest parameter value in order to generalize solutions. Once a few interesting behaviors are identified, a requirements engineer might then wish to explore a broader set of representative behaviors by reducing the k-nearest parameter value.

Several mitigation strategies can be applied to disallow the undesirable behaviors produced by these operational contexts, such as KAOS obstacle mitigations [107]. In addition to specifying obstacle mitigations, engineers can also apply the RELAX requirements specification language [13, 114] to specify the extent to which requirements may be temporarily unsatisfied without violating system invariants. To this end, Loki supports iterations where the DAS goal model can be RELAXed to explicitly address discovered adverse operational contexts.

4.7 Summary

This chapter described Loki, an evolutionary computation-based approach for automatically exploring the space and impact of system and environmental conditions upon the behavior of a DAS. Loki uses a genetic algorithm to generate configurations that specify sources of uncertainty, their likelihood, and their impact. These sources of uncertainty manifest themselves at the shared boundary between the DAS and its execution environment in two ways. First, the environment itself can introduce adverse conditions that directly hinder the DAS's ability to satisfy its requirements. Second, potentially noisy and unreliable sensors can alter how a DAS perceives, and therefore reacts, to events in its execution environment. By harnessing the concept of novelty search, Loki can generate and evaluate how operational contexts lead to different DAS behaviors, including desirable behaviors, latent behaviors, and requirements violations.

We demonstrated the use of Loki by applying it to an RDM application to generate different combinations of link failures and unreliable messaging conditions, such as dropped, delayed, or corrupted data messages. Throughout this study, we presented several interesting examples of requirements violations and latent behaviors discovered by Loki. Moreover, experimental results show that Loki is capable of finding a significantly greater number of different behaviors in response to different operational contexts when compared with randomized and traditional genetic algorithm-based testing approaches. Analyzing these operational contexts, and their corresponding impact upon the DAS's requirements, enabled us to identify both missing and inadequately mitigated obstacles.

Chapter 5

Automatically RELAXing Goal Models to Cope with Uncertainty

This chapter presents how our model-based framework supports the automatic RELAXation of goal models to mitigate identified sources of system and environmental uncertainty. First, we motivate the need to automatically explore possible goal RELAXations and how these affect the ability of a DAS to satisfy its requirements in unpredictable and uncertain environments. We then introduce AutoRELAX, the component in our model-based framework responsible for automatically generating RELAXed goal models. Next, we describe the process AutoRELAX uses to generate and evaluate candidate RELAXed goal models, including its expected inputs and outputs. We then apply AutoRELAX to the RDM application and present and discuss experimental results. Lastly, we summarize main findings.

5.1 Motivation

As presented in Chapter 4, unreliable monitoring information and unpredictable or unanticipated environmental conditions can introduce contextual uncertainty into a DAS and thereby limit its adaptation capabilities. Unfortunately, it is unlikely for

a human to manually identify and evaluate all possible combinations of system and environmental conditions that a DAS may encounter throughout its lifetime. In light of this implication, the RELAX specification language can be used in goal-oriented modeling approaches [13] to specify and mitigate sources of uncertainty in a DAS (see Table 2.2 for a description of RELAX operators). In particular, RELAX can be used to extend goal modeling languages, such as KAOS, with fuzzy logic-based operators that specify the extent to which a goal can become *temporarily* unsatisfied and yet deliver acceptable behavior. This increased flexibility can be a valuable mitigation strategy for unanticipated obstacles [13, 105, 107, 114].

While RELAX explicitly models sources of environmental uncertainty that can affect a DAS and provides greater flexibility in how and when a DAS achieves its objectives, it is not necessarily a straightforward approach to apply. Specifically, it can be a challenging task for a requirements engineer to determine at design time which goals to RELAX, what RELAX operators to apply, and how a goal's RELAXation will affect the overall behavior of the DAS at run time. Furthermore, there may be many possible ways to RELAX a goal model depending on whether a requirements engineer wants to maximize goal satisfaction flexibility or minimize the number of RELAXations. As such, having automated support to generate and evaluate RELAXed goal models could help manage these different tradeoffs and facilitate more effective use of RELAX.

5.2 Introduction to AutoRELAX

AutoRELAX extends and automates the approach previously introduced by Cheng *et al.* [13] for addressing identified sources of environmental uncertainty in a DAS with the RELAX specification language. Specifically, AutoRELAX introduces RELAX operators into a KAOS goal model to *temporarily* tolerate unsatisfied non-

invariant goals in a DAS. To this end, AutoRELAX specifies whether a given non-invariant goal should be RELAXed, and if so, which RELAX operator to apply, and to what extent to lessen the constraints or bounds that define a goal's satisfaction criteria.

A requirements engineer can apply AutoRELAX to automatically generate one or more RELAXed models. Each of these RELAXed models enables a DAS to satisfy invariant requirements, cope with specific manifestations of system and environmental conditions, and reduce the number of adaptations performed. AutoRELAX leverages a genetic algorithm as a search heuristic to efficiently search through candidate RELAXed goal models. Throughout this search process, AutoRELAX evaluates the effects of a RELAXed goal model upon the behavior of a DAS in different environments. Ultimately, a DAS uses the resulting set of RELAXed goal models at run time to monitor the satisfaction of requirements even in operational contexts that may expose the adaptive system to sources of system and environmental uncertainty.

5.3 AutoRELAX Process

This section overviews the expected inputs and outputs of AutoRELAX, as well as describes each step that AutoRELAX applies to generate and evaluate RELAXed goal models.

5.3.1 Expected Inputs and Outputs

AutoRELAX requires four input elements to generate RELAXed goal models: a goal model of the DAS, a set of utility functions for requirements monitoring, an executable specification of the DAS, and operational contexts that subject the DAS to adverse combinations of system and environmental uncertainty. Next, we describe how AutoRELAX uses each of these inputs:

Goal Model. AutoRELAX requires a goal model that captures the hierarchy of requirements and constraints that the DAS must satisfy. Currently, AutoRELAX targets the KAOS goal modeling language [18, 105] given its emphasis on capturing functional requirements and its support for identifying and resolving obstacles. Note that each goal must be classified as either an invariant or non-invariant goal since RELAX operators are only applicable to non-invariant goals.

Utility Functions. AutoRELAX also requires a set of utility functions for requirements monitoring in a DAS at run time. As described in Chapter 3, each utility function comprises a mathematical relationship to map gathered monitoring data to a scalar between the inclusive ranges of zero and one. AutoRELAX uses these utility functions, in conjunction with the executable prototype, to evaluate how goal RELAXations affect the behaviors of a DAS.

Executable Specification. AutoRELAX uses an executable specification of the DAS, such as a simulation or prototype, to evaluate the effectiveness of goal RELAXations at addressing system and environmental uncertainty. This executable specification must support the specification of sources of uncertainty that can affect the DAS at run time. Furthermore, this executable specification must incorporate the utility functions for requirements monitoring such that AutoRELAX may trace how well the DAS satisfied each goal at run time in response to different conditions.

Uncertain Operational Contexts. Lastly, AutoRELAX requires a set of operational contexts that specify different combinations of system and environmental uncertainty. Ideally, this set of operational contexts should exercise different parts of the DAS' adaptive logic and produce different types of behaviors ranging from desirable behaviors to requirements violations. For this chapter, AutoRELAX reuses Loki's archive of operational contexts that produce a wide range of representative DAS behaviors.

As output, AutoRELAX generates a suite of RELAXed goal models. Each RELAXed

goal model addresses different tradeoffs between reducing the number of RELAXed goals and minimizing the number of adaptations triggered.

5.3.2 AutoRELAX Process Description

The data flow diagram in Figure 5.1 overviews the AutoRELAX process. Next, we present each step in detail:

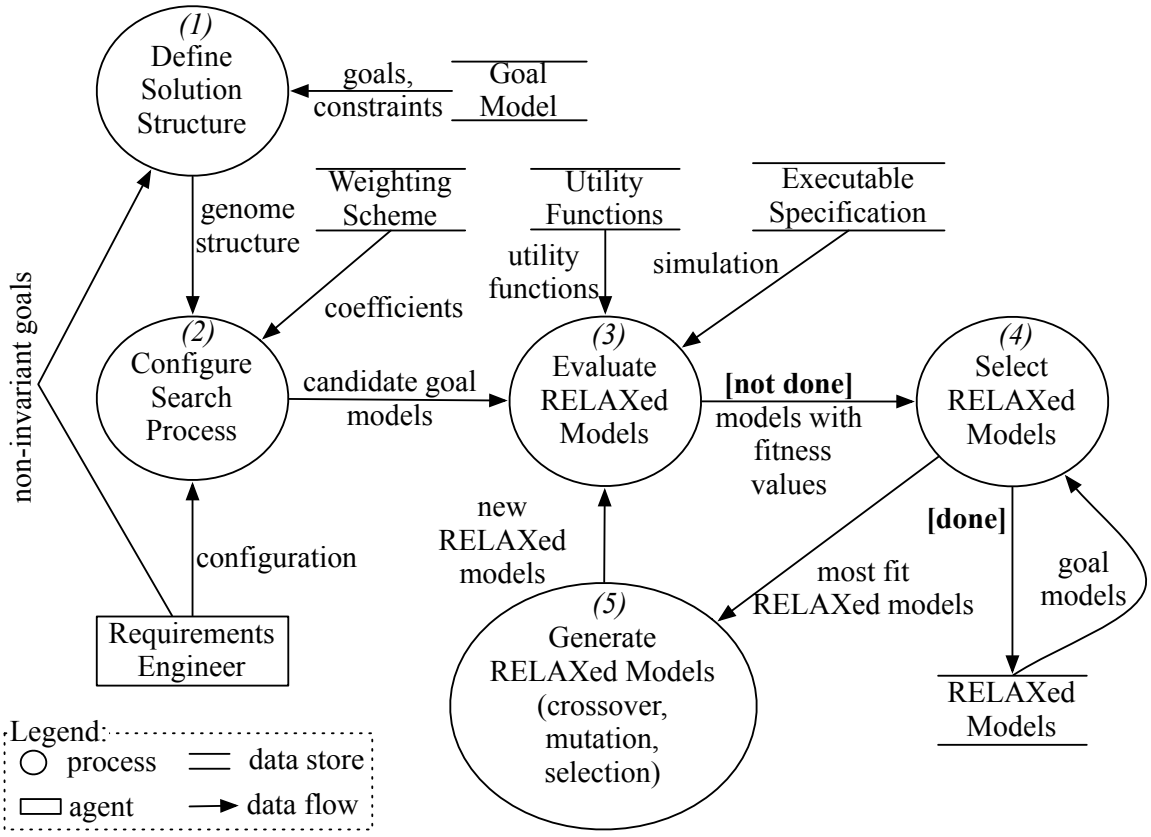
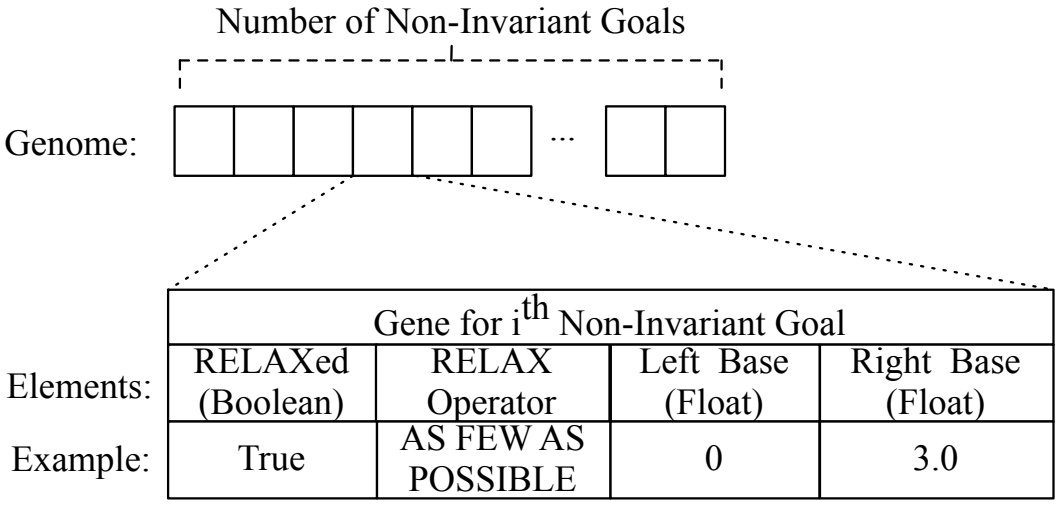


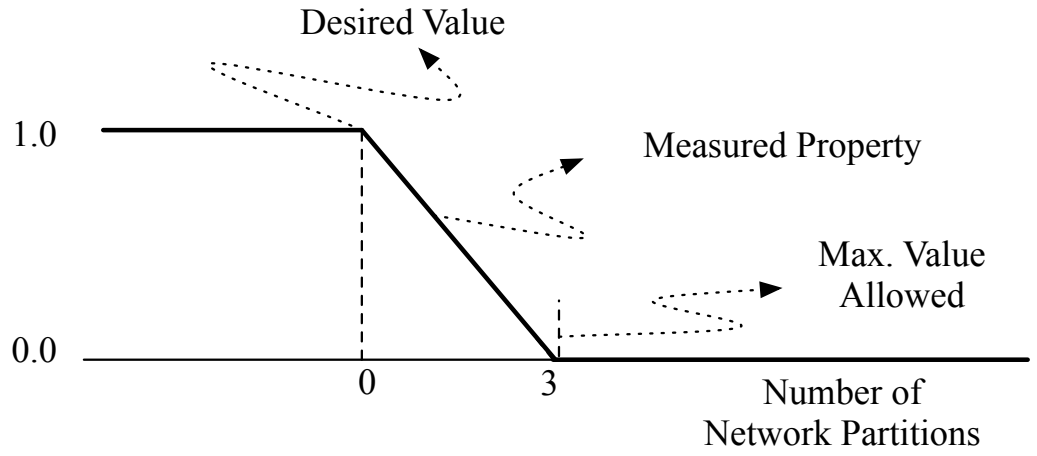
Figure 5.1: DFD diagram of AutoRELAX process

(1) Define Solution Structure. Each candidate solution in AutoRELAX comprises a vector of n elements or *genes*, where n is equal to the total number of non-invariant goals in the KAOS goal model of the DAS. Figure 5.2(A), in turn, shows the structure of each gene. As this figure illustrates, each gene comprises a boolean variable that specifies whether a non-invariant goal will be RELAXed, a corresponding

RELAX operator (see Table 2.2), and two floating point values that define the left and right boundaries of the fuzzy logic function, respectively.



(A) AutoRELAX solution encoding



(B) Mapping a gene to a RELAXed goal

Figure 5.2: Encoding a candidate solution in AutoRELAX

Figure 5.2(B) shows how each gene is mapped to a corresponding fuzzy logic function that can evaluate the satisfaction of a goal. In this example, the unRELAXed satisfaction for Goal (F) in the RDM application (see Figure 2.4) returns 1.0 if the network is connected (i.e., number of network partitions equals zero) and 0.0 otherwise. However, as long as the network partition is transient, then it might be possible

to continue diffusing data amongst connected data mirrors while the network topology is reconfigured. As the bolded lines in Figure 5.2(B) depicts, this goal can be made more flexible by introducing the "AS FEW AS POSSIBLE" ordinal RELAX operator that maps to a *left shoulder*-shaped fuzzy logic function [114]. For this RELAXed goal, the apex is centered upon the ideal value of a system or environmental property, zero network partitions in this case, and the downward slope from the apex to the right endpoint reflects values that are not ideal but might be temporarily tolerated at run time.

In terms of complexity, an individual with this representation scheme comprises a vector of n genes that can be configured to RELAX non-invariant goals in the model. Applying the six operators defined in the RELAX specification language yields a *base* solution landscape that comprises $2^n * 6^n$ possible configurations. This base representation means that the RDM goal model presented in Figure 3.2 can be RELAXed in 8^8 possible ways. Furthermore, each gene must also specify the left and right bounds that establish the new satisficement criteria for its corresponding non-invariant goal. These bounds are defined as *unbounded* floating-point values, thus adding significant complexity to the solution space. Overall, the total number of possible configurations makes it infeasible to exhaustively evaluate all possible configurations in a reasonable amount of time.

(2) Configure Search Process. A requirements engineer must configure AutoRELAX by specifying a population size, crossover and mutation rates, and a termination criterion. The population size determines how many candidate RELAXed goal models AutoRELAX can explore in parallel during each generation; the crossover and mutation rates specify how AutoRELAX will generate new RELAXed goal models; and the termination criteria specifies when AutoRELAX will stop searching for new solutions and output the resulting RELAXed goal models.

(3) Evaluate RELAXed Models. To evaluate the quality of a RELAXed

goal model, AutoRELAX first maps the RELAX operators encoded in an individual to their corresponding utility functions for requirements monitoring in the executable specification (see Step 1). Next, AutoRELAX simulates the executable specification and records the satisfaction of each goal as well as the number of adaptations performed by the DAS. Two fitness sub-functions use this information to reward candidate RELAXed goal models for minimizing the number of RELAXed goals as well as how many adaptations are triggered by minor environmental conditions.

The first fitness sub-function, FF_{nrg}, rewards candidate solutions that minimize the number of RELAXed goals in order to limit the introduction of unnecessary flexibility into a goal model:

$$FF_{nrg} = 1.0 - \Big(\frac{|relaxed|}{|Goals_{non-invariant}|}\Big) \tag{5.1}$$

where |relaxed| and $|Goals_{non-invariant}|$ are the number of RELAXed and non-invariant goals in the DAS goal model, respectively. This fitness sub-function explicitly discourages AutoRELAX from unnecessarily introducing RELAX operators.

The second fitness sub-function, FF_{na}, rewards candidate solutions that minimize the number of adaptations performed by the DAS in response to minor and transient environmental conditions, in order to reduce overhead incurred on performance and cost:

$$FF_{na} = 1.0 - \Big(\frac{|adaptations|}{|faults|}\Big) \tag{5.2}$$

where $|adaptations|$ represents the total number of adaptations performed by the DAS, and |faults| measures the total number of adverse environmental conditions introduced throughout a simulation. This fitness sub-function rewards RELAXed goal models that better tolerate unanticipated environmental conditions, thereby reducing the number of adaptations a DAS performs. Reducing the number of adaptations a DAS performs, in turn, also reduces the number of passive and quiescent components

at run time (see Chapter 2). Moreover, by reducing the number of passive and quiescent components at run time, this fitness sub-function seeks to minimize the impact of an adaptation by enabling the DAS to continue providing important functionality to its stakeholders even during the reconfiguration process.

These two fitness sub-functions can be combined with a linear weighted sum:

$$FitnessValue = \begin{cases} \alpha_{nrg} * FF_{nrg} + \alpha_{na} * FF_{na} & \text{iff invariants true} \\ 0.0 & \text{otherwise} \end{cases} \tag{5.3}$$

where the α_{nrg} and α_{na} coefficients reflect the relative importance of each fitness sub-function, the sum of which must equal 1.0. Although fitness sub-functions can be combined in different ways, we find that a linear-weighed sum facilitates the balancing of competing concerns.

Lastly, the fitness value of a RELAXed goal model depends upon the satisfaction of all invariant goals. For example, if a RELAXed goal model in our RDM application does not replicate every data item or does so while exceeding the allocated budget, then its fitness value is 0.0. This penalty ensures that AutoRELAX only outputs *viable* RELAXed goal models that satisfy all invariant goals.

(4) Select RELAXed Models. AutoRELAX uses the fitness value associated with each evaluated RELAXed goal model to *select* the most promising individuals from the population and thus guide the search process towards that area of the solution space. To this end, AutoRELAX applies tournament selection [46], a technique that randomly selects k individuals from the population and *competes* them against one another. The RELAXed goal model with the highest fitness value amongst these k solutions survives onto the next generation.

(5) Generate RELAXed Models. AutoRELAX uses two-point crossover and single-point mutation to generate new RELAXed goal models, which were set to 50% and 40% for this work, respectively. As Figure 5.3(A) shows, two-point crossover

takes two individuals from the population as *parents* and produces two new RELAXed goal models as *offspring*. As this figure illustrates with different shading, two-point crossover exchanges the genes that lie between the boundaries of two randomly chosen indices. In contrast, Figure 5.3(B) shows how single-point mutation takes an individual from the population and randomly modifies the values of a single gene. In this particular example, the effect of the mutation operator is to change a gene such that its corresponding non-invariant goal is now RELAXed with the "AS MANY AS POSSIBLE" RELAX operator. In this manner, while crossover attempts to construct better solutions by combining good elements from existing RELAXed goal models, mutation introduces diverse goal RELAXations that might not be obtainable via the crossover operator alone.

(6) Output RELAXed Models. AutoRELAX iteratively applies steps (3) through (5) until it reaches its generational limit. Then, AutoRELAX outputs one or more RELAXed goal models with the highest fitness values in the population.

5.4 Experimental Results

This section applies AutoRELAX to automatically RELAX goals and thereby address identified sources of uncertainty in the RDM application. First, we state the objective of the experiment, as well as identify possible sources of uncertainty that might cause a DAS to unnecessarily adapt. We then state experimental hypotheses, as well as the RDM simulation configuration used to evaluate these hypotheses. Lastly, we present experimental results and compare these with both unRELAXed and manually RELAXed goal models as baselines.

Experimental Objectives. This experiment, Experiment 5.1, serves two key purposes. First, this experiment evaluates the effectiveness of introducing RELAX operators as an uncertainty mitigation strategy. Second, this experiment evaluates

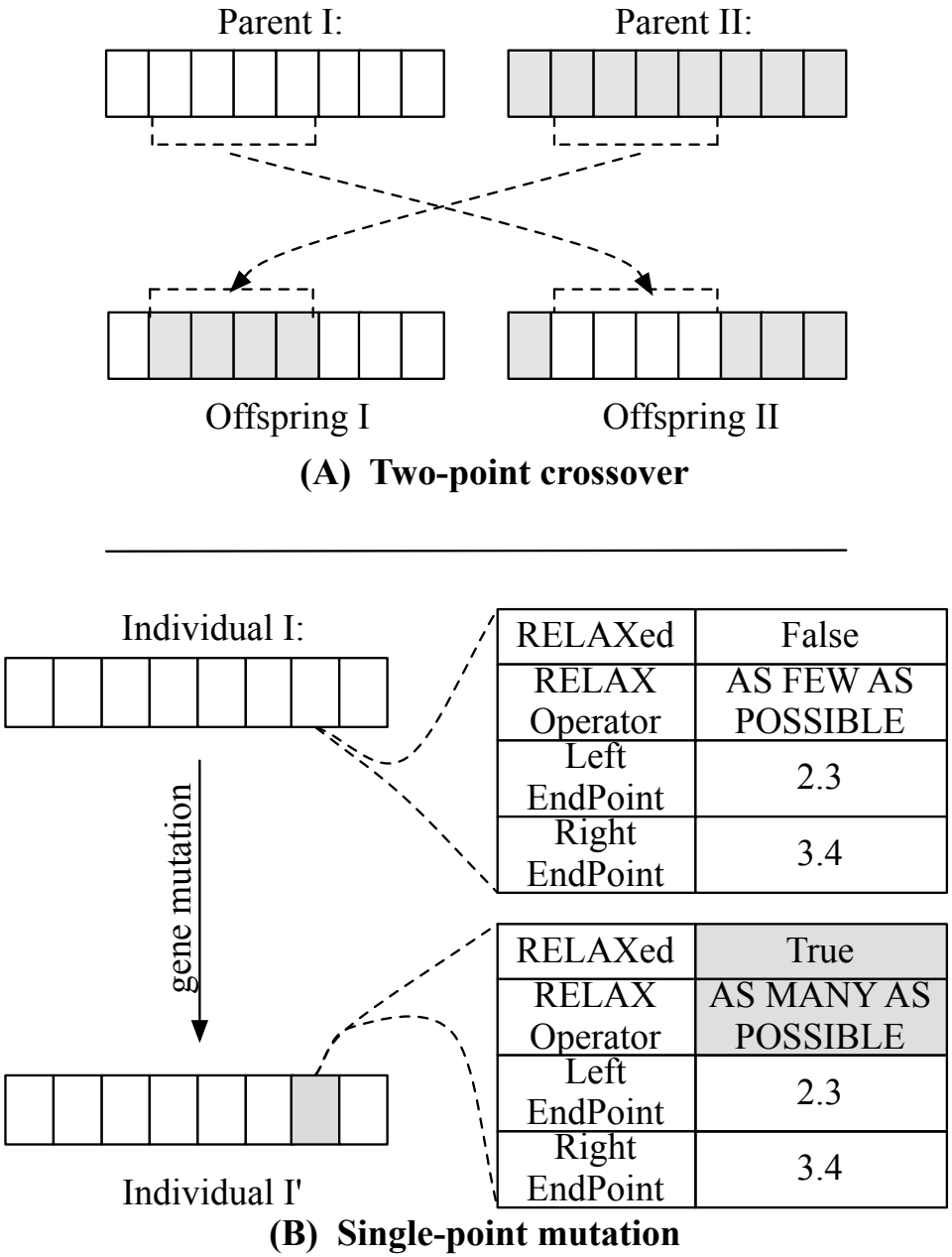


Figure 5.3: Generating new RELAXed goal models with crossover and mutation operators

whether AutoRELAX is capable of automatically generating RELAXed goal models that address sources of system and environmental uncertainty in a DAS. To this end, we compare the quality of goal models generated by AutoRELAX with the quality of unRELAXed and manually RELAXed goal models.

Hypothesis. For this experiment, we defined the first null hypothesis, $H1_0$,

to state that there is no difference in fitness values achieved by a RELAXed and an unRELAXed goal model. In addition, we also defined a second null hypothesis, $H2_0$, 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. Lastly, we also defined an alternate hypothesis, $H2_1$, to state that RELAXed goal models generated by AutoRELAX will achieve a higher fitness value than those manually created by a requirements engineer.

Configuration. This experiment reuses the RDM goal model, utility functions, and executable specification previously used in Chapters 3 and 4. As in the previous chapters, the executable specification can introduce system and environmental uncertainty. Specifically, system uncertainty is introduced in the monitoring infrastructure of the RDM network by fuzzing raw monitoring data and thereby rendering it potentially unreliable. Environmental uncertainty is introduced into the RDM network by failing network links and delaying, dropping, and corrupting data messages. In contrast to previous chapters, the RDM network only executes for 150 time steps and must distribute 20 data items throughout each simulation. These parameters were scaled back to reduce the amount of time required for AutoRELAX to complete and output valid goal models.

As with Experiment 3.2, Experiment 4.1, and Experiment 4.2, system and environmental uncertainty can cause the RDM network to self-reconfigure at run time. Throughout each simulation time step, the RDM network uses collected monitoring data to compute utility values and monitor how well it satisfies its requirements. Unsatisfied goals serve as adaptation triggers that cause the RDM to re-evaluate its current configuration and goal realization strategy. If an adaptation is warranted, then the RDM network selects a more suitable network topology and data propagation configuration. Once a target system configuration is selected, the dynamic change management protocol (see Section 2.2.2) is applied to generate an adaptation

path to safely transition the executing system to its target configuration

For the following experiments we reuse the AutoRELAX fitness sub-functions (i.e., equations 5.1, 5.2, and 5.3) to uniformly compare different goal models that may include goal RELAXations. Reusing these fitness functions for comparing goal models serves two key purposes. First, this objective evaluation criteria enables us to assess the benefits of RELAXing a goal model to address different sources of environmental uncertainty. Second, these fitness functions enable us to demonstrate whether AutoRELAX is capable of generating viable RELAXed models that are as good, if not better, than those manually created by a requirements engineer. Note that for this experiment, we manually balanced competing concerns in the set of fitness functions used by AutoRELAX. Specifically, we set α_{nrg} to 0.3 and α_{na} to 0.7, thus emphasizing the reduction in the number of performed adaptations.

As stated in the objectives, this experiment evaluates and compares the resulting RELAXed goal models produced by AutoRELAX with two different goal models of the same RDM application: the unRELAXed goal model partially shown in Figure 2.4 and several goal models that were manually RELAXed by different requirements engineers. [1] In general, requirements engineers introduced the following five different goal RELAXations into the RDM's goal model: Goal (C) was RELAXed to allow larger exposures to data loss, Goal (D) was RELAXed to add temporal flexibility when diffusing data, Goal (F) was RELAXed to allow up to there simultaneous network partitions, and Goals (I) and (J) were RELAXed to tolerate dropped data messages.

Lastly, since there is a randomized component in both AutoRELAX and the RDM network simulation, we conducted 50 trials of each experiment and, where applicable, plot mean values with corresponding error bars.

Results. Not all adaptations incur the same cost. As such, Figure 5.4 presents three sets of box plots that capture the fitness values achieved by 1) AutoRELAX-

[1] This study involved requirements engineers besides the author of this dissertation.

generated models, 2) a manually created RELAXed goal model, and 3) an unRELAXed goal model, respectively. As these box plots illustrate, despite the fitness boost unRELAXed goal models obtain by not introducing any goal RELAXations (see Equation 5.1, FF_{nrg}), RELAXed goal models achieved statistically significant higher fitness values than unRELAXed goal models ($p < 0.001$, Welch Two Sample t-test). These results enable us to reject our first null hypothesis, $H1_0$, as well as conclude that RELAX does reduce the number of adaptations when addressing system and environmental uncertainty.

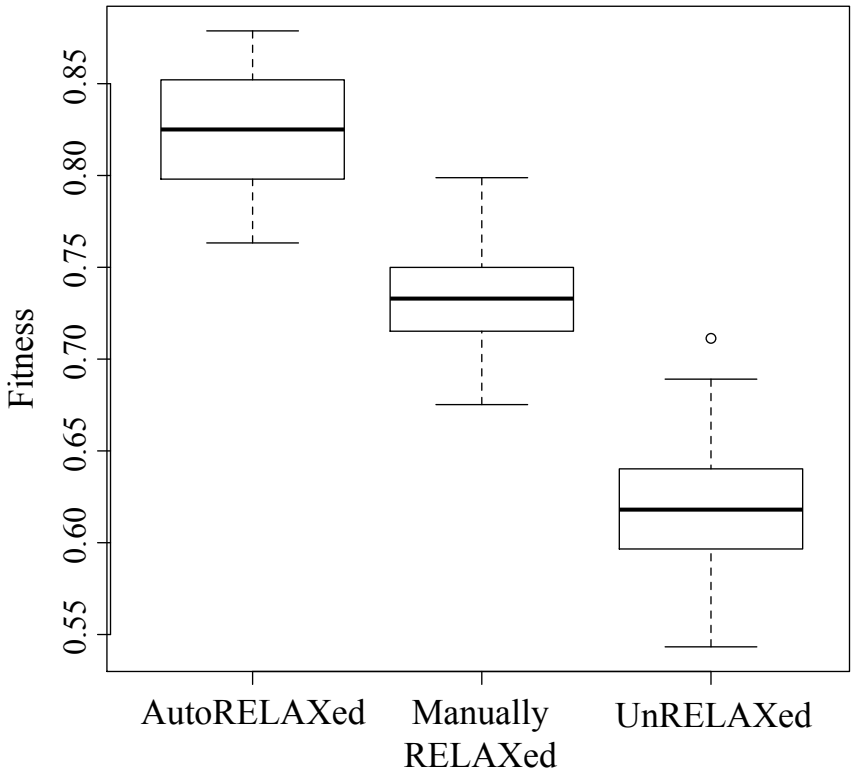


Figure 5.4: Fitness value comparison between AutoRELAX, manually RELAXed and unRELAXed goal models.

The box plots in Figure 5.4 also demonstrate that AutoRELAX generated RELAXed goal models achieved statistically significant higher fitness values than those manually

RELAXed by a requirements engineer ($p < 0.001$, Welch Two Sample t-test). As a result, we also reject our second null hypothesis, $H2_0$, and accept our alternate hypothesis, $H2_1$. These results enable us to conclude that AutoRELAX is capable of generating RELAXed goal models that better address specific sources of uncertainty than manually RELAXed goal models.

Figure 5.5 presents three sets of box plots that capture the adaptation costs incurred by 1) AutoRELAX-generated models, 2) a manually created RELAXed goal model, and 3) an unRELAXed goal model, respectively. Specifically, each set of box plots measures the amount of time that components in the RDM network spent in active, passive, and quiescent modes *during* reconfigurations (these plots do not include time outside of a reconfiguration). As Figure 5.5 shows, option (1) (AutoRELAX) is preferable because it has less negative impact on overall system functionality. Specifically, by carefully lessening the satisfaction criteria of non-invariant goals, the number of adaptations decrease and so does the cumulative amount of time components spend in passive and quiescent modes during a reconfiguration.

Within the RDM application, adaptations are particularly disruptive because data mirrors may be unable to continue the data replication and distribution process required to ultimately satisfy Goal (A). In particular, self-reconfigurations in the RDM network attempt to improve the satisfaction of Goals (C), (D), and (F) by reconnecting a partitioned network and balancing performance and reliability data propagation concerns in response to adverse system and environmental conditions. Nevertheless, adaptations can directly hinder the data diffusion task by placing data mirrors in passive and quiescent states. While passive data mirrors are unable to distribute new data, quiescent data mirrors are unable to receive, replicate, and distribute new data across the network. In this manner, an adaptation can *temporarily* restrict the process of message diffusion such that Goals (C), (D), (G), (H), (I), and (J) are unsatisfied until the reconfiguration completes.

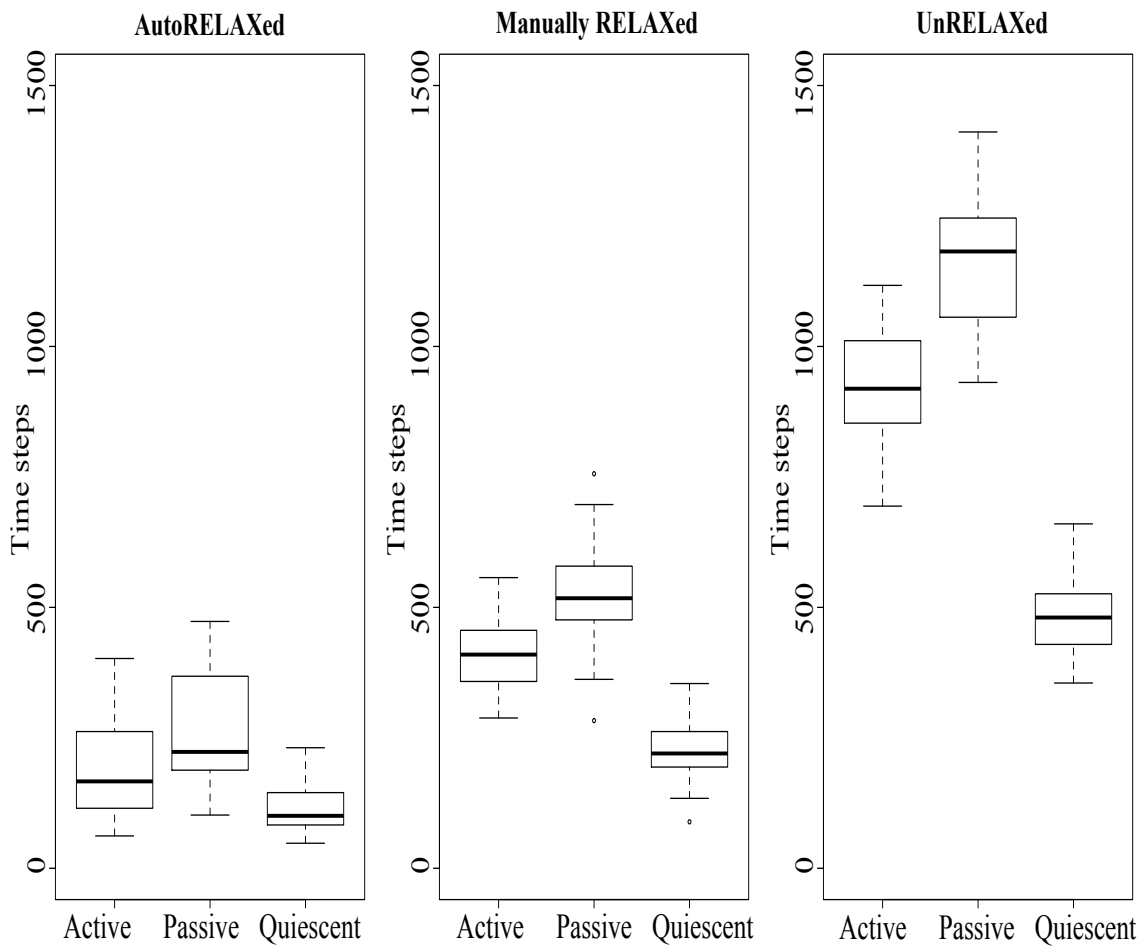


Figure 5.5: Adaptation costs comparison between RELAXed and unRELAXed goal models.

Both Figures 5.4 and 5.5 show that AutoRELAX is able to generate RELAXed goal models that perform better than manually RELAXed goal models. While the manually RELAXed goal model introduced RELAXations to Goals (C), (D), (F), (I), and (J), AutoRELAX mostly introduced RELAX operators to Goals (F), (I), and (J), thereby slightly boosting its fitness value in comparison. Furthermore, the manually RELAXed goal model contained some goal RELAXations that were too constrained. For instance, AutoRELAX was able to extend the goal satisfaction boundary of Goal (F) beyond the bounds applied in the manually RELAXed goal model.

In general, the goal models produced by AutoRELAX were able to tolerate a greater

number of temporary network partitions while allowing components to remain actively distributing data throughout the network. For instance, Figure 5.6 shows an RDM network that becomes partitioned after the network link between data mirrors 3 and 6 failed. Without RELAXing Goal (F), that RDM network would have been immediately self-reconfigured after such a partition occurred. Nevertheless, RELAXing that goal enables the data diffusion process to continue *within* Partition 1 and Partition 2. In this manner, AutoRELAX *coalesces* adaptations in order to reduce functionality disruption at run time.

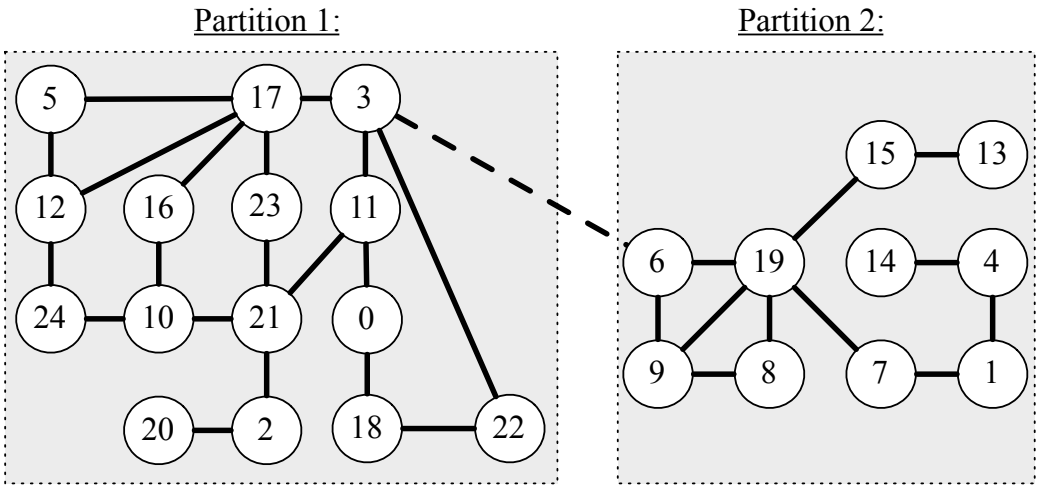


Figure 5.6: Partitioned RDM network that facilitates partial data diffusion.

Figure 5.7 provides additional information about the types of goal models generated by AutoRELAX for varying degrees of uncertainty. In particular, this figure plots the *sorted* number of RELAXed goals per trial where the first environment has a low degree of uncertainty and the second environment has a high degree of uncertainty. As this figure illustrates, in 49 out of 50 trials, AutoRELAX introduced a greater than or equal number of RELAX operators to the goal model subjected to a higher degree of uncertainty than to the goal model subjected to a lower degree of uncertainty. In contrast, in 29 out of 50 trials, AutoRELAX introduced either zero or one RELAX operator to the goal model subjected to a lower degree of uncertainty. Moreover, the positive correlation between the two curves suggest that AutoRELAX gradually

introduces goal RELAXations in response to increasing degrees of system and environmental uncertainty. This plot suggests that AutoRELAX introduces flexibility in how a goal can be satisficed only if necessary.

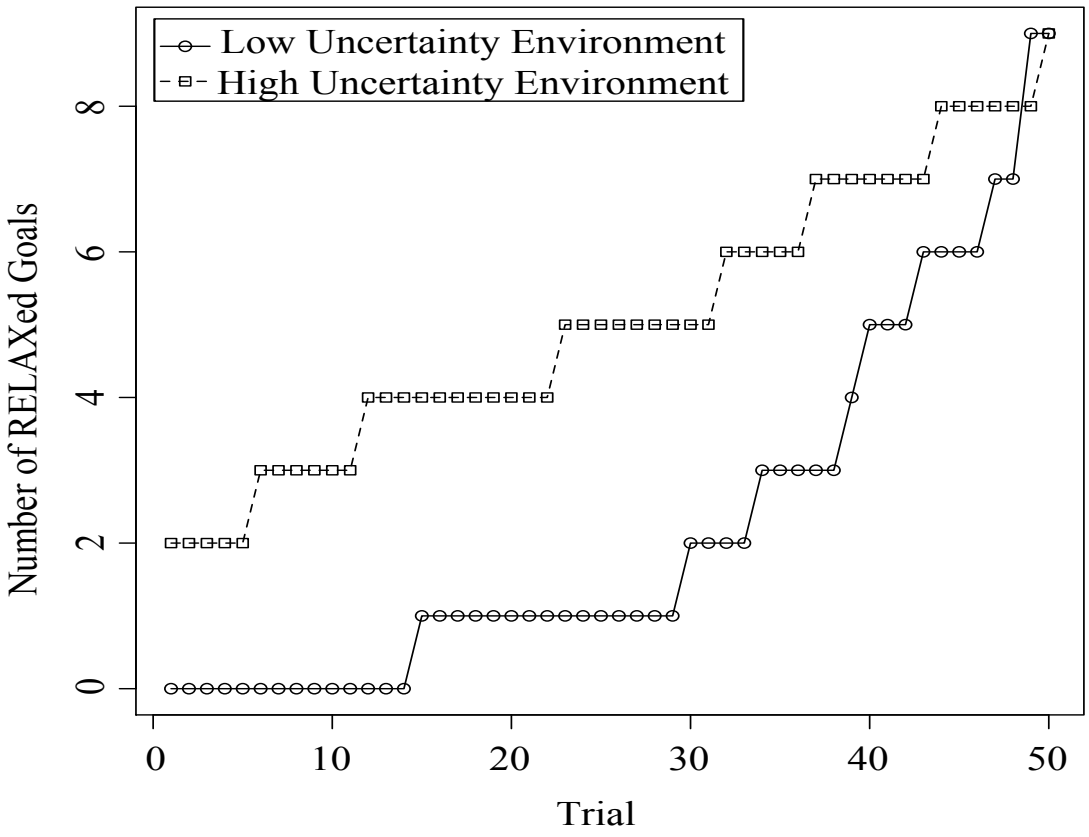


Figure 5.7: Mean number of RELAXed goals for varying degrees of system and environmental uncertainty

Lastly, Figure 5.8 presents a complementary view of the plot shown in Figure 5.7. This plot shows the effects of goal RELAXation upon the number of adaptations triggered during each experiment trial, sorted again in the x-axis by the number of goal RELAXations introduced in the goal model per trial. This figure illustrates that as the number of goal RELAXations increased, the number of adaptations triggered due to system and environmental uncertainty decreased. These observations confirm that

goal RELAXation can prevent possibly unnecessary run-time self-reconfigurations in response to minor and transient environmental conditions. In particular, AutoRELAX introduced RELAX operators that enabled the RDM network to tolerate minor variance in the set of encountered operational contexts. Conversely, as the variance in operational contexts increased, more adaptations were required for the RDM network to continue satisfying its requirements.

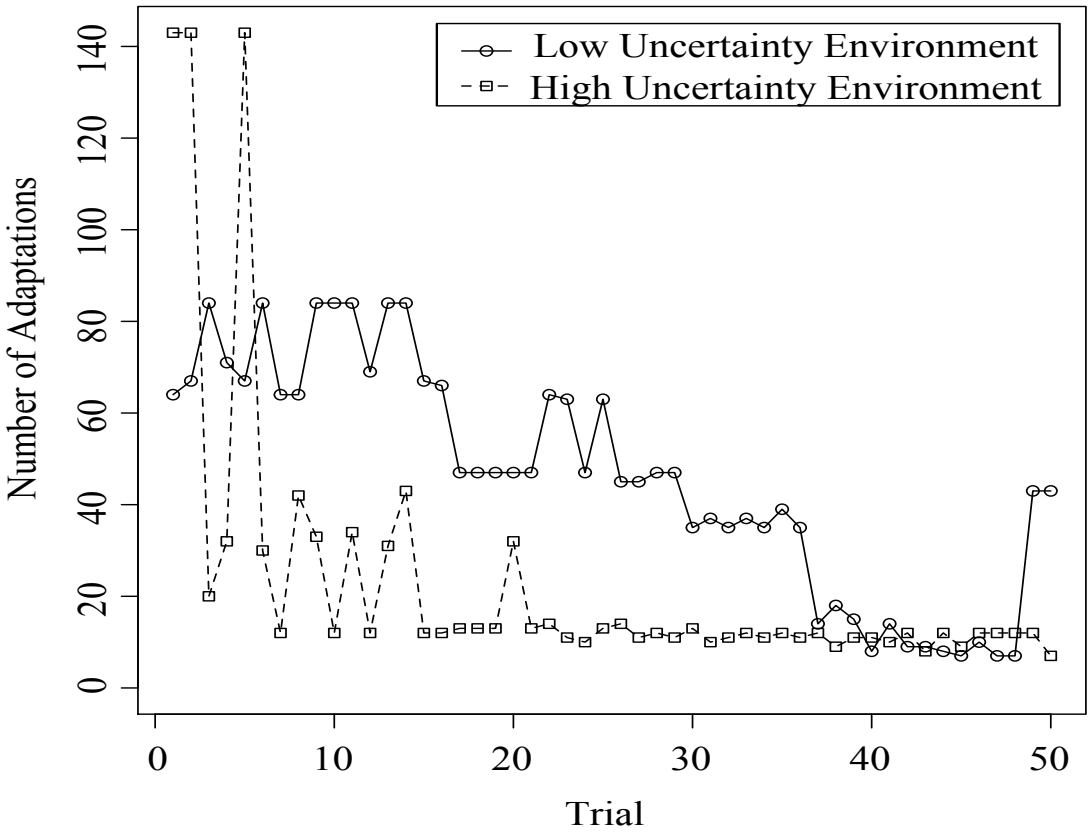


Figure 5.8: Mean number of adaptations triggered, sorted by number of RELAXed goals

5.5 Discussion

Cheng *et al.* [13] introduced four strategies for addressing sources of environmental uncertainty in goal-oriented requirements models of a DAS. The first mitigation strategy revises an unsatisfied goal such that it is no longer affected by the same sources of environmental uncertainty. The second mitigation strategy also revises an unsatisfied goal except it achieves this objective by introducing RELAX operators to add flexibility in how and when a DAS must satisfy its requirements. The third, and possibly most drastic, mitigation strategy introduces a new higher-level goal to prevent or resolve the occurrence of an obstacle that would otherwise lead to unsatisfied goals. The fourth mitigation strategy simply tolerates the obstacle "as is" and does not change the goal model or the DAS in any way.

AutoRELAX automates the second and fourth uncertainty mitigation strategies introduced by Cheng *et al.*. Specifically, AutoRELAX introduces RELAX operators to non-invariant goals to lessen their satisfaction criteria without adversely affecting the satisfaction of invariant goals. In this manner, AutoRELAX automates the second uncertainty mitigation strategy to address adverse environmental conditions that may not have been identified or fully understood at design time. As experimental results suggest, this additional flexility enables adaptive systems to reduce the number of adaptations performed in response to minor and transient sources of uncertainty. Moreover, by reducing the number of adaptations performed, the DAS is able to deliver more functionality to its stakeholders at run time.

This chapter presented one of the first empirical results about how goal RELAXation can affect the abilities of a DAS to satisfy its requirements in the presence of system and environmental uncertainty. Experimental results demonstrate that flexibility in how and when a DAS satisfies its requirements can be desirable. However, too much flexibility can also be detrimental to how a DAS provides its functionality to stakeholders. To this end, AutoRELAX explicitly rewards candidate goal models for

balancing competing concerns between minimizing goal RELAXations and minimizing the number of adaptations triggered due to uncertainty. As such, AutoRELAX only introduces goal RELAXations when it provides a benefit to the DAS. This observation implies that AutoRELAX can and will produce goal models without any RELAX operators if the DAS can handle such system and environmental uncertainty on its own, as was the case in several experiment trials.

Lastly, we fixed the weights for the AutoRELAX fitness sub-functions throughout each experiment presented in this Chapter. However, with a technique such as Loki, a DAS can be subjected to a wide range of system and environmental conditions across different AutoRELAX executions. As such, different weighting schemes may be more suitable than the ones selected for this chapter in order to address each of these specific operational contexts. While this weighting scheme must be currently derived manually by a requirements engineer, evolutionary algorithms can also be leveraged at this stage to automatically optimize the weighting scheme between fitness sub-functions.

5.6 Summary

This chapter discussed AutoRELAX, a component in our model-based framework that automatically generates RELAXed goal models. AutoRELAX leverages a genetic algorithm to explore possible goal RELAXations to be used in order to mitigate system and environmental uncertainties, relieving requirements engineers from having to consider the large amount of possible strategies for handling uncertainty during system design. We applied AutoRELAX to an RDM application that distributes data to all data mirrors while self-reconfiguring as adverse system and environmental conditions arise. Results suggest that AutoRELAXed goal models were as good, if not better, than those manually RELAXed by a requirements engineer or simply not RELAXed at all.

Chapter 6

Generating Reconfigurations

This chapter describes how our model-based framework supports the on-demand generation of target reconfigurations. First, we motivate the need to automatically generate reconfigurations that specify *how* a DAS should modify its system components, and if necessary also its monitoring infrastructure elements, in response to changing system and environmental conditions while balancing non-functional concerns. We then introduce Plato, the component in our model-baed framework responsible for generating target reconfigurations. Next, we use the RDM application to describe how to apply Plato within the decision-making process of a DAS. Subsequently, we present experimental results obtained by applying Plato to the RDM case study. Lastly, we discuss Plato and summarize main findings.

6.1 Motivation

Once an adaptation is triggered, a DAS must then determine precisely how to self-reconfigure in order to either prevent or mitigate the occurrence of an obstacle. A DAS must first determine what kinds of parameter modifications or compositional adaptations must be performed in order to better address current system and environmental conditions [73]. While parameter adaptations fine-tune the behavior of

the DAS, compositional adaptations involve the addition, removal, and modification of structural components. As such, a target system reconfiguration must specify both the structure and behavior that a DAS should exhibit after an adaptation is performed. Moreover, a target system reconfiguration must conform to system and environmental constraints and domain assumptions.

In general, developers have traditionally encoded target reconfigurations at design time, where reconfiguration tasks are influenced by anticipated future execution conditions [15, 39, 53, 109]. These rule-based decision-making engines typically match events and conditions with specific actions (i.e., reconfigurations) that address multiple reconfiguration objectives [15, 20, 35, 53, 99, 109, 117]. For example, in the RDM application, an adaptation engineer might encode several reconfiguration rules to switch network link propagation parameters to synchronous modes in anticipation of possible network link failures at run time. In this manner, the rule-based decision-making engine analyzes monitoring information at run time to detect network link failures and, if applicable, then reconfigures the RDM network by switching to synchronous propagation modes that improve data reliability.

While rule-based adaptation approaches enable efficient mapping of specific conditions to reconfigurations, they only enable a DAS to self-adapt against scenarios considered at design time. Unfortunately, it is often infeasible for an adaptation engineer to identify all possible system and environmental conditions that a DAS may encounter at run time and might warrant an adaptation [13, 110, 111, 112, 114]. Furthermore, as the complexity of adaptive logic grows, designing and managing the set of reconfiguration rules becomes unmanageable and potentially inconsistent [15]. These observations imply that rule-based decision-making engines may be unable to safely reconfigure a DAS in response to unanticipated or poorly understood conditions. For example, a DAS may self-reconfigure elements in its monitoring infrastructure to balance tradeoffs between monitoring costs and accuracy. However, inadequate rule-

based reconfigurations might lead to excessive monitoring where monitoring accuracy improves at the expense interfering with and possibly altering the behavior of a DAS in unpredictable ways. Inadequate rule-based reconfigurations might also lead to insufficient monitoring that may conserve system resources at the expense of failing to detect events leading to a requirements violation.

To address these concerns, several researchers have applied evolutionary computation techniques to the design of adaptive and autonomic systems [39, 41, 61]. Although these approaches enable developers to explore richer sets of structural and behavioral models that satisfy system and adaptation requirements, most are applicable only at design time due to the significant amount of computational time required to evolve target reconfigurations. Moreover, this uncertainty about what conditions a DAS may encounter throughout its lifetime implies that certain adaptation-specific decisions must be deferred until run-time, when a DAS has access to data about current system and environmental conditions to guide its reconfiguration needs. As a result, it is desirable to leverage evolutionary algorithms with light-weight evaluation techniques to generate target system reconfigurations at run time, when actual system and environmental conditions can be observed by a DAS.

6.2 Introduction to Plato

Whenever possible, an adaptation engineer should identify, analyze, and encode target system reconfigurations to handle common adverse system and environmental conditions. Since rule-based adaptations may not be sufficient for an adaptive system to self-reconfigure at run time when it encounters unanticipated operational contexts, an adaptation engineer should also introduce alternate reconfiguration strategies that can be generated on-demand. To this end, evolutionary algorithms can be harnessed at run time to generate target system reconfigurations in response to changes in

system and environmental conditions while simultaneously balancing competing objectives.

Plato is a component in our model-based framework that supports the automatic identification of target system reconfigurations by incorporating a genetic algorithm into the decision-making process of a DAS. Specifically, Plato efficiently searches parts of the vast space of all possible adaptations to identify a target system reconfiguration that satisfies functional concerns while balancing non-functional properties based on current system and environmental conditions. To achieve this objective, Plato integrates monitoring information with a set of domain-specific fitness functions to generate suitable reconfigurations. A key feature of Plato is that developers need not prescribe reconfigurations in advance to address specific situations that warrant reconfiguration. Instead, Plato harnesses the power of Darwinian evolution by natural selection to evolve suitable target reconfigurations at run time, when actual system and environmental conditions can be observed by a DAS.

6.3 Plato Process Description

In this section, we state assumptions that must hold true when applying Plato. We then describe each step that a developer must apply in order to integrate Plato within the decision-making process of a DAS.

6.3.1 Assumptions

Several assumptions must hold true in order for Plato to automatically generate target system reconfigurations in response to changing system and environmental conditions. In particular, we assume that:

- a requirements engineer has constructed a KAOS goal model capturing the functional and non-functional goals of the DAS.

- the genetic algorithm used by Plato can access current system and environmental monitoring data.
- a requirements engineer or developer has identified parameters and components that can be adapted in the DAS.
- domain-specific criteria can be applied to evaluate the viability of a candidate solution in terms of satisfying functional and non-functional requirements.

In addition, the following assumptions must hold true in order for Plato to automatically reconfigure the monitoring infrastructure of a DAS at run time:

- sensors in the monitoring infrastructure of the DAS must be *passive*. In particular, a DAS must pull monitoring data from passive sensors, thereby enabling Plato to control how often monitoring data is collected from each sensor.

6.3.2 Target Reconfiguration Generation Process

We designed Plato with the objective of automatically deciding how a DAS should self-reconfigure in response to changing requirements and environmental conditions. Throughout this process, a developer first identifies parameters and components in the DAS that can be reconfigured and encode these within a genome. Based on this representation, a developer then defines and implements a set of crossover and mutation operators to generate candidate target system configurations. In addition, a developer also configures the search parameters of the genetic algorithm and implements domain-specific fitness sub-functions to evaluate how each candidate target system configuration addresses current system and environmental conditions.

We continue to reuse the RDM application as a working example to present and describe each step that must be performed in order to integrate Plato with the decision-making process of a DAS. In particular, the RDM application leverages Plato

to automatically reconfigure system components, such as RDMs and network links, as well as elements in its monitoring infrastructure, such as RDM and network link sensors.

Select Representation. Developers start by selecting a suitable representation scheme for encoding candidate solutions as individuals in a genetic algorithm. For example, in the RDM application, each individual in the population encodes a complete overlay network, where each link is either active or inactive and is associated with one of seven possible propagation methods (see Table 2.1). In particular, synchronous propagation (time interval equal to 0) provides the maximum amount of data protection while asynchronous propagation with a 24-hour time interval provides the least amount of data protection. Nonetheless, as the level of data protection increases, the overall performance of the network decreases. Specifically, as the average data batch size decreases, fewer opportunities arise for coalescing data writes. As a result, synchronous propagation methods typically incur worse network performance than asynchronous propagation methods with larger data batch sizes [55].

For this application domain, Plato uses a representation scheme similar to the one illustrated in Figure 6.1, where each overlay network link can be set to either active or inactive and is associated with one of the seven propagation methods presented in Table 2.1. In terms of complexity, an individual with this representation scheme comprises a vector of $\frac{n(n-1)}{2}$ links that can be activated and configured to form an overlay network. With 25 nodes, for example, there are $2^{300} * 7^{300}$ possible overlay network configurations. The total number of possible configurations makes it infeasible to exhaustively evaluate all possible configurations in a reasonable amount of time.

Likewise, Figure 6.2 shows another presentation scheme that Plato uses in order to reconfigure the monitoring infrastructure of the RDM network at run time. As this figure illustrates, Plato represents each candidate monitoring configuration as a

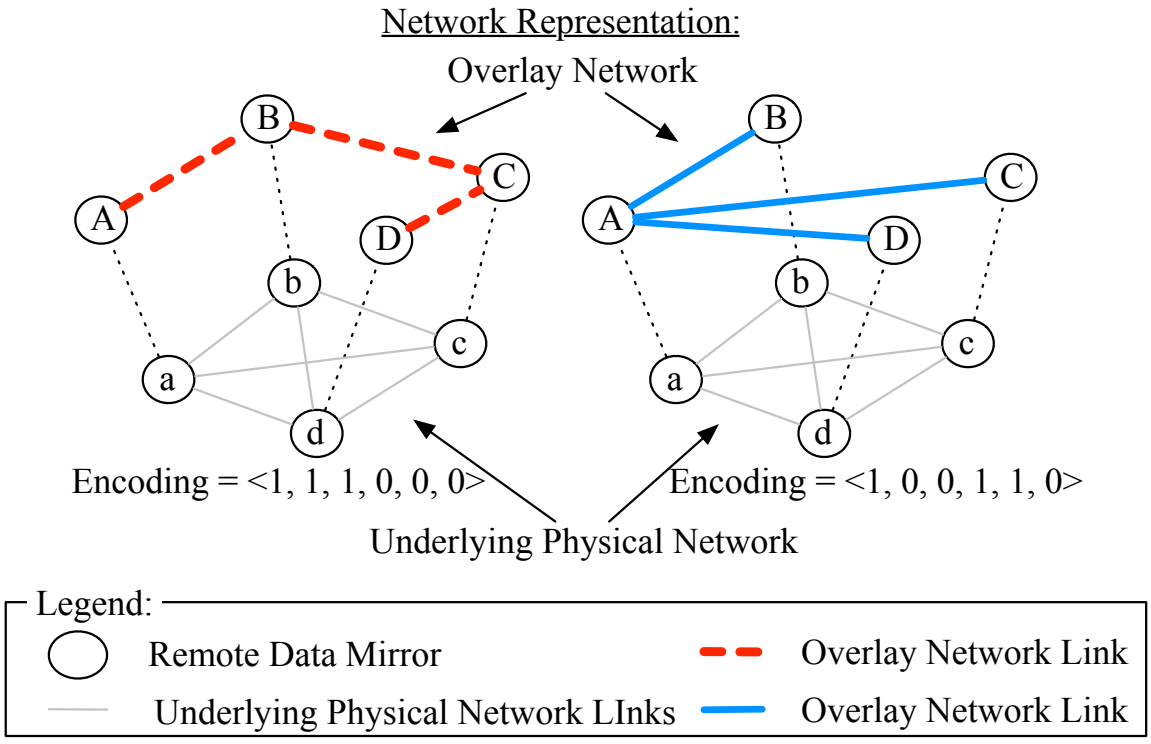


Figure 6.1: Encodings of two overlay network solutions as individuals in a genetic algorithm.

fixed-length vector of integers. Each integer in this vector encodes how often each system component and sensor should probe or gather data. Depending upon software or hardware limitations, each frequency value in this vector may need to be specified as a multiple of some time unit. For example, each monitoring frequency in the RDM application must be specified as a multiple of the predefined time_step unit. As a result of this representation scheme, each frequency in the vector corresponds to an integer within the bounds of 1 and f_{max}, the maximum timespan allowed between monitoring data gathering.

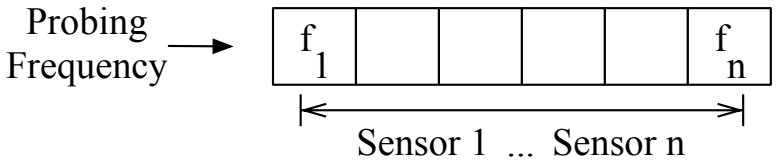


Figure 6.2: GA encoding for monitoring configurations.

As with the encoding scheme selected for representing the RDM network itself, the encoding defined for the monitoring infrastructure of the RDM network also defines the complexity of the solution space that the genetic algorithm will search at run time to generate new monitoring configurations. In particular, if we define f_{max} as above, and n to be the number of sensors or components that can be probed for monitoring data, then exactly $(f_{max})^n$ possible monitoring configurations can be generated. As an example, with 25 RDMs and 300 network links, each one configurable with 10 different probing frequencies (f_{max}), the solution space would comprise over 1.3147×10^{25} different monitoring configurations. The vast number of possible monitoring configurations motivates the need for efficient search heuristics, such as a genetic algorithm, to generate near-optimal solutions in a short amount of time.

GA Operators. Crossover and mutation operators are specific to the encoding scheme used. The default crossover and mutation operators are designed to work on fixed-length binary string representations [46]. If a different representation scheme is used to encode candidate solutions for a particular domain, then specialized crossover and mutation operators need to be developed and applied. For example, for this application domain each individual in Plato encodes a candidate solution that comprises binary, integer, and floating-point values. As a result, Plato uses domain-specific crossover and mutation operators to directly manipulate overlay network topologies. The crossover operator used by Plato in this application, as shown in Figure 6.3, exchanges link properties between two parents (underlined elements represent portion of genome affected). Specifically, two network links in the link vector are selected at random, and the segments between them are exchanged, thereby producing two new candidate network configurations

In addition to the crossover operator, a genetic algorithm uses a mutation operator to introduce variation into the current population. Figure 6.4 presents the mutation operator used for this network-based representation. As this figure illus-

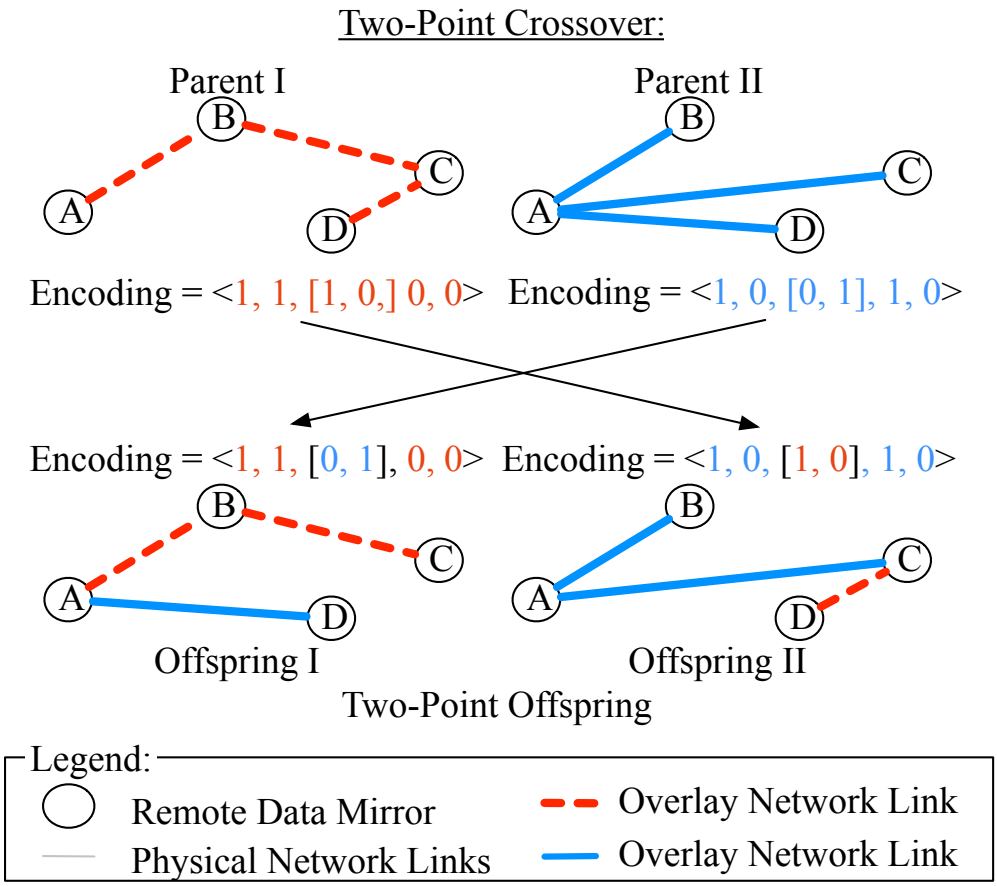


Figure 6.3: Crossover operator for network-based representation.

trates, a mutation can randomly activate or deactivate a network link, as well as change its propagation method.

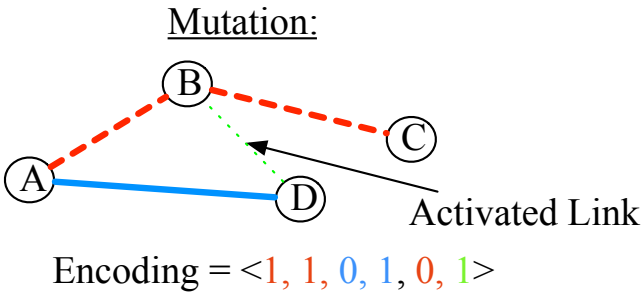


Figure 6.4: Mutation operator for network-based representation.

GA Setup. A developer also configures the genetic algorithm for the problem being solved. Typical parameters include population size, crossover type, crossover and mutation rates, selection methods, and the allotted execution time, typically

expressed in generations. To this end, Plato uses two-point crossover and single point crossover. In addition, Plato applies tournament selection to select which individuals *survive* to the next generation. In tournament selection, the fitness value of two randomly selected individuals from the current population are compared and the individual with the highest fitness value survives onto the next generation.

Fitness Sub-Functions. In general, a single fitness function is not sufficient to quantify all possible effects of a particular reconfiguration when balancing multiple objectives [15, 21]. Instead, developers should define a *set* of fitness sub-functions to evaluate a target reconfiguration according to the optimization dimensions specified by end-users. Each fitness sub-function should have an associated coefficient that determines the relative importance of that sub-function in comparison to others. As in previous techniques in our model-based framework, a weighted sum can be used to combine the values obtained from each fitness sub-function into one scalar value [27, 70, 76].

Next, we present two collections of fitness functions that Plato uses to evaluate the quality of target reconfigurations for the RDM network and its monitoring infrastructure, respectively. Specifically, these two collections of fitness functions are orthogonal to each other. Subsequently, the RDM application executes two independent Plato instances to reconfigure both the network topology and its propagation parameters, as well as its monitoring infrastructure.

Network Reconfiguration Fitness Sub-Functions. For the RDM application, Plato uses several fitness sub-functions to approximate the effects of a particular overlay network in terms of costs, network performance, and data reliability. Most of the fitness sub-functions used by Plato were derived from studies on optimizing data reliability solutions [55] and modified for our specific problem. This set of sub-functions enables Plato to search for overlay networks that not only satisfy the previously mentioned properties, but that also yield the highest fitness based on how the

end-user defined the sub-function coefficients to reflect the priorities for the various fitness sub-functions.

We use the following fitness sub-function to calculate an overlay network's fitness in terms of *cost*:

$$F_{cost} = 100 - \left(100 * \frac{cost}{budget}\right) \tag{6.1}$$

where cost is the sum of operational expenses incurred by all active links, and budget is an end-user supplied constraint that places an upper bound on the maximum amount of money that can be allocated for operating the overlay network. This fitness sub-function, F_{cost}, guides the genetic algorithm toward overlay network designs that minimize operational expenses.

Likewise, we use the following two fitness sub-functions to calculate an overlay network's fitness in terms of *performance*:

$$F_{perf1} = 50 - \left(50 * \frac{latency_{avg}}{latency_{wc}}\right), \tag{6.2}$$

and

$$F_{perf2} = 50 * \left(\frac{band_{sys} - band_{\mathit{eff}}}{band_{sys} + bound}\right), \tag{6.3}$$

where $latency_{avg}$ is the average latency over all active links in the overlay network, $latency_{wc}$ is the largest latency value measured over all links in the underlying network, $band_{sys}$ is the total available bandwidth across the overlay network, $band_{\mathit{eff}}$ is the total effective bandwidth across the overlay network after data has been coalesced, and *bound* is a limit on the best value that can be achieved throughout the network in terms of bandwidth reduction. The first fitness sub-function, F_{perf1}, accounts for the case where choosing links with lower latency will enable faster transmission rates. Likewise, the second fitness sub-function, F_{perf2}, accounts for the case where network performance can be increased by reducing the amount of data sent across a network due to write coalescing. We note that the maximum achievable value of F_{perf1} +

F_{perf2} is 100.

Lastly, we use the following two fitness sub-functions to calculate an overlay network's fitness in terms of *reliability*:

$$F_{rel1} = 50 * \Big(\frac{links_{used}}{links_{max}}\Big); \tag{6.4}$$

and

$$F_{rel2} = 50 - \Big(50 * \frac{dataloss_{potential}}{dataloss_{wc}}\Big); \tag{6.5}$$

where $links_{used}$ is the total number of active links in the overlay network, $links_{max}$ is the maximum number of possible links that could be used in the overlay network given the underlying network topology, $dataloss_{potential}$ is the total amount of data that could be lost during transmission as a result of the propagation method (see Table 2.1), and $dataloss_{wc}$ is the amount of data that could be lost by selecting the propagation method with the largest time window for write coalescing. The first reliability fitness function, F_{rel1}, accounts for the case where an overlay network with redundant links may be able to tolerate link failures while maintaining connectivity. The second reliability fitness function, F_{rel2}, accounts for the case where propagation methods leave data unprotected for a period of time. We note that the maximum achievable value of $F_{rel1} + F_{rel2}$ is 100, the same as the fitness sub-functions for *cost* and *performance*.

The following fitness function combines the previous fitness sub-functions into one scalar value:

$$FF = \alpha_{cost} * F_{cost} + \alpha_{perf} * (F_{perf1} + F_{perf2}) + \alpha_{rel} * (F_{rel1} + F_{rel2}), \tag{6.6}$$

where α_i's represent weights for each dimension of optimization as encoded into the genetic algorithm by the end user. These coefficients can be scaled to guide the genetic

algorithm towards particular designs that reflect different priorities of non-functional properties. For example, if developers want to evolve types of overlay network designs that optimize only with respect to *cost*, then α_{cost} could be set to 1 and α_{perf} and α_{rel} could be set to 0.

Adaptive Monitoring Fitness Sub-Functions. In addition, for the RDM application, Plato uses the following fitness sub-functions to evaluate the quality of candidate monitoring configurations. Each fitness sub-function focuses on a single concern when evaluating monitoring configurations. The first set of fitness sub-functions focus on minimizing monitoring costs, as measured by energy consumption or the amount of times a sensor is probed. In contrast, the second set of fitness sub-functions maximizes the monitoring data accuracy by probing system components and sensors more frequently. Combined, these two sets of fitness sub-functions search for monitoring configurations that maximize monitoring accuracy while minimizing monitoring costs.

We use the following fitness sub-function, F_e, to evaluate a monitoring configuration from the perspective of minimizing monitoring costs:

$$F_e = 100 - 100 * \Big(\frac{\sum_{i=1}^{|sensors|} cost(i) * freq(i)}{max\_cost} \Big) \tag{6.7}$$

where |sensors| is the number of sensors and components that produce monitoring data, $cost(i)$ returns the energy consumed by probing the i^{th} sensor, $freq(i)$ measures the number of times a sensor is probed within a specified time frame, and max_cost measures the maximum amount of energy that can be consumed by configuring all sensors to probe for monitoring data as often as possible. As such, this fitness sub-function guides Plato towards solutions that reduce the monitoring frequency of most sensors, thus minimizing monitoring costs.

We use the following fitness sub-function, F_{a1}, to evaluate a monitoring config-

uration from the perspective of maximizing the coherency of monitoring data:

$$F_{a1} = 50 * \Big(\frac{\sum(f_i * \phi_i)}{|sensors| * f_{max}}\Big) \tag{6.8}$$

where |sensors| and f_{max} are the same as previously defined, f_i is the frequency at which the i^{th} sensor gathers data, and ϕ_i is a value between 0 and 1 proportional to the change in monitoring values of the i^{th} sensor. In particular, ϕ_i is computed as a ratio between the magnitude of change in gathered values and the current value reported by the i^{th} sensor. This fitness sub-function rewards monitoring configurations that increase probing frequencies of sensors whose monitoring data indicates changing system and environmental conditions. As a result, this fitness sub-function guides Plato towards solutions that proportionally allocate monitoring resources towards monitoring components and sensors reporting changing system and environmental conditions.

Similarly, the following fitness sub-function, F_{a2}, evaluates a monitoring configuration from the perspective of maximizing the coverage of monitoring data:

$$F_{a2} = 50 * \Big(\frac{\sum |fi_{prev} - fi_{new}|}{|sensors| * (f_{max} - 1)}\Big) \tag{6.9}$$

where |sensors| and $f_{\max}$ are the same as previously defined, fi_{prev} is the current probing frequency for the ith sensor, and fi_{new} is the probing frequency for the ith sensor as specified by the generated monitoring configuration. This fitness sub-function rewards monitoring configurations that allocate monitoring resources to sensors currently probed at low frequencies. As a result, this fitness sub-function guides Plato towards solutions that cycle the set of sensors that are probed at high frequencies, thus maximizing monitoring coverage.

As with the set of fitness sub-functions for reconfiguring the topology and propagation parameters of the RDM network, the set of fitness sub-functions for adaptive

monitoring can be combined using a linear-weighed sum as follows:

$$F = \alpha_e * (F_e) + \alpha_a * (F_{a1} + F_{a2}) \tag{6.10}$$

If requirements are likely to change while the application executes, then developers should introduce code to *rescale* the coefficients of individual fitness sub-functions at run time. In particular, the fitness landscape is shifted when the coefficients of a fitness sub-function are rescaled. By updating the relevance of each fitness sub-function at run time, the genetic algorithm will be capable of evolving target reconfigurations that address changes in requirements and environmental conditions without specifying *how* the system should be reconfigured. For example, when an overlay network link fails, Plato automatically doubles the current coefficient for network reliability. Note that although we prescribe how these coefficients should be rescaled in the RDM case study in response to high-level monitoring events, our model-based framework does not explicitly specify target reconfigurations. That is, Plato does not prescribe how many network links should be activated in the overlay network nor what their propagation methods ought to be.

6.4 Case Study

This section applies our model-based framework to the dynamic reconfiguration of both a network of RDMs, as well as its monitoring infrastructure. In particular, the following experiments are intended to study whether Plato can evolve target system reconfigurations for RDMs operating across dynamic and unreliable networks. Each of these experiments focuses on a single aspect of this problem, namely constructing and maintaining an overlay network that enables the distribution of data to all nodes. Different environmental factors and scenarios presented throughout these experiments provide insight with respect to the suitability of genetic algorithms for supporting the

decision-making process of an adaptive and autonomic system. For each set of results presented, we performed 30 trials of the experiment to account for the stochastic variation of genetic algorithms. Each experiments presented throughout this section was run on a MacBook Pro with a 2.53GHz Intel Core 2 Duo Processor and 4GB of RAM. Moreover, each plot presents mean values obtained from these trials.

Single Dimensional Optimization

Experimental Objective. This experiment, Experiment 6.1, confirms that, for degenerate scenarios involving single fitness sub-functions (i.e., operational cost, performance, and reliability), Plato will produce solutions consistent with those that can be predicted.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that Plato *will not generate viable target system reconfigurations.* We also defined an alternate hypothesis, H_1, that states that Plato *will generate viable target system reconfigurations.* Here, the viability of a target system reconfiguration is evaluated by using the same fitness sub-functions for network reconfigurations (i.e., equations 6.1-6.5). Moreover, the viability of a specific solution also depends on the weighting scheme used for each fitness sub-function.

Configuration. In this experiment we configured Plato to only consider a single optimization dimension at once. That is, we scaled the network reconfiguration fitness sub-functions as a zero-sum game where only one optimization dimension, such as minimizing operational costs or maximizing network reliability, can be maximized.

Results. First, we explored whether Plato was able to minimize operational costs. Specifically, we set the network reconfiguration fitness sub-function coefficients as follows: $\alpha_{cost} = 1$, $\alpha_{perf} = 0$, $\alpha_{rel} = 0$. As a representative example, consider the evolved overlay network shown in Figure 6.5. This overlay network comprises 24 links and connects all remote data mirrors. Thus, the genetic algorithm was able to

reduce the overlay network to a spanning tree that connects all remote data mirrors while incurring operational costs significantly below the maximum allocated budget.

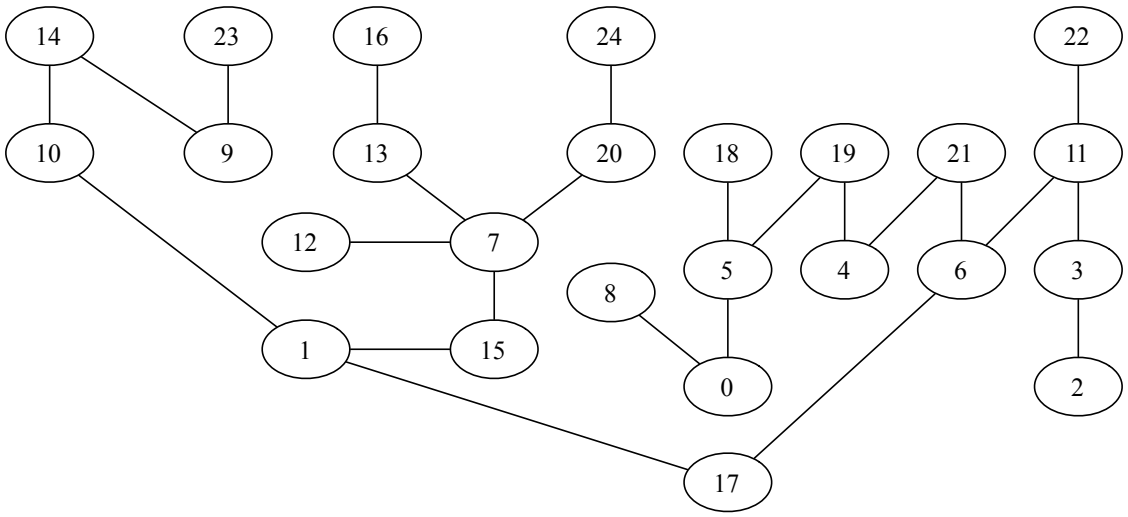


Figure 6.5: Overlay network produced when optimizing for cost.

Figure 6.6 shows the maximum fitness achieved by the candidate overlay networks as Plato executed. Plato converged upon an overlay network topology with a fitness value of approximately 50, indicating that Plato found overlay networks whose operational costs were roughly 50% of the allocated budget. Although the first few hundred generations obtained negative fitness values due to ill-formed candidate topologies that were either disconnected or exceeded the allocated budget, Plato found suitable overlay network designs by generation 500 (approximately 30 seconds), well within the practical range for applications such as remote data mirroring.

Next, we explored whether Plato was able to maximize network reliability. Specifically, we set the network reconfiguration fitness sub-function coefficients as follows: $\alpha_{cost} = 0$, $\alpha_{perf} = 0$, $\alpha_{rel} = 1$. The evolved overlay network provides the maximum amount of reliability possible by activating all 300 links, thereby constructing a complete overlay network. Furthermore, the dominant propagation method for this overlay network was synchronous propagation, which minimizes the amount of data that can be lost during transmission.

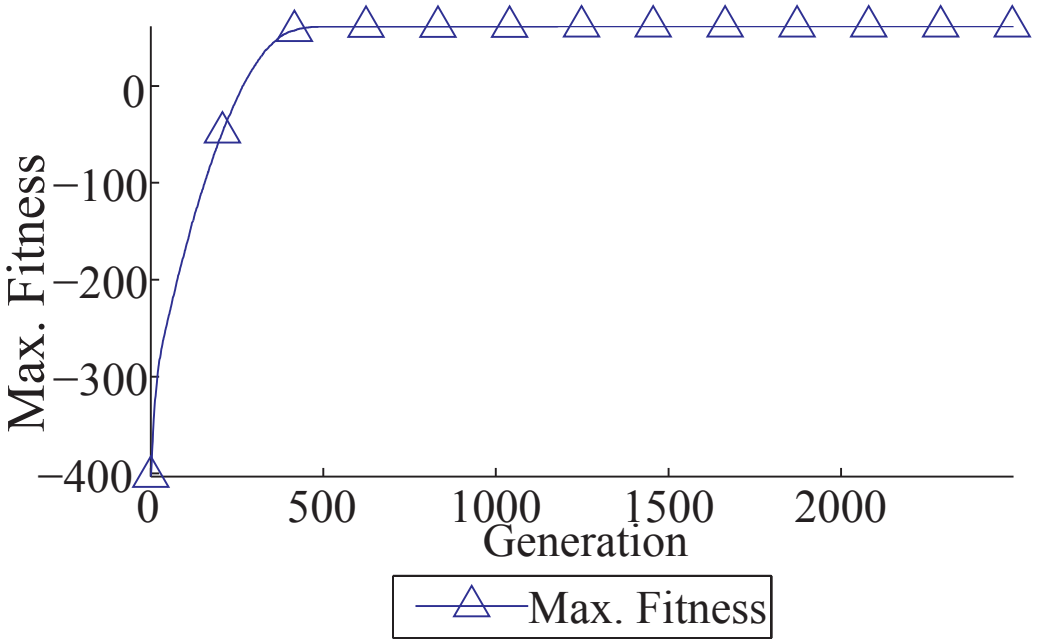


Figure 6.6: Fitness of overlay networks when optimizing for cost only.

Figure 6.7 plots the maximum fitness achieved by Plato in this experiment. Specifically, Plato converged upon a maximum fitness value of 88. In the context of reliability, a value of 88 means that although the overlay network provides a high-level of data reliability, it is not completely immune against data loss. Although all 300 links were activated in the overlay network to provide redundancy against link failures, not every link in the overlay network used a synchronous propagation method. Instead, a few links in the overlay network used asynchronous propagation methods with 1 and 5 minute time bounds. Nonetheless, we note the rapid development of fit individuals achieved by generation 600; by this point, Plato had evolved complete overlay networks with most links using synchronous propagation.

As these results demonstrate, Plato was able to evolve viable target system reconfigurations for both scenarios considered. As such, these results enable us to reject our null hypothesis, H_0 (t-test, $p < 0.05$). In addition, when minimizing operational costs, Plato consistently generated target RDM networks that comprised a spanning tree of data mirrors, thus reducing operational costs. Similarly, when maximizing

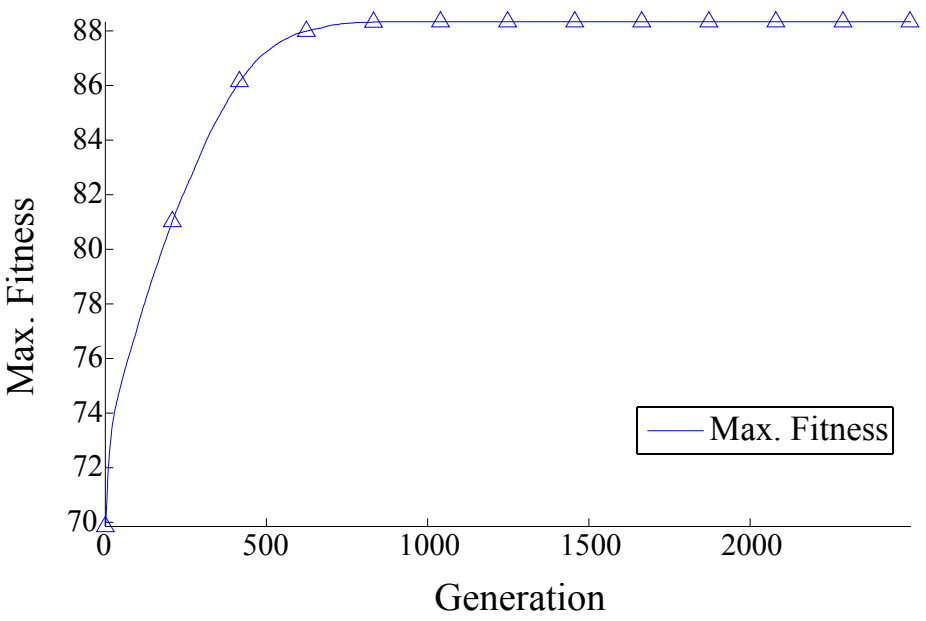


Figure 6.7: Fitness of overlay networks when optimizing for reliability only.

network reliability, Plato consistently generated target RDM networks that formed a complete graph with most network links using synchronous propagation methods. These results enable us to accept our alternate hypothesis, H_1, as Plato discovered viable RDM network configurations that satisfied requirements as specified by the scaled fitness sub-functions (t-test, $p < 0.05$).

Multi-Dimensional Optimization

This experiment, Experiment 6.2, evaluates whether Plato is able to efficiently balance multiple objectives, such as minimizing operational costs while maximizing network performance and data reliability. For this experiment we configured Plato to produce network designs that emphasize network performance and reliability over operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 2$, $\alpha_{\text{rel}} = 2$. Figure 6.8 shows a representative overlay network design that Plato evolved. This overlay network comprises 32 active

links, the majority of which uses asynchronous propagation methods with 1 and 5 minute time bounds. Overall, this overlay network provides a combination of performance and reliability while keeping operational expenses well below the allocated budget.

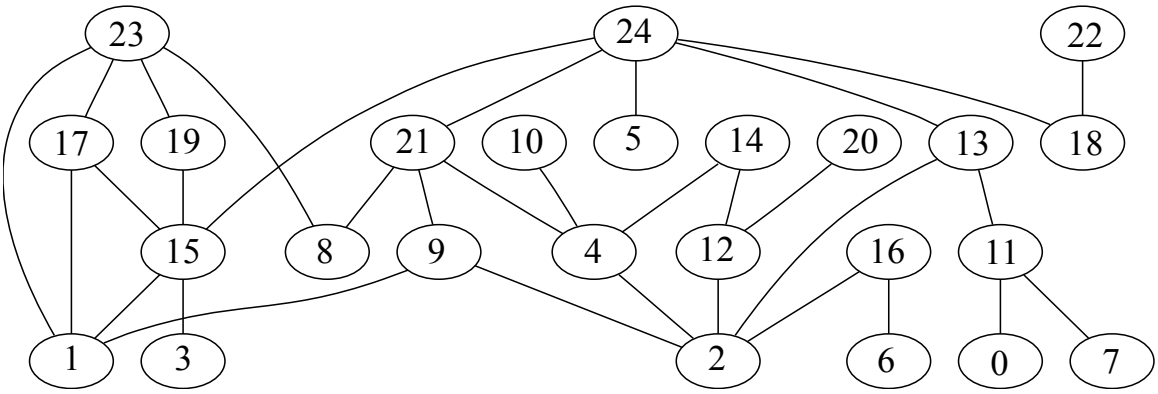


Figure 6.8: Overlay network produced when optimizing for cost, performance, and reliability.

Figure 6.9 plots the average rate at which Plato converged on the resulting overlay network designs. On average, Plato terminated within 3 minutes. In particular, Plato found relatively fit overlay networks by generation 500 (approximately 30 seconds). Thereafter, Plato fine-tuned the overlay network to produce increasingly more fit solutions.

Figure 6.10 provides additional information about the fitness achieved in Figure 6.9 by plotting the mean number of active links in the evolved overlay network. At first, fitter overlay networks comprised the fewest number of active links while still maintaining connectivity. Before Plato terminated, 8 additional links had been added to the overlay network. Although these additional edges increased the overall operational cost of the overlay network, they also increased the network's fault tolerance against link failures, thus improving the overlay's reliability fitness value. Moreover, subsequent generations achieved higher fitness values by using asynchronous propagation methods of 5 minutes and 1 hour, thus improving network performance while providing some level of data protection during transmission.

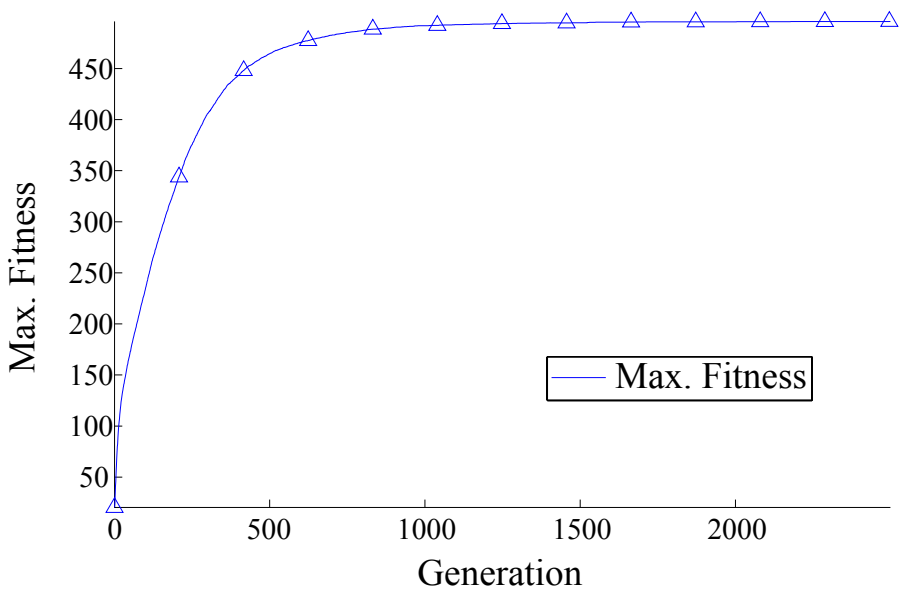


Figure 6.9: Maximum fitness of overlay networks when optimizing for cost, performance, and reliability.

Reconfiguration Against Link Failures

Experimental Objective. This experiment, Experiment 6.3, evaluates the feasibility of using Plato to dynamically reconfigure the overlay network topology in real-time.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that Plato *will not generate viable RDM network reconfigurations in response to adverse system and environmental conditions.* We also defined an alternate hypothesis, H_1, that states that Plato *will generate viable RDM network reconfigurations in response to adverse system and environmental conditions.* As with Experiment 6.1, we evaluate the viability of a target RDM network reconfiguration by reusing the network reconfiguration fitness sub-functions defined in equations 6.1-6.5.

Configuration. We configured this experiment as a three-step process where

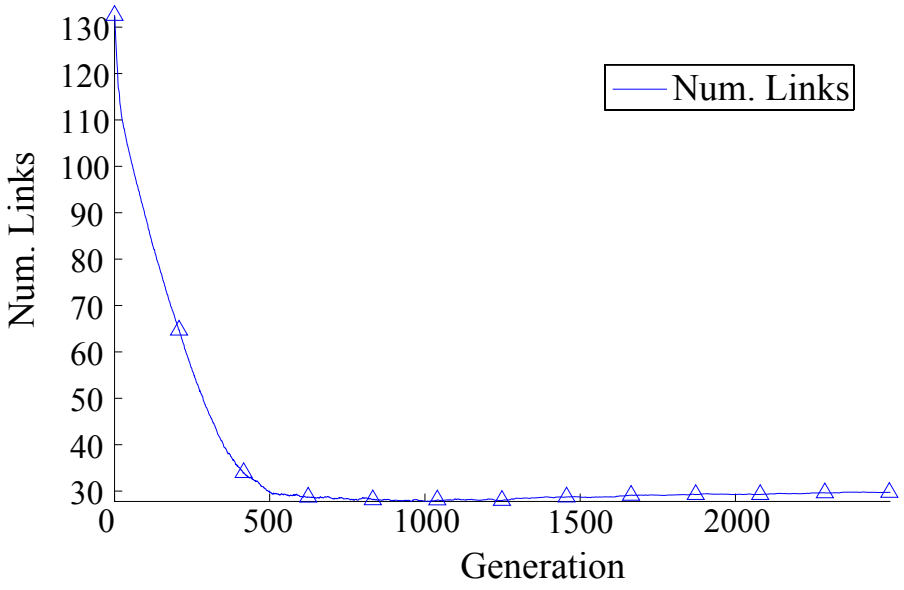


Figure 6.10: Number of active links in overlay network when optimizing for cost, performance, and reliability.

we first ran Plato to produce an initial overlay network design whose primary design objective was to minimize operational costs, i.e., $\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 0$, and $\alpha_{\text{rel}} = 0$. Figure 6.11 presents a representative overlay network evolved by Plato that comprises 24 active links, a spanning tree. Although we could have generated a design to account for both cost and reliability, the objective of this experiment was to force the reconfiguration of the overlay network.

Next, we randomly selected an active network link in the overlay network and set its operational status to *faulty*. Since this link failure disconnected the RDM network and produced an isolated data mirror, the utility functions for Goals (F) and (F') became unsatisfied. Thus, our model-based framework reran Plato to evolve a target system configuration that addressed changes in the underlying network topology.

Results. In response to the network link failure, Plato evolved a new overlay network topology that addressed these environmental changes. Since the initial overlay

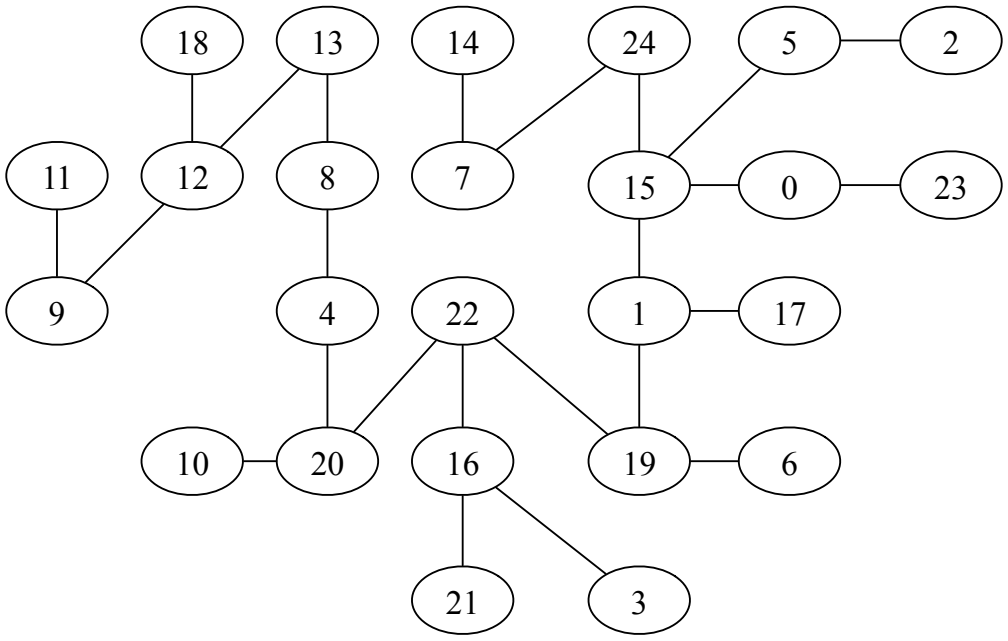


Figure 6.11: Initial overlay network topology with cost being the lone design factor.

network suffered from a link failure, the individual fitness sub-functions were automatically rescaled such that reliability became the primary design concern. Whenever an individual was evaluated, if the encoded overlay network made use of the faulty link, then it was severely penalized by assigning it a low fitness value. Figure 6.12 shows the overlay network evolved in response to the environmental change in the underlying network. This new overlay network, with 6 redundant links, provides more reliability against link failures than the initial overlay network.

Figure 6.13 plots the maximum fitness achieved by Plato as it evolved both the initial and the reconfigured overlay network designs. Specifically, we failed an active link at generation 2500. As a result, the maximum fitness achieved at generation 2501 dropped to negative values. Within roughly 1000 generations (1 min.), Plato had evolved considerably fitter overlay network topologies. Notice the relative difference in maximum fitness achieved by Plato before and after reconfiguration. The initial overlay network optimizes only with respect to operational costs, while the reconfigured overlay network optimizes primarily for reliability, but also optimizes

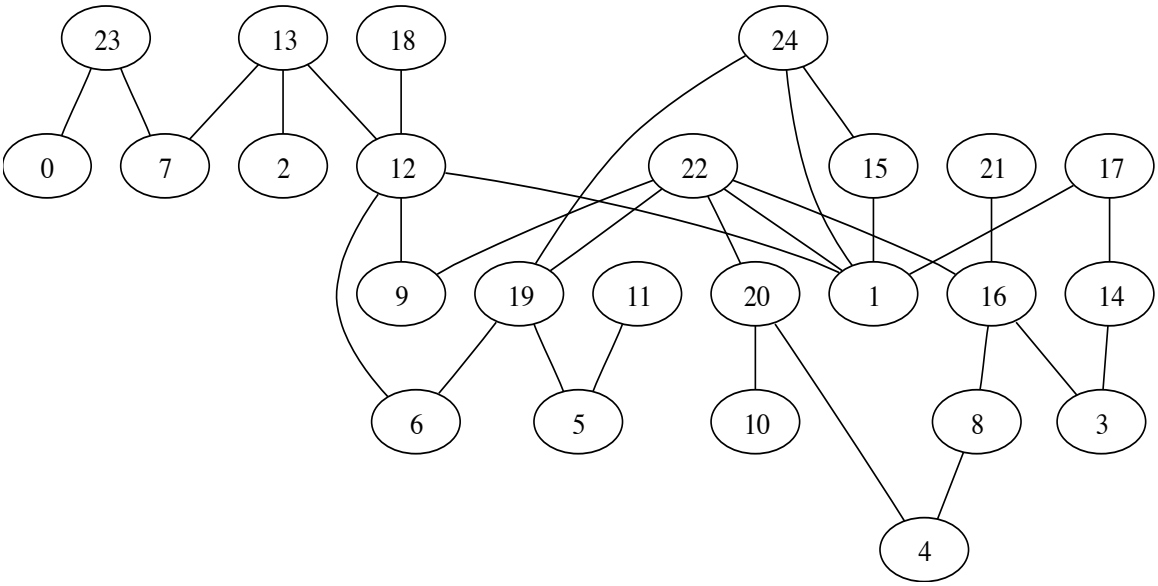


Figure 6.12: Overlay network evolved in response to a link failure.

with respect to operational costs. Since Plato doubled the coefficients for reliability in comparison to cost ($\alpha_{\text{cost}} = 1$, $\alpha_{\text{perf}} = 2$, and $\alpha_{\text{rel}} = 2$), candidate overlay networks after generation 2500 achieved a higher relative fitness value than the initial overlay network.

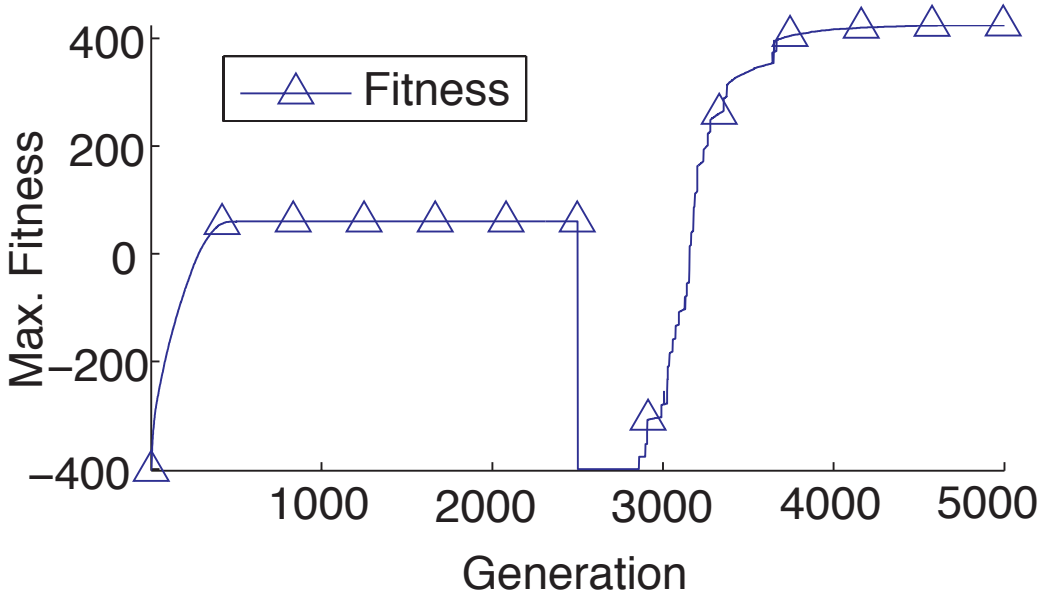


Figure 6.13: Maximum fitness achieved before and after reconfiguration.

Likewise, Figure 6.14 plots the mean number of active links in both the initial and

reconfigured overlay network designs. While the initial overlay design obtains a higher fitness value by reducing the number of active links, the reconfigured overlay design obtains a higher fitness value by adding several active links to improve robustness against future link failures. As this plot illustrates, after the network link failure violates goals (F) and (F'), Plato rebalanced non-functional requirements such that maximizing reliability became more important than minimizing operational costs. As such, after the reconfiguration, the resulting overlay network comprises a greater number of active network links than before the reconfiguration.

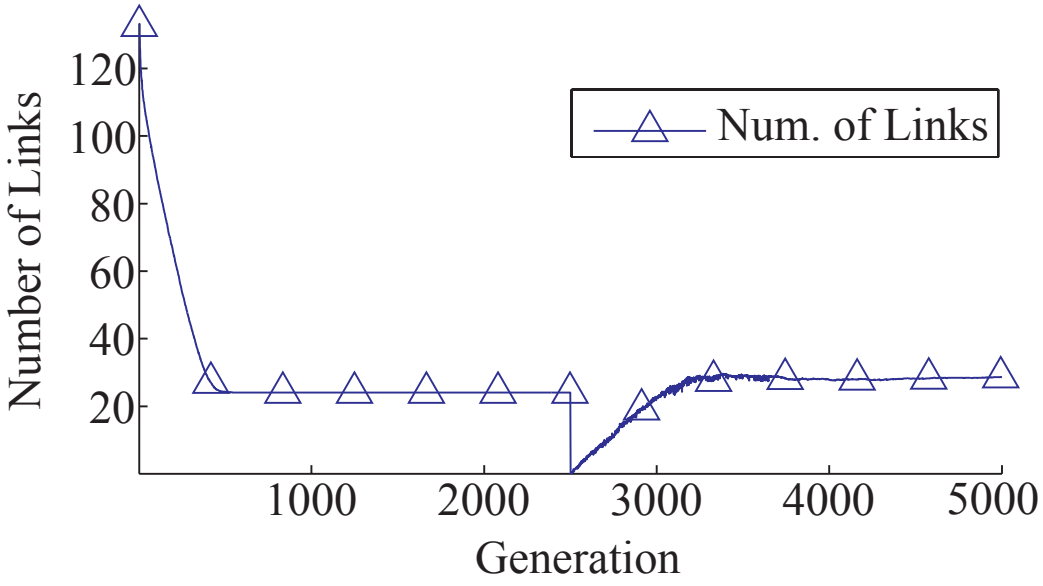


Figure 6.14: Number of active links in overlay network before and after reconfiguration.

Lastly, Figure 6.15 plots the mean potential data loss for a remote data mirror in both the initial and reconfigured overlay networks. The average potential data loss, which is a byproduct of the propagation methods, measures the amount of data, in gigabytes, that may be lost at a remote data mirror as a result of some failure. Lower average potential data loss values imply data is better protected against link failures and vice-versa. As this plot illustrates, after a link failure occurs, the reconfigured overlay network design reset *most* propagation methods to either synchronous or

asynchronous propagation with a 1 or 5 minute time bound, thus improving data protection at the expense of degraded network performance.

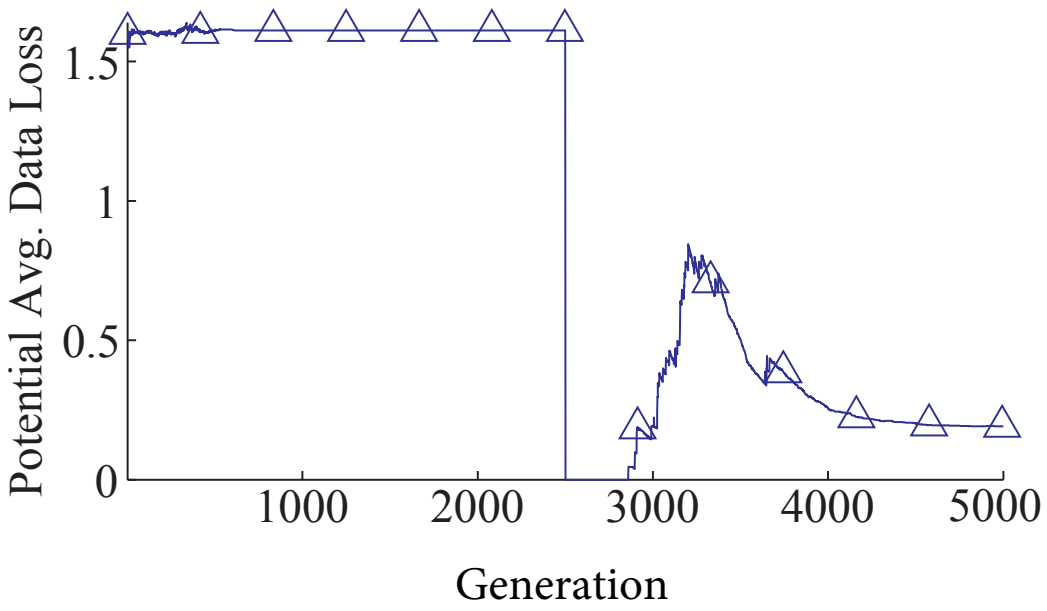


Figure 6.15: Potential average data loss across overlay network before and after reconfiguration.

Collectively, these results show that Plato was able to find viable target RDM network reconfigurations in response to the network link failure. As such, we reject our null hypothesis, H_0 (t-test, $p < 0.05$). Based on these results, we also accept our alternate H_1 hypothesis. Specifically, these results show how Plato automatically rebalanced concerns in response to adverse system and environmental conditions, thereby producing an RDM network topology that improves reliability. These results also confirm that Plato can evolve these target reconfigurations within a minute or less, which is well within the operational limits for an application domain such as RDM.

6.4.1 Reconfiguration of Monitoring Infrastructure

Experimental Objective. This experiment, Experiment 6.4, evaluates the feasibility of using Plato to reconfigure the monitoring infrastructure of the RDM network. This experiment compares the tradeoffs between monitoring costs and monitoring accuracy between Plato and both a static monitoring configuration that is never reconfigured at run time, and an adaptive sampling approach that rescales monitoring intensity in response to changing environmental conditions.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that *monitoring reconfigurations evolved by* Plato *will detect requirements violations while incurring monitoring costs that are not different from static monitoring or adaptive sampling approaches.* We also defined an alternate hypothesis, H_1, that states that *monitoring reconfigurations evolved by* Plato *will detect requirements violations while incurring fewer monitoring costs when compared with static monitoring and adaptive sampling approaches.* Here, monitoring costs are measured in terms of the number of times the RDM network pulls data from a sensor.

Configuration. This experiment comprises three different configurations. First, we configured the static monitoring approach to gather data from each sensor in the RDM network at each time step. Next, we configured an adaptive sampling approach to initially gather data from all RDM network sensors every 4 time steps. If monitoring data changes by more than 15%, then the adaptive sampling approach doubles the frequency for each sensor, with a maximum frequency equivalent to gathering dat at every time step. In contrast, if monitoring data does not change, then the adaptive sampling approach halves the frequency for each RDM network sensor, with a minimum frequency equivalent to gathering data every 8 time steps. Lastly, we configured Plato to balance competing concerns between minimizing monitoring costs and maximizing monitoring accuracy (i.e., $\alpha_e = 0.5$ and $\alpha_a = 0.5$). In this manner, Plato executes at each time step and, if necessary, reconfigures the monitoring in-

frastructure by specifying the rate at which individual RDM network sensors should gather data in response to changes in the execution environment.

For this experiment, we evaluate the three different monitoring strategies by subjecting the RDM network to the same operational context. Specifically, this experiment reuses an operational context that Loki discovered in Experiment 4.1 that introduces most adverse system and environmental conditions during the middle to late data replication and distribution phases. We performed 30 replicate trials for statistical significance and we plot the mean values obtained from these simulations.

Results. Figure 6.16 plots the mean fitness progression achieved by Plato when generating target monitoring configurations at run time. This plot illustrates how Plato initially achieved fitness values that were around 50% of the maximum possible fitness value; here, both fitness sub-functions were diametrically opposed such that randomly generated configurations often balanced concerns approximately equal. From that initial configuration, Plato searched for new monitoring configurations to minimize monitoring costs while maximizing monitoring accuracy. By the last generation, Plato found monitoring configurations that achieved a mean fitness value that was approximately 72% of the maximum possible fitness value. In general, fitter monitoring configurations probed RDM network sensors whose values changed more often, thereby conserving resources by reducing probing frequencies on relatively static sensors.

Figure 6.17 plots the cumulative monitoring costs incurred by each of the three different monitoring approaches as the RDM network replicated and distributed data messages. As this figure illustrates, the static monitoring approach incurred the highest monitoring costs by constantly gathering data from all RDM network sensors regardless of whether system and environmental conditions change. Given a total of 25 `RDM Sensors` and 300 `Link Sensors`, static monitoring probed sensors a total number of $48,750.0$ times.

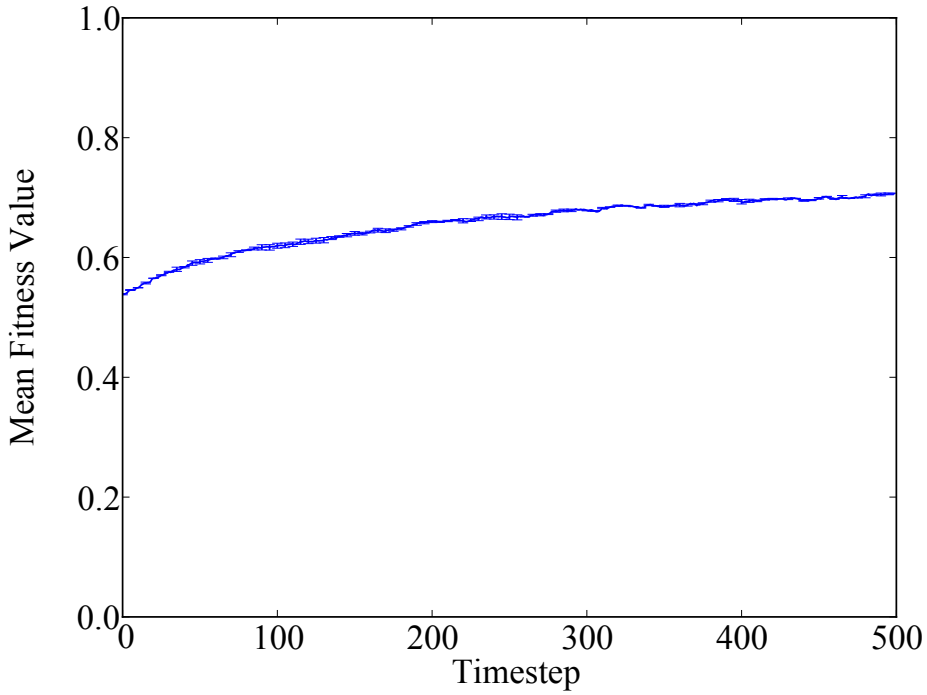


Figure 6.16: Mean fitness progression of Plato when evolving monitoring configurations.

As Figure 6.17 shows, adaptive sampling probed RDM sensors a cumulative number of 30,199.36 times, and Plato probed RDM sensors a cumulative number of 23,155.28 times. Compared with static monitoring, adaptive sampling and Plato reduced monitoring costs by approximately 38.05% and 52.52%, respectively. These results confirm that both adaptive sampling and Plato are able to significantly reduce monitoring costs by reducing the frequency at which the RDM network sensors are probed based on changes in the execution environment.

Although both adaptive sampling and Plato were able to reduce monitoring costs, Plato was able to further reduce monitoring costs by fine-tuning the frequency at which individual RDM network sensors were probed. In particular, adaptive sampling rescales monitoring frequencies uniformly throughout the RDM network. As such, a significant change in one part of the RDM network, such as an individual network link failure, may cause the entire RDM network to increase the rate at which it collects

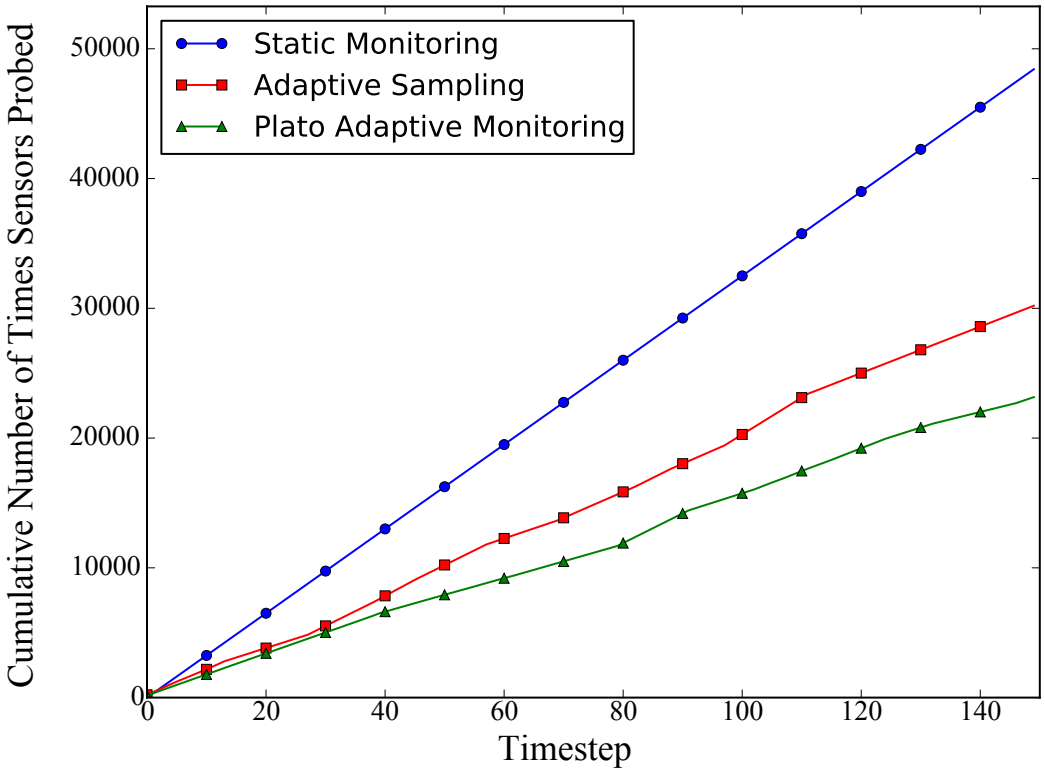


Figure 6.17: Comparison of monitoring costs between static configuration, adaptive sampling, and Plato techniques.

data from all its sensors. Plato, on the other hand, was able to further isolate changes in the execution environment such that it increases the rate at which affected sensors are probed while conserving resources in other areas of the RDM network that are not changing.

Reconfiguration Against Complete Network Failure

Experimental Objective. This experiment, Experiment 6.5, evaluates the operational limits of applying Plato to dynamically reconfigure an overlay network in real-time in response to a degenerate number of link failures. While Experiment 6.2 previously assessed whether Plato could evolve suitable target reconfigurations in response to a network link failure, this experiment seeks to assess whether Plato can

evolve target reconfigurations in response to a continuously degrading network environment.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that Plato *will not be able to repeatedly evolve target RDM network reconfigurations in a continuously degrading network environment.* We also defined an alternate hypothesis, H_1, that states that Plato *will repeatedly evolve target RDM network reconfigurations in a continuously degrading network environment.* Here, we evaluate the feasibility of Plato to evolve target RDM network reconfigurations by using the network reconfiguration fitness sub-functions in equations 6.1-6.5.

Configuration. Initially, Plato produced an initial overlay network design that maximized data reliability while balancing performance and operational costs, i.e., $\alpha_{cost} = 1$, $\alpha_{perf} = 1$, and $\alpha_{rel} = 3$. Next, we selected 5 active overlay network links and set their operational status to *faulty* such that either Goal (F) or (F') became unsatisfied. Our model-based framework then restarted Plato in response to these unsatisfied goals to generate target reconfigurations that addressed specific changes in the environment. We repeated this process every $2,500$ generations, which is the equivalent to one full Plato iteration, for a total of 60 iterations. By the end of the experiment all network links in the underlying network topology were set to faulty status.

Results. Figure 6.18 plots the mean fitness achieved by Plato as it evolved target reconfigurations in response to repeated link failures. This plot illustrates the resilience of evolved Plato reconfigurations even as the entire overlay network suffered major failures. For example, this plot shows how Plato was able to rapidly evolve suitable target reconfigurations, within 30 seconds, of each link failure. In addition, note how the mean fitness value of the generated target overlay networks remains relatively stable at around a value of 400 during the majority of the experiment. This slow decay in fitness values implies that Plato was able to generate solutions at a

consistent level of quality even though network conditions were severely deteriorating. A sharp decay in fitness values is evident after 80% of the overlay network links have failed, which occurs after approximately 125,000 generations. Shortly after 90% of links have failed, fitness values plummet to a value of -400 as there are not enough *non-faulty* links available for Plato to maintain connectivity across data mirrors.

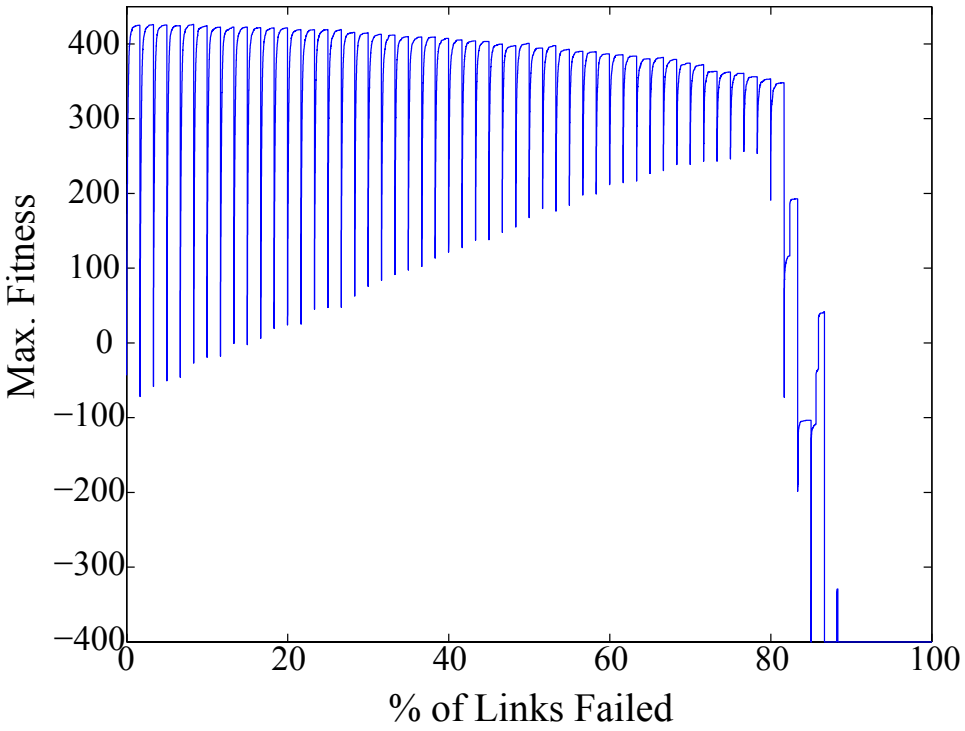


Figure 6.18: Mean fitness of overlay networks achieved throughout multiple reconfigurations until complete network failure.

The plot in Figure 6.19 shows the mean number of active network links in the evolved target reconfigurations. Initially, Plato reduced the number of active network links to create a spanning tree that minimized operational costs. As network links failed, however, Plato increased the number of redundant active network links to maximize data reliability while minimizing operational costs. As Figure 6.19 illustrates, throughout the majority of this experiment, Plato generated target reconfigurations where overlay networks comprised approximately 30 active network links. Moreover,

note how the starting number of active network links in the target reconfigurations is progressively lower during each successive reconfiguration, most likely because there are fewer *non-faulty* network links available for Plato to activate. This plot also highlights how Plato attempts to re-establish connectivity across the RDM network after slightly more than 90% of the network links have failed. Unfortunately, beyond this point it is impossible for Plato, or any other system, to generate target reconfigurations that satisfy the main functional requirement of maintaining network connectivity.

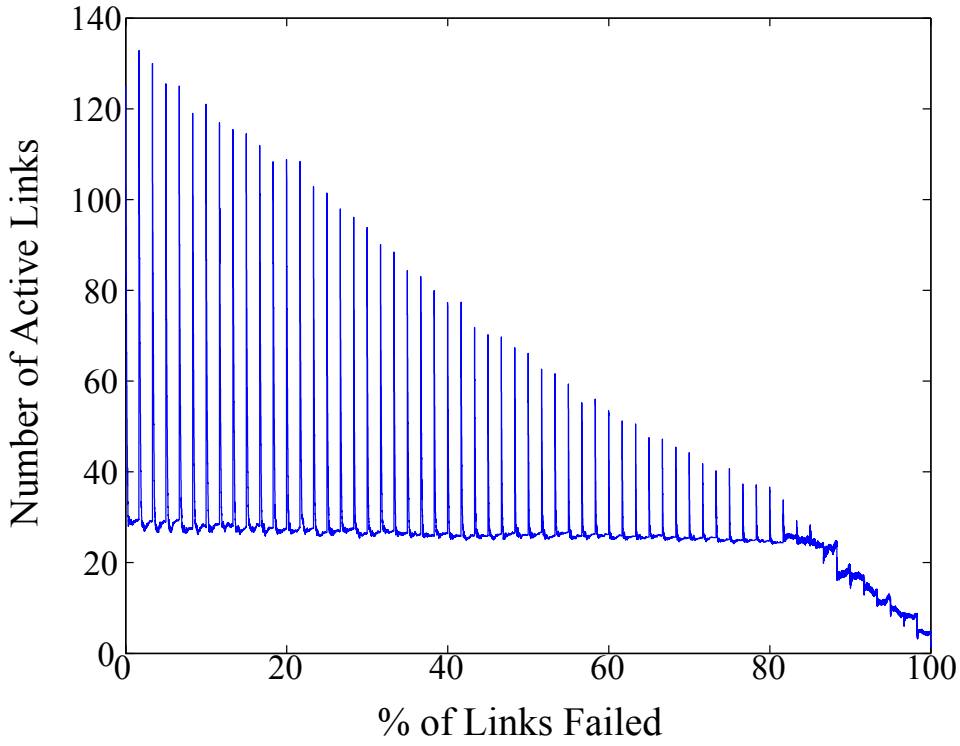


Figure 6.19: Mean number of active links throughout multiple reconfigurations until complete network failure.

Lastly, Figure 6.20 plots the mean potential data loss in the initial and reconfigured overlay network designs until all network links in the fail. The mean potential data loss measures the amount of data, in gigabytes, that could be lost as a result of some failure. This plot demonstrates how Plato repeatedly minimized the mean potential data loss across the RDM network during the majority of the experiment.

Note, however, that the mean potential data loss gradually increased as the number of faulty network links increased, resulting in Plato having fewer usable network links to select from when constructing a target overlay network. Eventually, the mean potential data loss reached its maximum possible value when Plato was unable to build an overlay network that maintained connectivity. At that point data was no longer protected against failures.

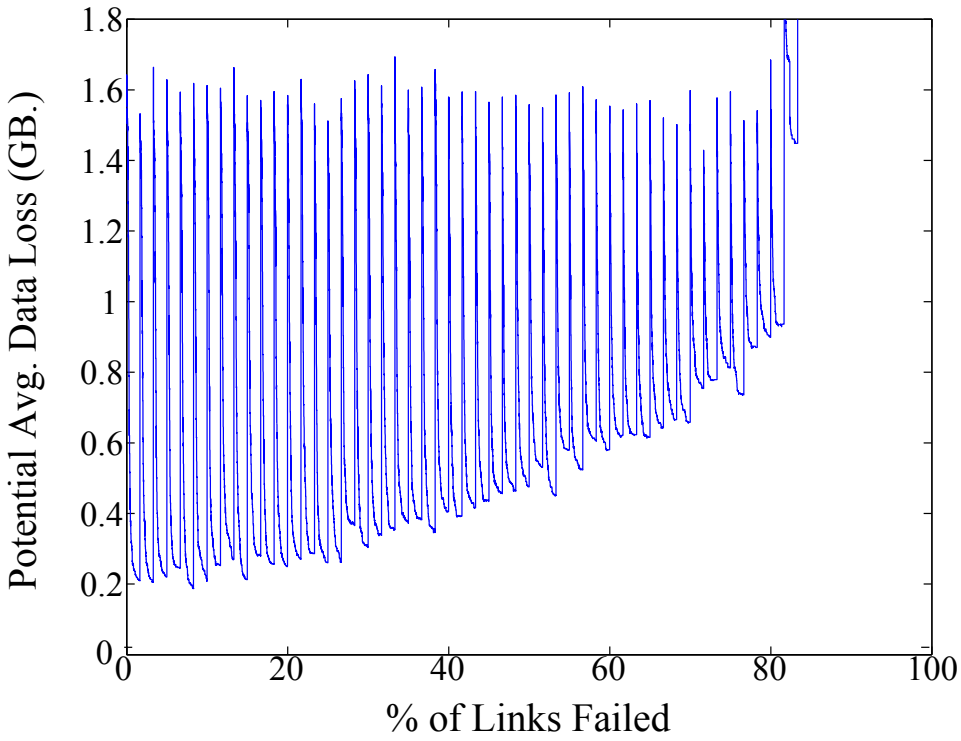


Figure 6.20: Mean potential data loss throughout multiple reconfigurations until complete network failure.

These results confirm that Plato can continuously generate viable target RDM network reconfigurations in response to degenerating network conditions. Based on these results, we reject our null hypothesis, H_0 (t-test, $p < 0.01$). Furthermore, these results enable us to accept our alternate hypothesis, H_1 (t-test, $p < 0.01$). Collectively, these results show how Plato was able to automatically generate target RDM networks throughout each consecutive network link failure until it was impossible for

Plato, or any other reconfiguration engine, to reconnect the RDM network.

6.5 Discussion

These experiments demonstrate how our model-based framework leverage evolutionary computation to support the dynamic reconfiguration of adaptive and autonomic computing systems. By using computationally inexpensive fitness functions, Plato was able to evaluate many candidate reconfiguration in a reasonably short amount of time. In terms of execution time, Plato typically terminated within 3 minutes or less, and typically converged upon a solution within one minute. In this amount of time, Plato typically evaluated approximately $100,000$ candidate target reconfigurations. Furthermore, Plato found viable solutions within 30 or less, well within the practical range for applications such as RDM.

Plato provides several advantages over more traditional approaches for decision-making in self-adaptive and autonomic computing systems. Specifically, Plato does not require developers to explicitly encode prescriptive reconfiguration strategies to address particular scenarios that might arise at run time. Instead, Plato exploits user-defined fitness functions to evolve target reconfigurations in response to changing system and environmental conditions. For instance, when an active network link failed, Plato did not explicitly encode how many network links to activate, which network links to activate, nor which propagation methods to select in response. Instead, Plato automatically evolved viable target reconfigurations while simultaneously balancing competing objectives as captured by non-functional goals. This approach enables Plato to automatically handle a richer set of reconfiguration scenarios than traditional prescriptive approaches.

Genetic algorithms are susceptible to changes in their configuration parameters and solution encodings. For instance, we experimented with various mutation rates

during the design phase to determine which value worked best when applying Plato to the RDM application. Although Plato was able to evolve viable target reconfigurations with modest changes in the mutation rate, higher mutation rates typically caused Plato to require additional computational time to converge upon particular solutions as considerable variation was being introduced into the population each generation. Similarly, the encoding used to represent adaptive systems is extremely important. While more compact representations are possible for encoding a network of remote data mirrors and the propagation methods of each network link, tradeoffs must be made between reducing the search space and altering the probability of different configurations emerging. For example, a more compact genome representation in Plato might encode the operational status and propagation method of each overlay network link as a binary string 3 bits long, thus enumerating each of the possibilities from 0 (a link not used) to 7 (an active link with asynchronous propagation with a 24 hour time bound). Even though this encoding reduces the search space, the probability of deactivating a link now drops from $\frac{1}{2}$ to $\frac{1}{8}$, possibly affecting the quality of solutions evolved by Plato. As a result, it is typically best to begin with default configurations [46] and experiment how the evolutionary algorithm behaves with different parameters.

One potential drawback of Plato is that a genetic algorithm is not guaranteed to find neither optimal nor viable solutions [46]. Although this problem did not arise in any of our experimental trials, it is possible for Plato to converge on suboptimal solutions that are not suitable for particular problems and domains. Genetic algorithm-based approaches tend to be most useful when solution landscapes are vast, complex, and non-linear. For this reasoning, Plato should not be used in autonomic systems where globally optimal solutions are required. Rather, Plato should be applied when acceptable solutions are viable. Finally, it may be possible to integrate Plato with traditional decision-making approaches. For instance, Plato could be leveraged

in the background of a traditional decision-making approach in case a reconfiguration strategy is not available for current system conditions, thereby serving as a backup.

6.6 Summary

In this chapter we presented how our model-based framework supports the dynamic reconfiguration of a DAS. Specifically, throughout this chapter we presented Plato, an evolutionary computation-based technique that generates target system reconfigurations. We showed that it is possible to integrate an evolutionary algorithm within the decision-making process of an autonomic system to dynamically evolve adaptations that balanced competing objectives at run time. By leveraging evolutionary algorithms to generate target reconfigurations, our model-based framework does not require a developer to provide prescriptive rules to address specific scenarios that warrant reconfiguration. Instead, Plato incorporates system and environmental monitoring information to evolve target reconfigurations that address these situations, which were not necessarily anticipated at design time.

Experimental results show that Plato resiliently evolved suitable target reconfigurations against severely deteriorating environmental conditions. Moreover, rescaling the relative importance of different non-functional concerns, obtained from the DAS's goal model, enabled Plato to evolve different types of networks in response to changing requirements and environmental conditions. Lastly, while Plato can be applied at design time to explore larger sets of adaptations, it may also be applied at run time if the application domain allows it.

Chapter 7

Generating Safe Adaptation Paths

This chapter presents how our model-based framework supports the automatic generation of safe adaptation paths that transition an executing system towards its target system configuration. First, we motivate the need to automatically generate adaptation paths that not only preserve system consistency before, during, and after adaptation, but also account for current system and environmental conditions. We then introduce Hermes, the component in our model-based framework that generates safe adaptation paths, and describe how to apply it within the decision-making process of a DAS. We then present experimental results obtained by applying Hermes to the dynamic reconfiguration of an RDM network. Lastly, we discuss Hermes and summarize main findings.

7.1 Motivation

While Plato supports the automatic generation of target system reconfigurations for a DAS, it does not specify *how* to safely reach that target configuration from its current configuration. As Figure 7.1 illustrates, once a DAS determines its target system reconfiguration, it must either select or generate an *adaptation path* to transfer the executing system to that target system configuration. An adaptation path

comprises a series of reconfiguration steps that modify the structure and behavior of a DAS. For instance, the reconfiguration steps that comprise the adaptation path in Figure 7.1 specify that data mirror 2 must be passivated before it may be removed from the RDM network. To prevent the loss of state or introduction of erroneous results during a reconfiguration, a *safe* adaptation path preserves dependency relationships and ensures component communications are not interrupted [64, 65, 121].

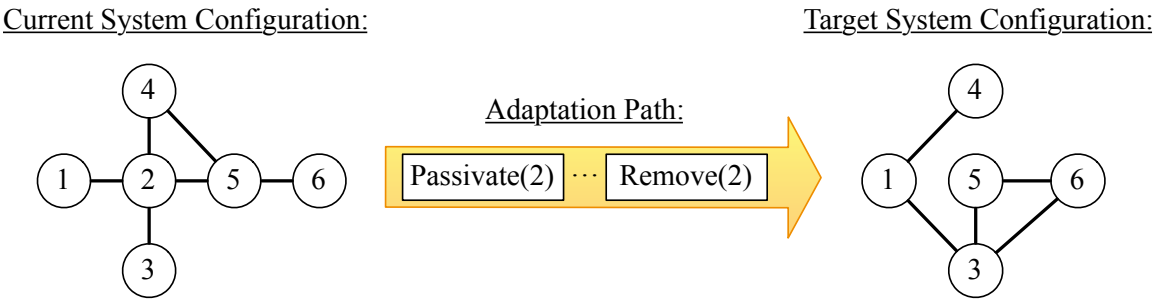


Figure 7.1: Adaptation path overview.

A safe adaptation path preserves system consistency *during* adaptation [64, 108, 120, 121]. To this end, component-dependency analysis approaches [64, 108, 121] first identify sets of system components that are affected by a reconfiguration and must thus be guided to active, passive, and quiescent states in bounded time. An *active* component may initiate and service transaction requests. In contrast, a *passive* component may service transaction requests, but may not initiate requests or be currently engaged in a transaction it initiated. A *queiscent* component, on the other hand, is neither engaged in a transaction nor can it receive or initiate new transactions. A safe adaptation path, therefore, orders and interleaves activate and passivate instructions with reconfiguration operations to eventually transfer the executing system to its desired target configuration without losing or corrupting system data.

Given a starting and a target system configuration, developers can either hand code or automatically generate these adaptation paths. For instance, developers can design and implement adaptation paths at design time to address specific reconfig-

uration scenarios that might arise at run time [15, 35, 53, 120]. Similarly, several automated approaches [108, 121] extend Kramer and Magee's dynamic change management protocol [64] to automatically generate safe adaptation paths that reconfigure distributed computing systems while preserving system consistency during the adaptation. Although multiple safe adaptation paths may exist for a given situation, the identification and selection process is non-trivial, as different paths may represent different tradeoffs between reconfiguration costs, performance, and reliability. Nevertheless, current automated approaches for generating safe adaptation paths consider only one optimization criterion, such as minimizing system disruption, when selecting an adaptation path [108, 121]. As a result, it is desirable to generate adaptation paths that are not only safe, but also balance competing concerns based on current system and environmental conditions.

7.2 Introduction to Hermes

Evolutionary computation can be harnessed to generate safe adaptation paths in response to current system and environmental conditions. Hermes, a component in our model-based framework, supports the automatic generation of safe adaptation paths. The adaptation paths generated by Hermes safely transition an executing system from its current configuration to its desired target configuration, as specified by Plato. Instead of focusing on a single criterion when generating adaptation paths, Hermes evolves solutions that balance competing objectives between functional and non-functional requirements, such as minimizing reconfiguration costs while maximizing reconfiguration performance and reliability.

As with Plato, Hermes can be applied at design time to generate alternative adaptation paths, and applied at run time to generate safe adaptation paths that handle changing system and environmental conditions. Hermes achieves this objective

by gradually transforming and improving an adaptation path by adding, removing, replacing, and reordering reconfiguration instructions in order to better balance competing objectives, while safely reaching the desired target configuration.

7.3 Hermes Process Description

In this section we state assumptions that must hold true when applying Hermes. We then describe each step that a developer applies in order to integrate Hermes into the decision-making process of a DAS.

7.3.1 Assumptions

The following assumptions must hold true for Hermes in order for it to automatically generate safe adaptation paths:

- the set of reconfiguration instructions allows a DAS to reach any possible valid target configuration.
- dependencies between reconfiguration instructions are known and documented.
- components in the DAS can be guided towards active and passive states in bounded time [64].

7.3.2 Safe Adaptation Path Generation Process

To generate safe adaptation paths between starting and target system configuration, Hermes applies genetic programming (see Sections 2.4.2 and 2.4.3) to efficiently evolve safe adaptation paths. In contrast, to many other evolutionary computation-based techniques, genetic programing [63] generates *executable* programs that solve specific and complex tasks, such as regression and robotic control. As such, each program evolved by Hermes comprises executable reconfiguration instructions that

specify structural and behavioral changes that a DAS must perform to safely reach a target system configuration.

Evolved solutions must satisfy two constraints while balancing multiple, potentially competing, factors affecting the system. First, evolved adaptation paths may not reconfigure the executing system to configurations other than those identified for the specified target system. Second, evolved adaptation paths may never cause a DAS to reach an inconsistent or erroneous state. Therefore, if an adaptive system begins a reconfiguration in a consistent state, then it will also reach the target system configuration in a consistent state. We next describe how we applied Hermes to the dynamic reconfiguration of a remote data mirror network with the primary objective of minimizing reconfiguration costs while maximizing reconfiguration performance and reliability.

The following steps describe Hermes's design and implementation for reconfiguring networks of RDMs. Note that Hermes may be applied to other application domains by extending the instruction set with application-specific reconfiguration instructions.

Instruction Set. The core instruction set that Hermes uses to construct safe adaptation paths is derived from Kramer and Magee's dynamic change management approach [64]. These instructions, comprising primitive reconfiguration operators, are overviewed in Table 7.1. An adaptation driver can issue these instructions to control the operational status of system components and reconfigure the application at run time. In addition, each reconfiguration instruction in Hermes is associated with a specific cost that measures the approximate amount of time required for the instruction to complete. For this chapter, cost values reflect approximate and relative estimates for the RDM domain. In practice, these costs may be refined with empirical measurements gathered by the monitoring infrastructure.

Terminal Set. In genetic programming, the terminal set specifies which objects are operands for the instructions. The terminal set in Hermes comprises remote data

Table 7.1: Description of reconfiguration instructions used by Hermes.

Instruction	Description	Cost (s)
Insert Component	Adds component to the network.	10
Remove Component	Removes component from the network.	3
Link Components	Establishes a communication path between the specified components.	3
Unlink Components	Removes a communication path between the specified components.	2
Activate Component	Sets the operational status of a component to active mode.	1
Passivate Component	Sets the operational status of a component to passive mode.	5

mirrors (components), as well as network links (connectors) that can be established between pairs of remote data mirrors. The specific terminal set is derived at run time by analyzing structural differences between starting and target system configurations. Thus, for this application the terminal set only includes remote data mirrors and network links involved in the reconfiguration.

GP Operators. For this application domain, we implemented Hermes as an interpreted linear genetic program [9]. As a result, the genetic program operators include two-point crossover. In terms of mutation operators, Hermes uses insertion, removal, modification, and swap operators. While the insertion operator adds a new instruction into the genome, the removal instruction deletes an instruction from the genome. Similarly, the modification instruction changes parameters for an existing instruction, such as the operand's target component. Lastly, the swap operator randomly exchanges the locations of two instructions in the genome and thus explores the effects of executing reconfiguration instructions in different orders.

Selection. In evolutionary algorithms, selection is the process by which better solutions thrive and sometimes even dominate the population. As with Plato, Hermes also applies *tournament selection*, a variation that randomly selects k individuals from the population and then competes them against each other. The individual with the

highest fitness value *survives* into the next generation where it may undergo further recombination and mutation.

Initialization. Genetic programs must be properly configured for the specific task being solved. Table 7.2 lists several genetic program parameters along with the specific values used in Hermes. Although these values were effective in the remote data mirroring case study, developers should explore various parameters when applying Hermes to other application domains.

Table 7.2: Genetic program configuration.

Parameter	Value
Population Size	1000
Crossover Type	Two-point
Crossover Rate	20%
Mutation Rate	50%
Selection Type	Tournament, $k = 5$
Selection Rate	30%
Max. Generations	1500

In addition to the common genetic program configurations, Hermes requires an additional setup step. While most genetic programs begin execution from a "blank slate" population comprising random individuals, Hermes initializes each individual's encoded program with an initial adaptation path comprising specific instructions *required* to safely transition a system to its target configuration. These instructions are derived through component-dependency analysis [64, 121]. For example, if two remote data mirrors are connected in the target configuration but not in the starting configuration, then it can be deduced that somewhere in the adaptation path, both remote data mirrors must be linked. To preserve these required reconfiguration instructions in the population, no mutation operator in Hermes may remove them. This additional constraint, which is not typical of genetic programming, is needed because the starting and ending points of the evolutionary process are known; traditional genetic programming is more open-ended, where the objective of evolution is

finding interesting endpoints. In contrast, Hermes is looking for interesting paths to get to known endpoints (i.e., target configuration). As a result, Hermes may modify the initial adaptation path in any possible way as long as it safely reconfigures the adaptive system to its target configuration.

It is important to consider the complexity of the solution space comprising all possible adaptation paths. First, we define n to be the number of instructions required to safely transition the system to its target configuration, as determined by component-dependency analysis. Exactly $n!$ possible alternative solutions may be constructed by simply reordering the initial genome. Given that most non-trivial reconfigurations may comprise well over 20 instructions, the solution space comprises over 2.43×10^{18} different adaptation path alternatives, some of which safely transition the system to its desired target configuration, while many others do not. Given the vast number of combinations possible, no current method (manual or automated, heuristic or brute-force exhaustive) is capable of exploring all possibilities in a reasonable amount of time.

Fitness Sub-Functions. A set of fitness sub-functions can be used to evaluate competing objectives, each focusing on a different concern. To this end, Hermes applies a set of fitness sub-functions derived and elaborated from published results in remote data mirroring [50, 55] and search-based software engineering [39, 121] domains.

The first criterion we consider in remote data mirroring is the cost of a reconfiguration. We measure reconfiguration costs as the time required to execute an adaptation path. The following fitness sub-function measures the cost of reconfiguration:

$$F_{cost} = 100 - \left(\frac{time_{ev} - time_{init}}{time_{init}} * 100 \right) \tag{7.1}$$

where $time_{init}$ and $time_{ev}$ measure the amount of time required for the initial and evolved adaptation paths to complete, respectively. This fitness sub-function guides

the selection of individuals whose encoded solution reconfigures the network of remote data mirrors in less time. While Hermes associates reconfiguration costs with the time required to complete a reconfiguration, this time measurement could also be further refined into lost profits due to system disruption.

Another important criterion in remote data mirroring is the performance degradation caused by a reconfiguration. We determine the performance of a reconfiguration by measuring the amount of data produced and diffused by remote data mirrors through the network during reconfiguration. The following two fitness sub-functions measure the performance of an encoded solution:

$$F_{act} = 100 * \left(\frac{components_{act}}{components_{tot}} \right) \tag{7.2}$$

and

$$F_{datasent} = \sum_{i=1}^{components_{tot}} time_{act}(i) * capacity(i) \tag{7.3}$$

where $components_{act}$ is the number of components in active mode during reconfiguration, $components_{tot}$ is the total number of components in the system, $time_{act}(i)$ measures the time a remote data mirror i is actively diffusing data during reconfiguration, and $capacity(i)$ measures the data output produced by a remote data mirror per time unit. The first performance fitness sub-function, F_{act}, measures the percentage of components in the system in active mode throughout the reconfiguration. The second performance fitness sub-function, $F_{datasent}$, measures the amount of data diffused through the network by remote data mirrors during reconfiguration. Together, these two fitness sub-functions guide the genetic program towards solutions that maximize the number of components actively diffusing data through the network during reconfiguration.

The third criterion we consider for remote data mirroring is the reliability, or

potential for data loss, of a reconfiguration. We determine the reliability of a reconfiguration by measuring the amount of data *queued* during reconfiguration. The following two fitness sub-functions measure the reliability of an encoded solution:

$$F_{pass} = 100 * \left(\frac{components_{pass}}{components_{tot}} \right) \tag{7.4}$$

and

$$F_{queued} = \sum_{i=1}^{components_{tot}} time_{pass}(i) * capacity(i) \tag{7.5}$$

where $components_{pass}$ is the number of components in passive mode during reconfiguration, $components_{tot}$ and $capacity(i)$ are the same as defined above, and $time_{pass}$ measures the time a remote data mirror i is in passive mode throughout the reconfiguration process. The first fitness sub-function, F_{pass}, measures the percentage of passivated remote data mirrors in the system. The second fitness sub-function, F_{queued}, measures the amount of data produced by remote data mirrors that is queued because the remote data mirror was in passive mode at the time. Together, these two fitness sub-functions guide the genetic program towards solutions that create large regions of quiescence during reconfiguration. From the perspective of data reliability, establishing large regions of quiescence throughout the system is desirable because it implies data is better protected against failures during reconfiguration.

In Hermes, each set of fitness sub-functions is associated with a vector of coefficients that determines the relative priority of each design concern. By default, coefficients in this vector are all equivalent, implying that no one competing design objective is more significant than others. However, system requirements and environmental conditions may impose different constraints upon the type and quality of the evolved adaptation path. For instance, if the cost of losing data outweighs performance requirements, then the coefficient for reliability should be set to a value larger than that of performance and cost, thus guiding the evolutionary algorithms

towards solutions that provide greater measures of reliability during reconfiguration. This vector of coefficients can also be updated at run time to address changing system and environmental conditions. Moreover, the set of fitness sub-functions and the vector of coefficients can be combined into a single scalar fitness value through a linear weighted sum, as follows:

$$F = \alpha_{cost} * F_{cost} + \alpha_{perf} * (F_{act} + F_{datasent}) + \alpha_{rel} * (F_{pass} + F_{queued}) - penalties \quad (7.6)$$

where *penalties* are reductions in fitness meant to punish individuals whose encoded solution produces undesirable effects or behaviors. For instance, any evolved adaptation path that either fails to transition the system to its desired target configuration, or does so while violating safety constraints, is severely penalized. The objective of penalizing individuals is to guide the evolutionary algorithm towards valid and meaningful solutions by removing individuals from the population with undesirable behaviors that do not promote safe adaptation.

7.4 Case Study

In this chapter we present a set of experiments we conducted to evolve safe adaptation paths that reconfigure a network of RDMs. Each experiment compares the relative fitness value of adaptation paths evolved by Hermes with those derived by component-dependency analysis. Specifically, we compare our results with Kramer and Magee's dynamic change management algorithm [64]. We selected this particular algorithm because it generates safe adaptation paths and is scalable. While the algorithm presented by Zhang *et al.* [121] generates globally optimal solutions that minimize system disruption, the algorithm is not scalable for the input sizes considered in these experiments. Similarly, the tranquility approach introduced by

Vandewoude *et al.* [108] may not be practical for this domain as remote data mirrors frequently propagate large amounts of data and bounded time adaptation is essential.

To compare our results, we implemented Kramer and Magee's dynamic change management protocol [64], which is shown in Figure 7.2. The input for this algorithm comprises a set of components to be inserted (N_c) and removed (N_r), a set of links to be created or removed (LS), and a set of components that must be passivated (CPS). The set of components to be passivated (CPS) comprises all components in N_r, all components with links to any component in N_r, and all components with a link in LS. The algorithm proceeds by passivating all components in CPS, thereby establishing a region of quiescence to preserve system consistency during reconfiguration. The algorithm then removes links from LS present in the system, followed by all components in N_r. Next, components in N_c are inserted, followed by creating all remaining links in LS. Finally, all inserted components, as well as those in CPS that were not removed, are set to active mode, thereby completing the reconfiguration process.

We performed 100 trials for each set of results presented in this subsection and plot the mean value along with corresponding standard error bars. Random starting and target system configurations were generated for each trial. Although the following experiments explore a wide range of possible reconfigurations, including the insertion and removal of several remote data mirrors at run time, each trial focuses mostly on reconfiguring the network topology. This decision was based on the observation that inserting and removing numerous remote data mirrors at run time is generally impractical due to excessive operational costs. Each experiment presented throughout this section was run on a MacBook Pro with a 2.53GHz Intel Core 2 Duo Processor and 4GB of RAM.

Finally, for the following set of experiments, we defined the *null* hypothesis, H_0, to state that *adaptation paths evolved by* Hermes *will show no difference in quality when compared to adaptation paths generated through component-dependency analy-*

```
% Algorithm 1 Reconfigure(N_c, N_r, LS, CPS):
for all i in CPS do
     Passivate i
end for

for all i in LS do
     if LS(i) exists then
          Unlink i
          LS = LS - LS(i)
     end if
end for

for all i in N_r do
     Remove i
end for

for all i in N_c do
     Create i
end for

for all i in LS do
     Link i
end for

for all i in  {CPS - N_r + N_c} do
     Activate i
end for
```

Figure 7.2: Dynamic change management algorithm for reconfiguring dynamic adaptive systems.

sis. Furthermore, we define the alternate hypothesis, H_1, to state that Hermes *will generate solutions better in quality than those produced through component-dependency analysis.* For each of these experiments, the quality of two different adaptation paths is determined by comparing the fitness values associated with each adaptation path.

Base Comparison

Experimental Objective. This experiment, Experiment 7.1, compares the relative quality of adaptation paths evolved by Hermes with those obtained from

component-dependency analysis.

Hypothesis. For this experiment we defined a null hypothesis, H_0, that states that *the quality of adaptation paths evolved by* Hermes *will not be different from the quality of adaptation paths generated by component-dependency analysis.* We also defined an alternate hypothesis, H_1, that states that *the quality of adaptation paths evolved by* Hermes *will be better than the quality of adaptation paths generated by component-dependency analysis.* Here, we measure the quality of an adaptation path by using the collection of fitness sub-functions previously defined in equations 7.1-7.6.

Configuration. In this experiment we consider a typical scenario where the primary objective is to safely reconfigure the network of remote data mirrors while minimizing reconfiguration costs and maximizing reconfiguration performance and reliability, i.e., $\alpha_{cost} = \alpha_{perf} = \alpha_{rel} = 0.333$. In addition, we explore the performance characteristics of Hermes by applying our approach to networks of varying sizes and topologies, where a larger network size typically implies a more complex reconfiguration.

Results. Figure 7.3 shows the average maximum fitness values for adaptation paths in this experiment. In particular, adaptation paths evolved by Hermes achieved greater fitness values than those produced by component-dependency analysis, with a statistical significance of $p < 0.01$ using a t-test. This difference in fitness values implies that our model-based framework can optimize the adaptation paths generated by component-dependency analysis to provide better reconfiguration performance and reliability with a minimal increase in reconfiguration costs. Furthermore, the difference in fitness values gradually increases as the networks and adaptation paths grow in size and complexity (n=15 and n=25). This observation suggests that more opportunities for balancing competing objectives arise as the complexity of a reconfiguration increases. This observation also suggests that Hermes is capable of exploiting such opportunities to improve the overall quality of a safe adaptation path.

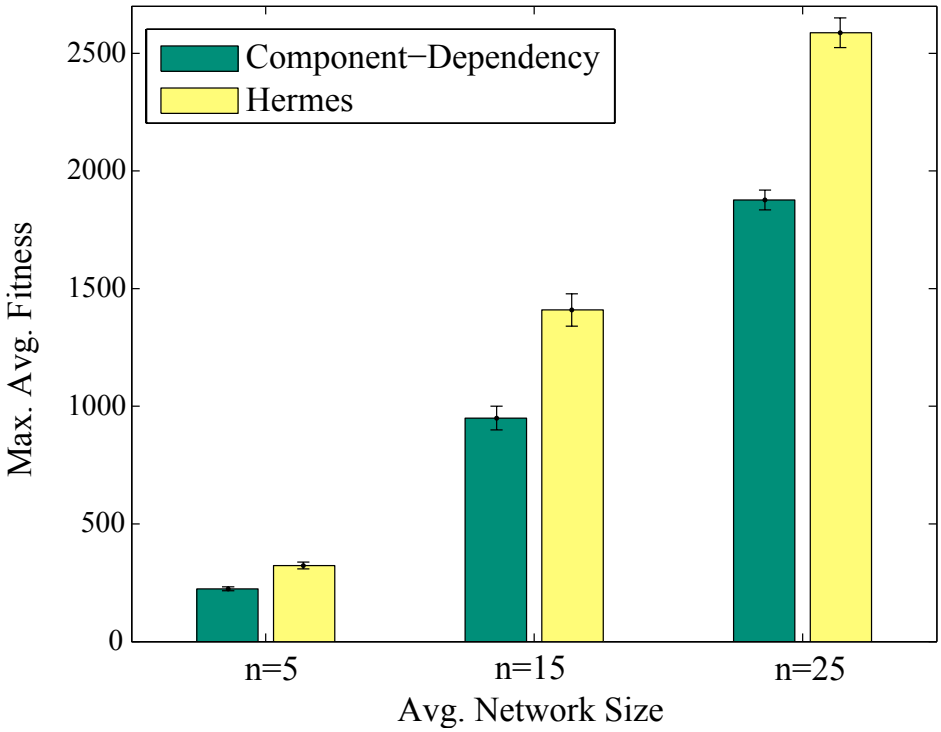


Figure 7.3: Comparison of adaptation path quality.

Figure 7.4 plots the average maximum fitness values of solutions evolved by Hermes for different sized networks per generation. This plot illustrates the rapid rate at which Hermes builds upon and improves the quality of adaptation paths generated by component-dependency analysis, which are represented by the fitness value plotted at the 0 generation before Hermes modifies them (filled icon). Specifically, Hermes achieves large boosts in fitness values within the first 600 generations (< 35 seconds), depending upon the relative size of the network. Thereafter, Hermes continues to fine-tune evolved adaptation paths until the maximum number of generations are exhausted.

Collectively, these results demonstrate how adaptation paths evolved by Hermes achieved greater fitness values than those generated by component-dependency analysis in a variety of different network sizes and topology configurations. As such, these results enable us to reject our null hypothesis, H_0, as well as accept our alternate

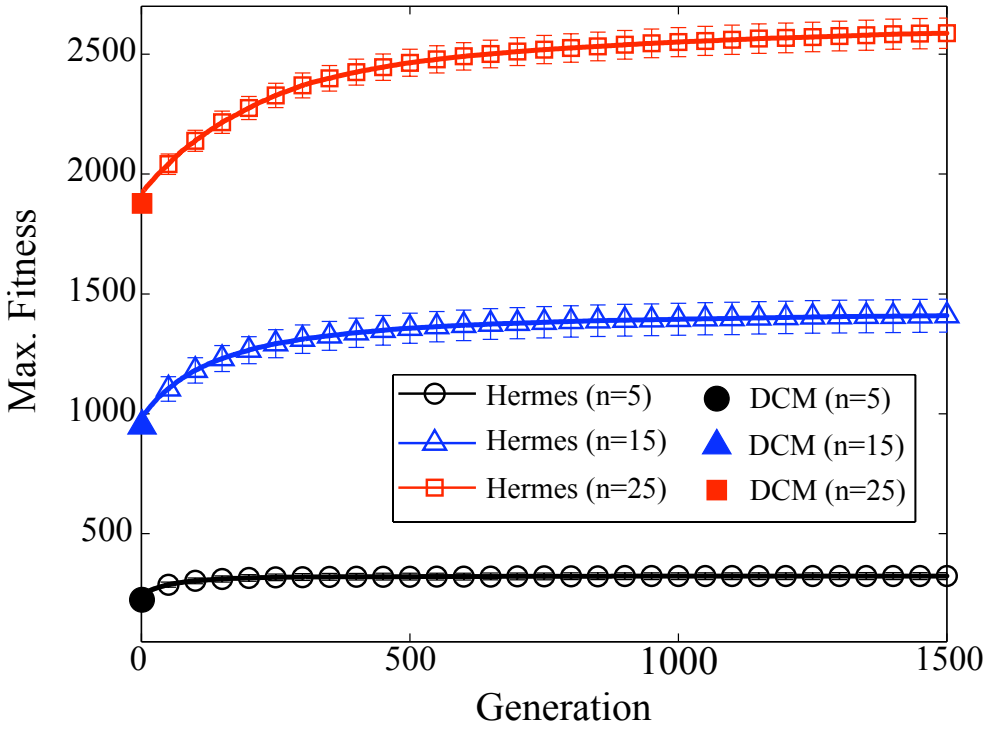


Figure 7.4: Progression of average maximum fitness values for different network sizes.

hypothesis, H_1 (in both cases using a t-test with $p < 0.01$).

Optimizing for Performance and Reconfiguration Costs

Experimental Objective. This experiment, Experiment 7.2, compares the relative quality of solutions evolved by Hermes with those obtained by component-dependency analysis when the main objective is to minimize reconfiguration costs while maximizing reconfiguration performance.

Hypothesis. As with Experiment 7.1, for this experiment we defined a null hypothesis, H_0, that states that *the quality of adaptation paths evolved by* Hermes *will not be different from the quality of adaptation paths generated by component-dependency analysis*. We also defined an alternate hypothesis, H_1, that states that *the quality of adaptation paths evolved by* Hermes *will be better than the quality of adaptation paths generated by component-dependency analysis*. Here, we measure

the quality of an adaptation path by using the collection of fitness sub-functions previously defined in equations 7.1-7.6.

Configuration. In this experiment we configured Hermes to minimize reconfiguration costs while maximizing reconfiguration performance, i.e., $\alpha_{\text{cost}} = 0.4$, $\alpha_{\text{perf}} = 0.4$, and $\alpha_{\text{rel}} = 0.2$. Such trade-off preferences may arise in scenarios where the reconfiguration is driven by variations in system performance rather than by failures that may threaten the functionality of the system. For example, communication paths between remote data mirrors may be reconfigured at run time as environmental conditions such as throughput and loss rate change. For all following experiments, the starting network of remote data mirrors comprises 25 components and at least 35 active network links.

Results. Figure 7.5 plots the average maximum fitness values of adaptation paths evolved by Hermes per generation. Solutions evolved by Hermes achieve an approximate fitness value of 3398. In contrast, adaptation paths generated by component-dependency analysis achieve an approximate fitness value of 1536, which is represented by the filled circle plotted at generation 0 before Hermes modifies it. In general, Hermes evolved solutions that maximized performance without significantly increasing reconfiguration costs and thus improved fitness by 220%. To achieve this objective, Hermes reordered sets of reconfiguration instructions to sequentially reconfigure small subsets of remote data mirrors and connections at any given time, thereby enabling the majority of remote data mirrors to continue propagating data in the meantime. On the average, Hermes increased reconfiguration costs by approximately 12 seconds, less than a 3% increase. As such, the 220% difference in fitness values emphasizes how reordering the initial adaptation path may improve a reconfiguration's performance by reducing system disruption during reconfiguration. Lastly, as this plot illustrates, Hermes achieved large fitness gains within the first 500 generations (< 30 seconds), suggesting that tradeoffs may be balanced in a reasonable

amount of time within the context of remote data mirroring.

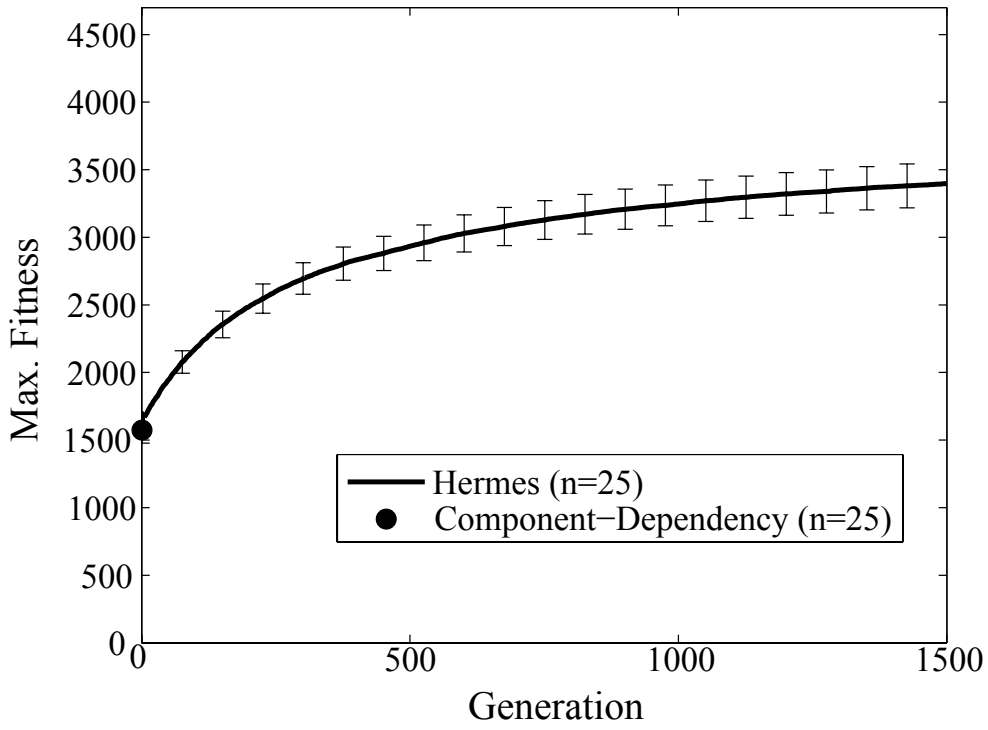


Figure 7.5: Progression of average fitness values when minimizing reconfiguration costs and maximizing reconfiguration performance.

Similarly, Figure 7.6 plots the average amount of data (in MB) sent and queued by remote data mirrors during reconfiguration. This plot is generated by analyzing evolved adaptation paths to determine the time period in which a remote data mirror is either in active or passive mode. As this plot illustrates, Hermes gradually evolves solutions that diffuse larger amounts of data, while queueing less data. These results confirm that adaptation paths are multi-dimensional. Furthermore, these results also suggest the inherent tradeoff between the performance and reliability of a reconfiguration. Specifically, minimizing system disruption enables remote data mirrors to diffuse greater amounts of data, but a single failure during reconfiguration could potentially lose significant amounts of data.

Independently of whether Hermes is generating an adaptation path to minimize

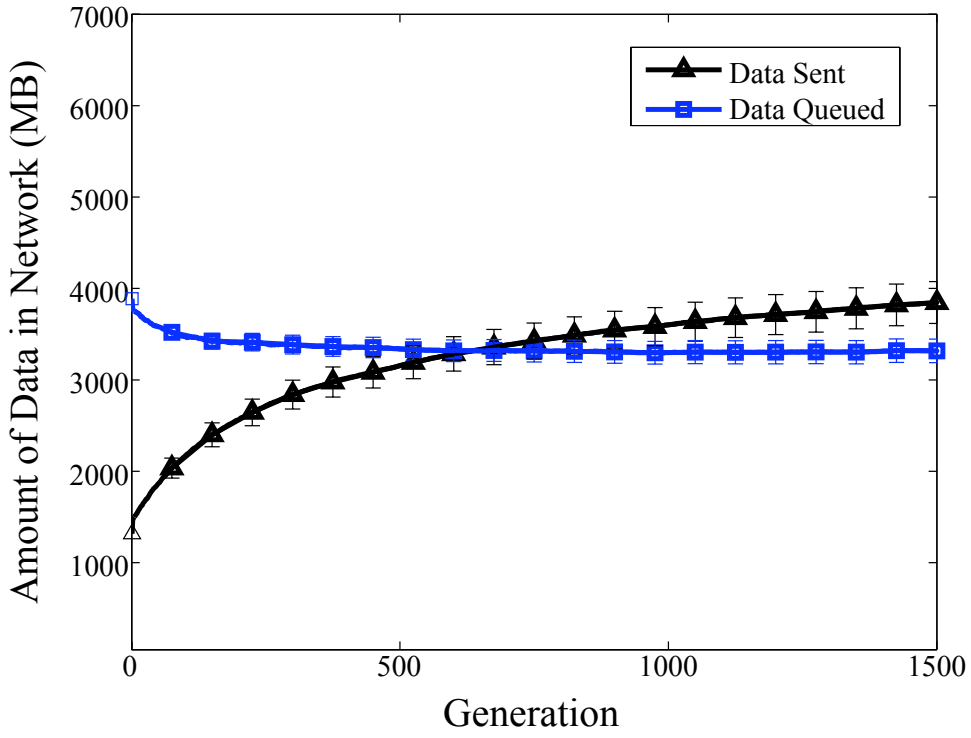


Figure 7.6: Performance and reliability tradeoffs in evolved solutions.

reconfiguration costs or not, it must still place the same number of data mirrors in passive and quiescent modes during the adaptation. Nevertheless, the *sequence* in which those reconfiguration instructions are applied can affect the resulting reconfiguration costs. As Figures 7.5 and 7.6 suggest, Hermes reorganized the initial safe adaptation path in order to group reconfiguration instructions that concurrently affect subsets of data mirrors. Specifically, for an RDM network comprising 25 data mirrors, Hermes placed a mean maximum of 4.3 data mirrors in quiescent mode during a reconfiguration. Likewise, Hermes placed a mean maximum of 7.2 data mirrors in passive mode during a reconfiguration, not counting those already in quiescent mode. This enables the remaining 13.5 data mirrors, over half of the RDM network, to continue diffusing data even as the system is being reconfigured.

These results demonstrate how adaptation paths evolved by Hermes achieved greater fitness values than those generated by component-dependency analysis

when minimizing reconfiguration costs and maximizing reconfiguration performance. Specifically, when compared with the adaptation paths generated by component-dependency analysis, the evolved adaptation paths achieved lower operational costs as well as achieved better reconfiguration performance. Collectively, these results enable us to reject our null hypothesis, H_0, as well as accept our alternate hypothesis, H_1 (in both cases using a t-test with $p < 0.01$).

7.5 Discussion

The set of experiments described in this chapter demonstrate how our model-based framework leverages evolutionary algorithms to support the generation of safe adaptation paths that can be used by adaptive and autonomic computing systems to reach their target system configuration. In terms of execution time, Hermes typically terminated within 2 minutes or less, and typically converged upon a solution within one minute while evaluating 1.5 million candidate solutions. By using computationally inexpensive fitness functions, Hermes was able to find viable solutions within 30 seconds, well within the practical range for applications such as RDM. Moreover, since Hermes always starts with a viable safe adaptation path generated by component-dependency analysis, solutions produced by Hermes can only improve. In this manner, Hermes does not require a predetermined amount of execution time before viable safe adaptation paths are available.

Similarly, experimental results in this chapter confirm that adaptation paths are multi-dimensional in nature and that the specific sequence of reconfiguration instructions produce non-linear effects upon the cost, performance, and reliability of a reconfiguration. Specifically, in each experiment trial Hermes evolved safe adaptation paths that achieved a higher fitness value than those produced by component-dependency analysis, with a significance of $p < 0.01$. As a result, we conclude that

Hermes is capable of not only evolving higher quality adaptation paths when compared to those generated by component-dependency analysis, but also of balancing multidimensional tradeoffs between non-functional requirements, as captured by the non-functional goals in the goal model of the DAS.

7.6 Summary

In this chapter we presented how Hermes supports the generation of safe adaptation paths in a DAS. Hermes is a genetic programming-based technique that generates adaptation paths that not only reach a target system configuration, but also preserve system consistency before, during, and after adaptation. We showed that it is possible to integrate an approach such as Hermes within the decision-making process of a DAS to dynamically evolve adaptations that balance competing objectives at run time. By leveraging evolutionary algorithms to generate safe adaptation paths, our framework does not require a developer to manually design adaptation paths for anticipated scenarios that warrant reconfiguration. Instead, Hermes incorporates system and environmental monitoring information to evolve safe adaptation paths that address situations that were perhaps not considered or anticipated at design time. Experimental results show Hermes evolved suitable adaptation paths against different sets of environmental conditions. Moreover, rescaling the relative importance of different non-functional concerns, obtained from the DAS's goal model, enabled Hermes to evolve different adaptation paths to balance competing concerns. Lastly, we discussed how Hermes can be applied at run time to generate safe adaptations.

Chapter 8

End-to-End RDM Example

Thus far, this dissertation has presented each component in our model-based framework and showed results of their application to the RDM network case study. This chapter presents an example workflow to illustrate how each component in our model-based framework can be applied from beginning to end when specifying, monitoring, and dynamically reconfiguring an RDM network. Specifically, this chapter describes how to use the suite of tools to start with a goal model and progress through refinements to RELAX goals according to identified sources of uncertainty and conclude with the dynamic reconfiguration of the system to mitigate the adverse effects of uncertainty. To this end, this chapter both reuses experimental results previously presented throughout this dissertation, as well as presents additional results based on refinements to the RDM goal model.

Given a goal model of the RDM application, in this chapter we first derive utility functions for requirements monitoring. Next, we anticipate possible sources of system and environmental uncertainty. We then analyze these sources of uncertainty and revise the RDM goal model to resolve adverse system and environmental conditions that lead to the violation of RDM invariant goals. Using the revised RDM goal model, we then regenerate the utility functions for requirements monitoring and show how

the revised goal model improves requirements satisfaction under the same scenarios. We then fine-tune these utility functions to account for sources of minor and transient uncertainty that might cause the RDM network to incorrectly self-reconfigure at run time. Lastly, we show how our model-based framework supports the dynamic reconfiguration of the RDM network.

8.1 Deriving Utility Functions

This section describes how our model-based framework supports the automatic derivation of utility functions for requirements monitoring in a DAS. To this end, we first describe an example KAOS goal model of the RDM application that does not contain any RELAXed goals. Next, we introduce several RELAX operators to address possible sources of system and environmental uncertainty that might prevent the RDM from satisfying its invariants. We then automatically derive invariant, non-invariant, and RELAXed utility functions for requirements monitoring in the RDM application. Lastly, we present simulation results that show how the derived utility functions capture requirements satisfaction in response to different system and environmental conditions.

RDM Goal Model. Athena requires a KAOS goal model of the RDM application in order to automatically derive utility functions for requirements monitoring. Figure 8.1 presents an example KAOS goal model that captures the requirements and constraints that the RDM application must satisfy. As this model illustrates, the primary objective of RDM is to maintain data available at all times by replicating and distributing copies of data to physically isolated RDMs while maintaining operational costs at or below the allocated budget. More specifically, Goals (A) and (B) are invariants that the RDM application must always satisfy. Other non-invariant goals in this model capture various ways for the RDM application to establish and main-

tain network connectivity between RDMs, distribute data messages, and minimize adaptation costs.

RELAXing the RDM Goal Model. The initial RDM goal model does not contain RELAX operators since sources of system and environmental uncertainty have not yet been identified. Nevertheless, at this stage we can anticipate several sources of system and environmental uncertainty that can prevent the RDM network from satisfying its objectives at run time. Figure 8.2 shows the RELAXed KAOS goal model for the RDM application. In general, this revised goal model introduces various RELAX operators to account for imperfect network links that can introduce uncertainty in the form of network link failures and dropped or delayed data messages. For example, a network link failure can disconnect the RDM network and thus violate the satisfaction of Goal (F). As such, we RELAX goal (F) to allow temporary RDM network partitions while the data diffusion process continues. Likewise, lossy network links can drop or delay data messages and thus hinder the satisfaction of Goals (A), (C), (G), (H), (R), and (T). Therefore, we also RELAX goals (C), (G), (H), (Q), (R), (S), and(T) to introduce flexibility in how and when the RDM network propagates data between data mirrors such that data is sufficiently protected from failures, as well as the data diffusion process completes in a reasonable amount of time.

Derived Utility Functions. In general, Athena uses a KAOS goal model of the DAS to automatically generate utility functions for requirements monitoring. As such, we now use Athena to derive utility functions from the RELAXed goal model of the RDM application shown in Figure 8.2. These utility functions are intended to capture how various system and environmental conditions affect the RDM network's ability to satisfy its requirements at run time. Applying Athena to the RELAXed RDM goal model yields a total of 15 utility functions for requirements monitoring. In particular, two state-based utility functions measure the satisfaction of Goals (A) and (B). In addition, thirteen fuzzy logic-based utility functions measure the satisficement

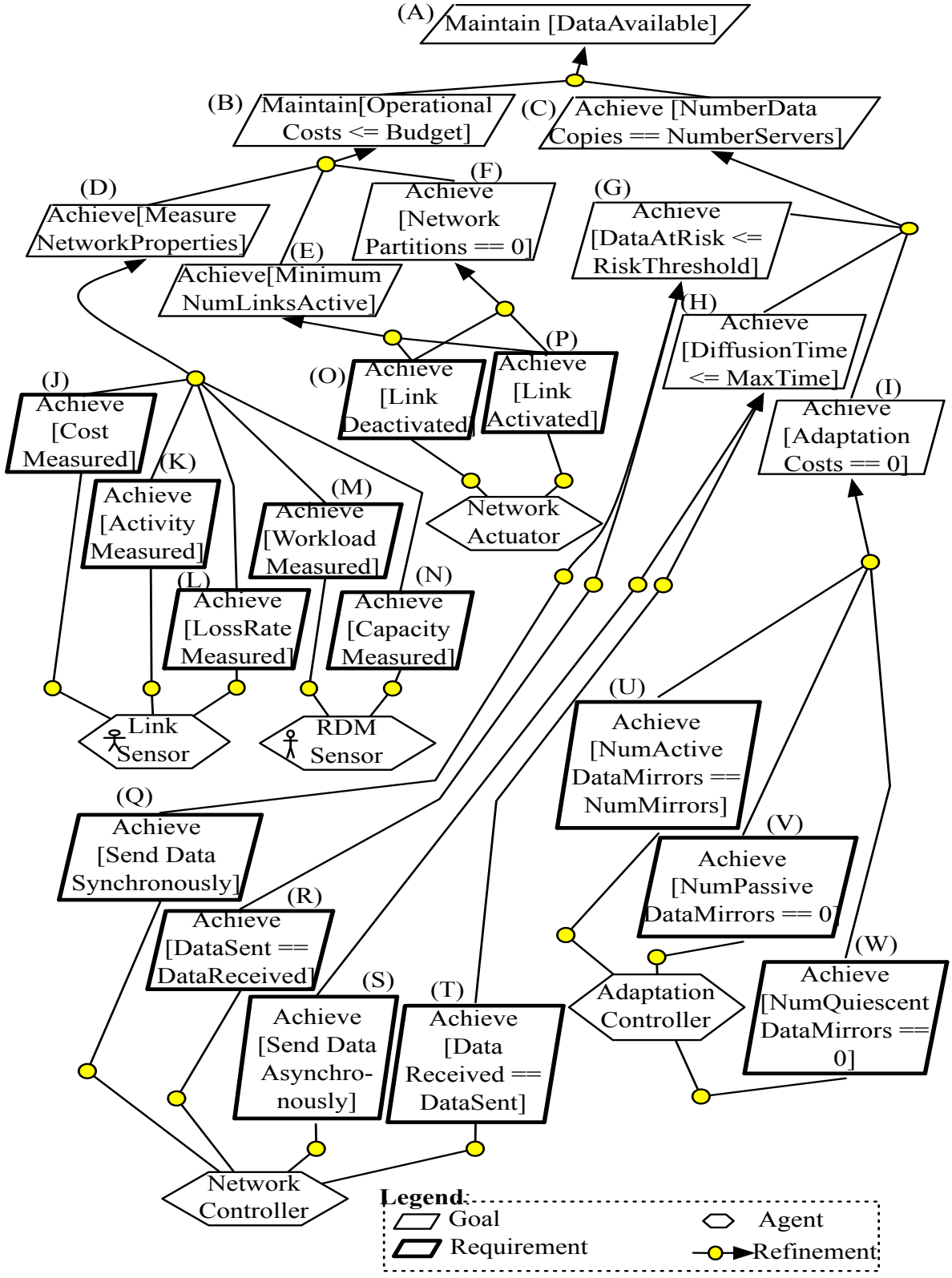


Figure 8.1: KAOS goal model for RDM application.

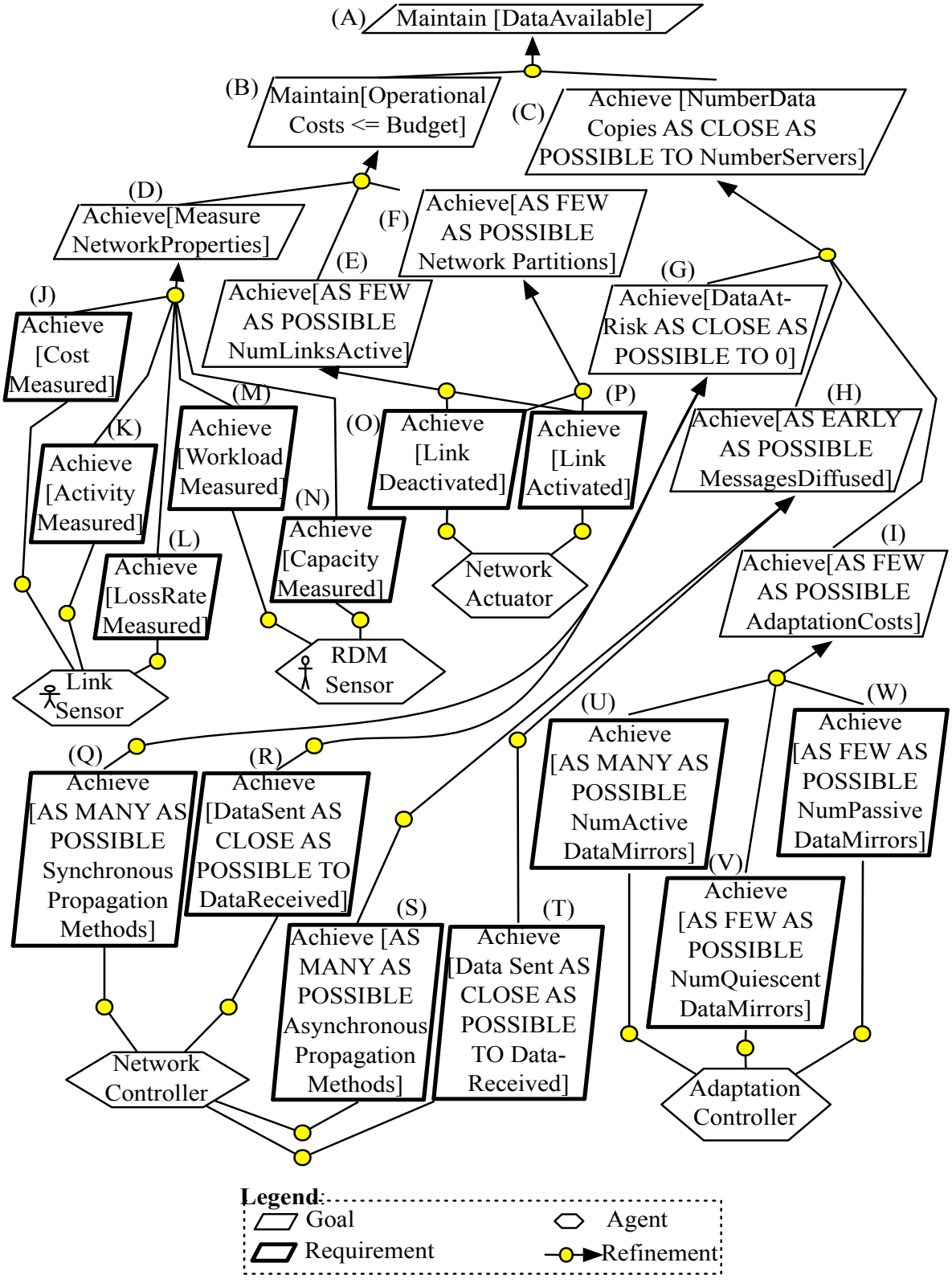


Figure 8.2: RELAXed goal model for RDM application.

of Goals (C), (E), (F) through (I), and (O) through (W).

Next, we use an executable specification of the RDM application to show how derived utility functions capture how system and environmental conditions affect the ability of the RDM network to satisfy its requirements. As Table 8.1 shows, we configured the RDM network simulation such that it randomly introduces adverse system and environmental conditions such as sensor noise, network link failures, and dropped, delayed, and corrupted data messages. These adverse system and environmental conditions may cause the RDM network to self-reconfigure at run time if goals become unsatisfied.

Table 8.1: Configuration for simulation with uncertainty.

Property	Value
Seed	1...30
Distribution	Binomial|Exponential|Normal| Poisson|Uniform
Number Data Mirrors	25
Underlying Network Topology	Complete
Budget	$500000.00
Base Data Mirror Capacity	6.0 Gb
Data Mirror Capacity Variance	0.25
Base Network Link Bandwidth	7.0 Gb per time step
Network Link Bandwidth Variance	0.25
Base Data Message Size	2.0 Gb
Data Message Size Variance	0.25
Prob. Data Mirror Failure	0.01
Prob. Network Link Failure	0.1
Prob. Data Message Drop	0.1
Prob. Data Message Delayed	0.05
Prob. Data Message Corrupted	0.05
Prob. Data Mirror Sensor Failure	0.05
Prob. Sensor Fuzz	0.05

The RDM network is unable to always satisfy its invariant requirements under these system and environmental conditions. Figure 8.3 shows that the RDM network only satisfies Invariant Goal (A) in 19 out of 30 different simulation trials. Furthermore, the corresponding utility value dips plotted in Figure 8.3 suggest that

Invariant Goal (A) becomes unsatisfied at different times in the simulation for various reasons. For instance, in one scenario, Goal (A) becomes unsatisfied at the beginning of the simulation when a newly inserted data message becomes corrupted. In another scenario, Goal (A) becomes unsatisfied near the end of the simulation when the congested RDM network was unable to completely diffuse data items before the simulation completed.

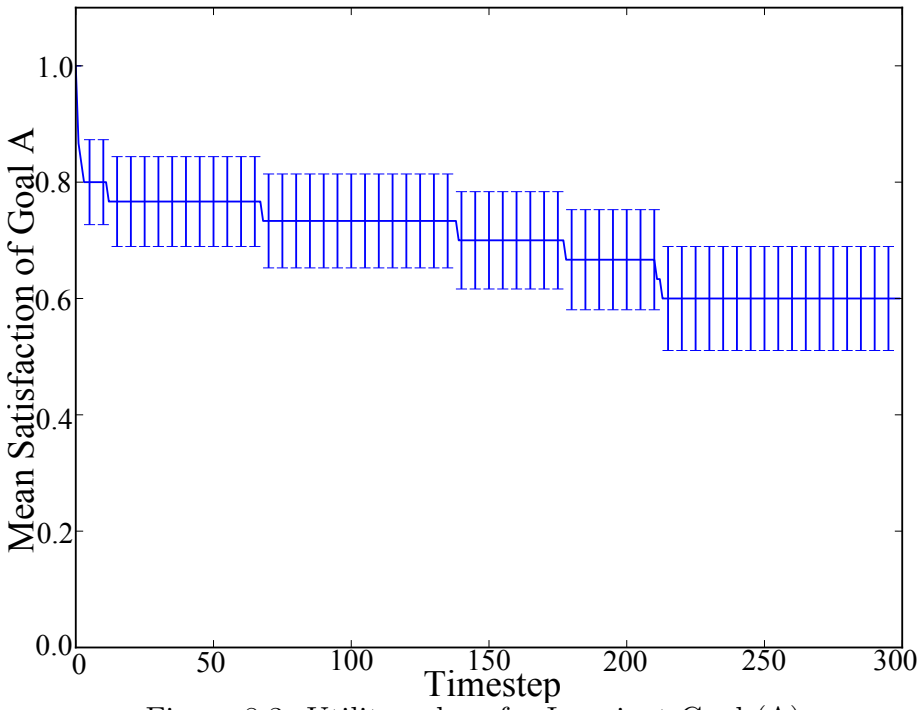


Figure 8.3: Utility values for Invariant Goal (A)

Figure 8.4 plots the satisfaction of Invariant Goal (B) across all 30 simulation trials. As this plot illustrates, in addition to violating Invariant Goal (A), the RDM network also violated Invariant Goal (B) in at least one simulation trial. In this particular scenario, the constructed RDM network exceeded the operational budget thereby violating Goal (B). Note that the RDM network satisfies Goal (B) in all other 29 simulation trials.

Figure 8.5 plots mean utility values for Goal (C) that states that the RDM network should store data replicates in as many data mirrors as possible. As this figure

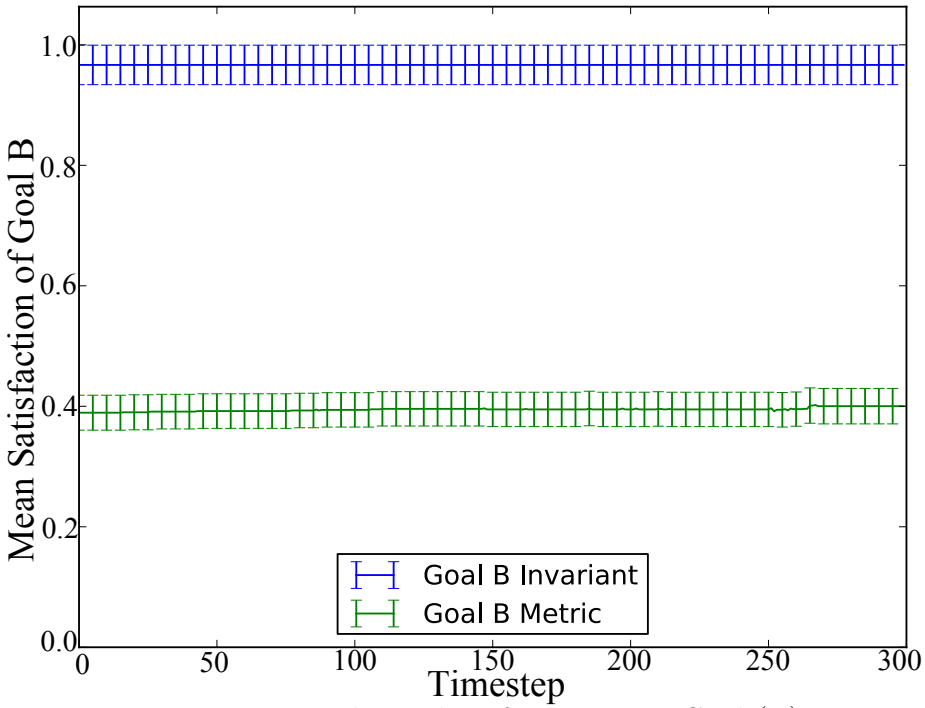


Figure 8.4: Utility values for Invariant Goal (B)

shows, the RDM network gradually satisfices Goal (C) by replicating and distributing data messages around time step 12. Thereafter the RDM continues replicating and distributing data messages in order to gradually increase the satisficement of Goal (C). Note, however, that the RDM network was unable to fully replicate and distribute all data messages by the end of the simulation, as shown by the final satisficement value of Goal (C) that is less than 1.0.

Figure 8.6 provides additional insight into how the RDM satisficed Goal (C) by plotting the mean ratio of total data messages replicated and distributed across the RDM network at each simulation time step. As this figure shows, the mean ratio of diffused data messages and the satisficement of Goal (C) are positively correlated; that is, as the mean ratio of diffused data messages increases, so does the satisficement of Goal (C). Moreover, this figure also captures how the RDM network is unable to diffuse all data messages by the end of the simulation due to adverse environmental conditions, such as repeatedly failed network links and dropped data messages.

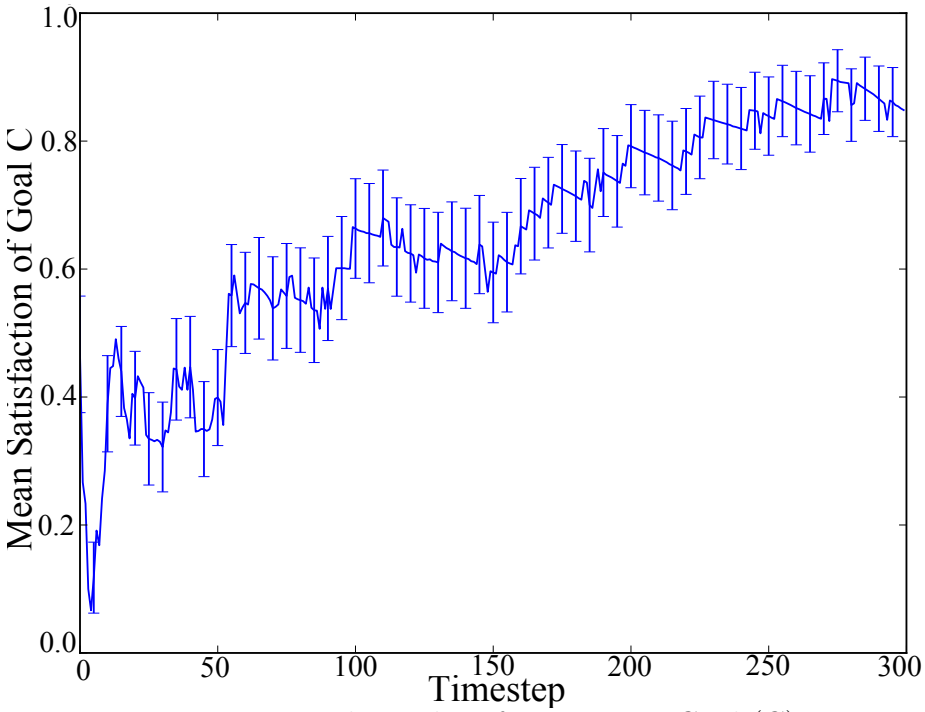


Figure 8.5: Utility values for Invariant Goal (C).

Combined, Figures 8.5 and 8.6 also support the utility values captured in Figures 8.3 and 8.4 since they show that not all data messages are protected against data mirror failures by the end of the simulation.

Given the possibility of network link failures, it is quite likely for the RDM network to become partitioned at run time. Figure 8.7 plots the mean utility values for Goal (F) that states that the RDM network should minimize the number of network partitions in order to distribute data messages to all RDMs in the network. As this plot shows, mean utility values for Goal (F) were within the inclusive ranges of 0.84 and 1.0. These range of utility values shows that at no point in any simulation did the RDM network suffer from more than one concurrent network partition.

Figure 8.8 provides additional insights regarding the satisficement of Goal (F) by showing a sample RDM network topology. As this figure shows, the RDM network became partitioned after the network link connecting data mirrors 16 and 24 failed. Although this network link failure partitioned the RDM network, it still allows data

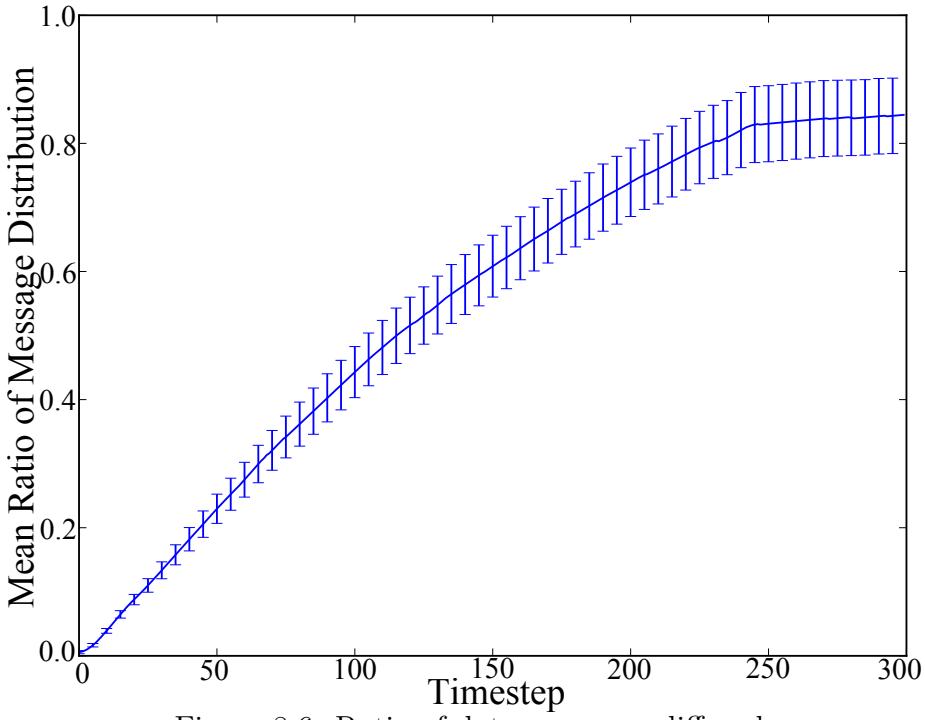


Figure 8.6: Ratio of data messages diffused.

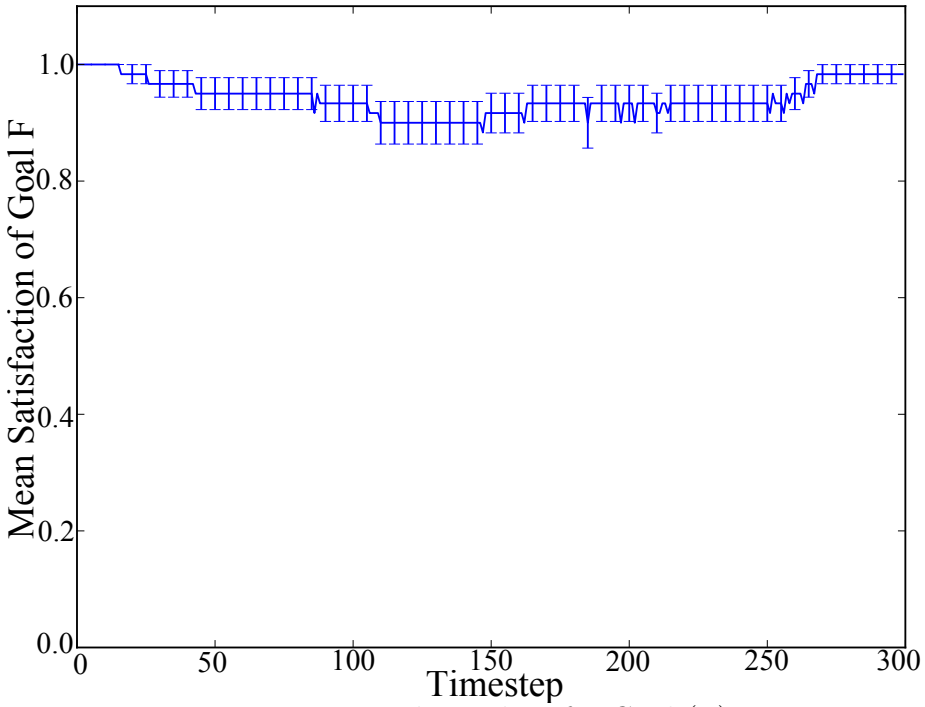


Figure 8.7: Utility values for Goal (F).

mirrors {0, 3, 4, 5, 8, 15, 16, 18, 19} and {1, 2, 6, 7, 9, 10, 11, 12, 13, 14, 17, 20, 21, 22, 23, 24} to diffuse data messages amongst themselves. Eventually, the RDM network is reconfigured and the data diffusion process continues.

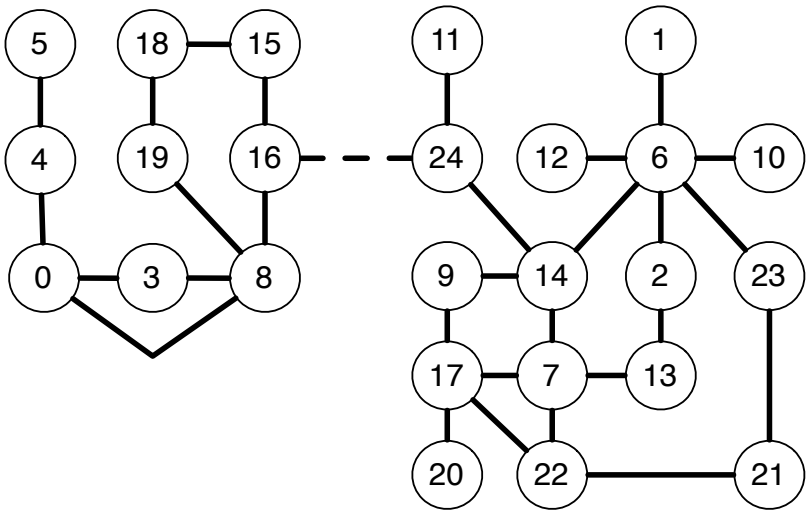


Figure 8.8: Partitioned RDM network.

Similarly, Figure 8.9 plots the mean number of active network links in the RDM network throughout each time step. As this plot shows, the RDM network activated approximately 28 to 29 network links. As such, four to five of these network links were redundant and improved data diffusion performance and reliability. Specifically, this redundancy protects the RDM network even when multiple concurrent link failures occurred such that the network did not end up with more than one partition.

Lastly, Figure 8.10 plots utility values for Goal (I) in Figure 8.2 that specifies that the RDM network should minimize adaptation costs. As this figure illustrates, utility values for Goal (I) were within the ranges of 0.86 and 1.0, thus implying that the RDM network is, for the most part, able to continue replicating and distributing data messages even while adaptations are performed.

Figure 8.11 plots the mean cumulative number of adaptations triggered throughout each simulation. This plot depicts how the RDM network self-reconfigured anywhere from 2 to 15 times per simulation in order to re-establish connectivity and

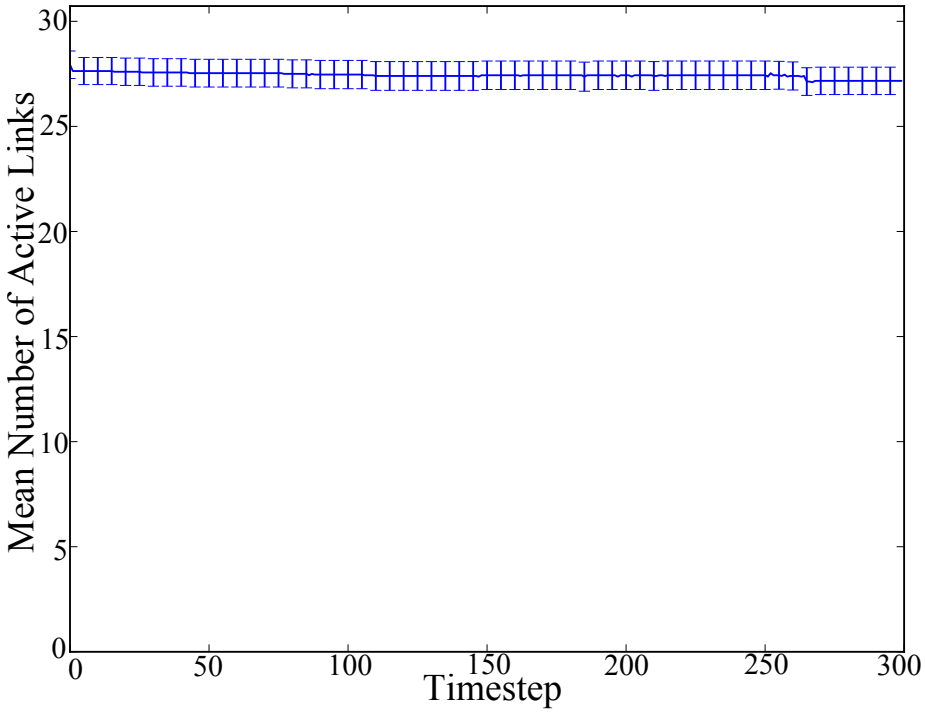


Figure 8.9: Number of active network links.

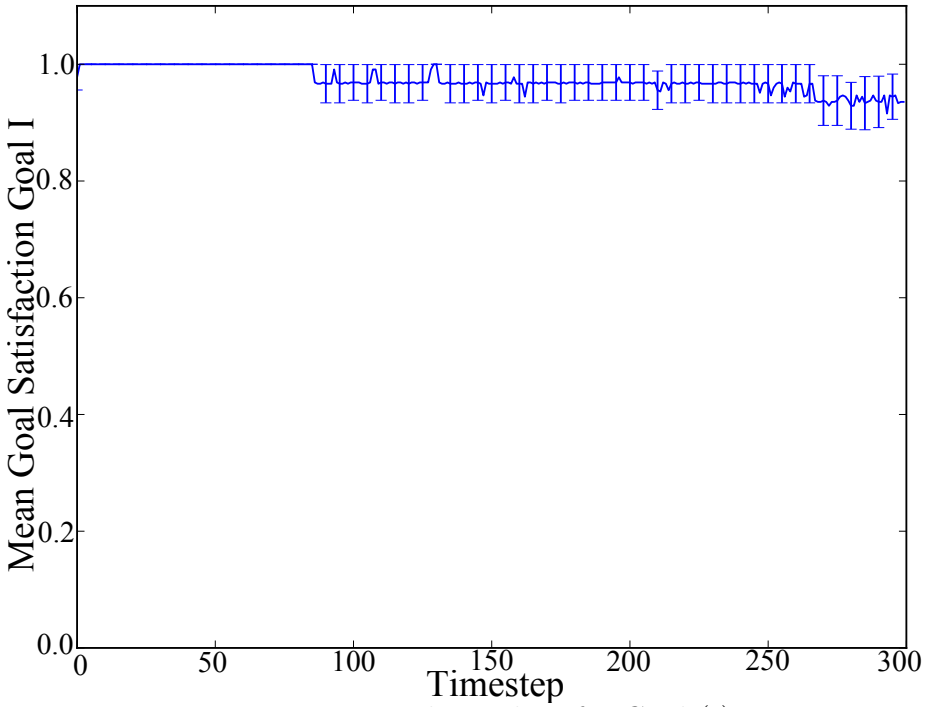


Figure 8.10: Utility values for Goal (I).

continue diffusing data due primarily to network link failures.

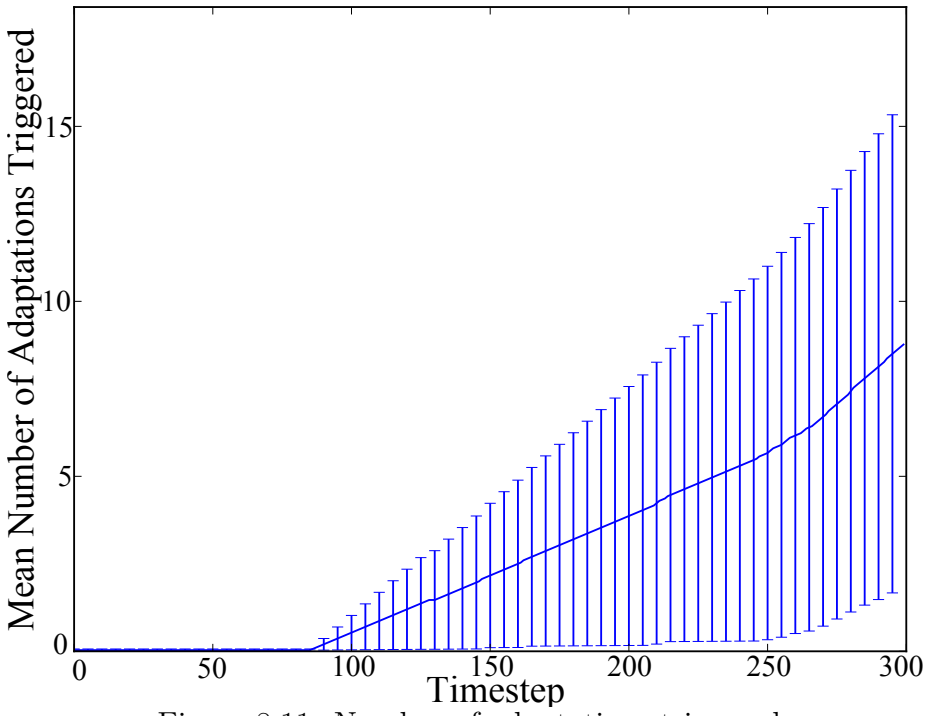


Figure 8.11: Number of adaptations triggered.

8.2 Identifying and Resolving Obstacles

This section presents how our model-based framework supports the iterative revision of a DAS's goal model to address identified sources of uncertainty. Specifically, this section presents how a requirements engineer can analyze the results produced by Loki in order to identify goals that might need revision via goal strengthening or RELAXation. The resulting goal model can then be further improved by reapplying Loki and Athena as necessary.

Identifying Obstacles. Our model-based framework supports iterations between the Athena and Loki techniques where a requirements engineer identifies, analyzes, and resolves sources of system and environmental uncertainty by modifying the DAS goal model. Next, we apply our Loki technique to identify combinations

of system and environmental conditions that produce diverse sets of RDM network behaviors.

Applying Loki to the RELAXed RDM goal model produces an archive of operational contexts that repeatedly cause network link, data mirror, and sensor failures at run time. Network link failures are, by far, the most common adverse environmental condition and typically result in a disconnected network that prevents data mirrors from replicating and distributing data within the allocated simulation time. Figure 8.12 plots the satisfaction of Goal (A) when the RDM was subjected to these operational contexts. As this figure shows, 60% of archived operational contexts caused Goal (A) to become unsatisfied at some point in the RDM simulation. Since Goal (A) is both an invariant *and* the root goal in the model, the model likely requires further revision to improve reliability in the RDM system.

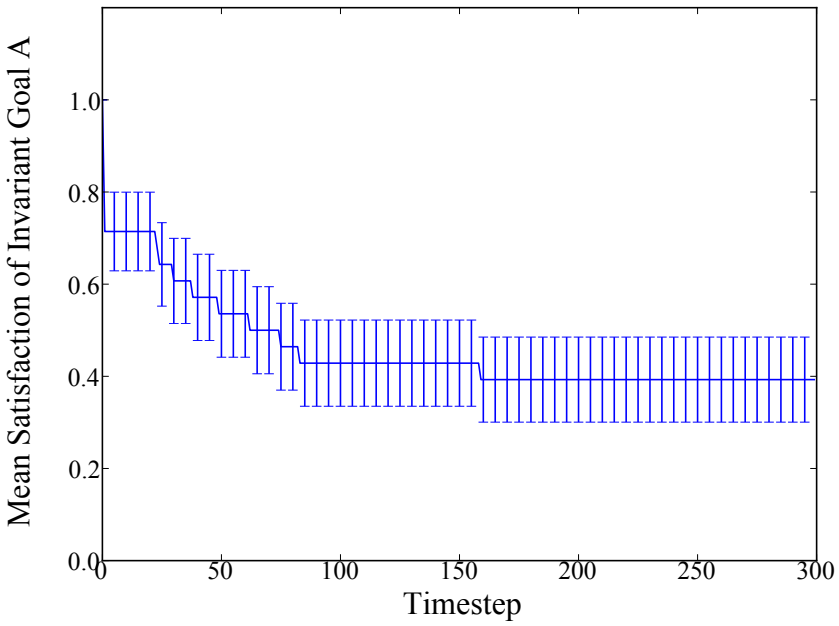


Figure 8.12: Mean satisfaction of Goal (A) under operational contexts in novelty archive.

Next, we search for two types of recurrent patterns in Loki's novelty archive. First, we rank goals by their depth within the goal model (i.e., placement). We then rank

goals within the same depth level by the number of times they were unsatisfied during simulations. When analyzed across all behaviors that involved a requirements violation, this sorting operation basically identifies finer-grained goals that are commonly unsatisfied. Second, for each candidate failed goal, we enumerate the combination of system and environmental conditions leading up to the time step when the goal became unsatisfied. Ideally, these patterns suggest operational contexts that are not properly addressed by the current goal model.

Based on the first filtering criteria described above, we identify Goal (F) as a common failure point in the RDM goal model. As Figure 8.1 shows, Goal (F) originally stated that the RDM network should remain connected during the data diffusion process. Nevertheless, as Figure 8.2 shows, we RELAXed this goal to explicitly account for temporary network partitions due to possible data mirror and network link failures at run time. Specifically, by introducing the "AS FEW AS POSSIBLE" RELAX operator to this goal, we allow temporary network partitions under the assumption that other data mirrors can continue diffusing data while segmented data mirrors are self-reconfigured.

Figure 8.13 depicts a representative and *valid* RDM network topology after introducing the "AS FEW AS POSSIBLE" RELAX operator to goal (F) and subjecting it to the operational contexts in the novelty archive. This figure shows an RDM network that comprises *three* partitions as a result of adverse operational contexts, such as repeated network link failures. In this case, these partitions occur when the network links connecting data mirrors 5-9 and 18-23 fail. These network partitions affect data diffusion in two key ways. First, *Partition 1* captures an isolated data mirror that cannot receive, replicate, or distribute data messages. This isolated data mirror allows the possibility of data loss either by a data mirror failure or a data message corruption. Likewise, *Partition 2* and *Partition 3* capture a component of RDMs that can distribute data amongst themselves, but not to data mirrors in other

partitions. This partition prevents the replication and distribution of data messages between the two other RDM components, thereby increasing the amount of time required to diffuse data.

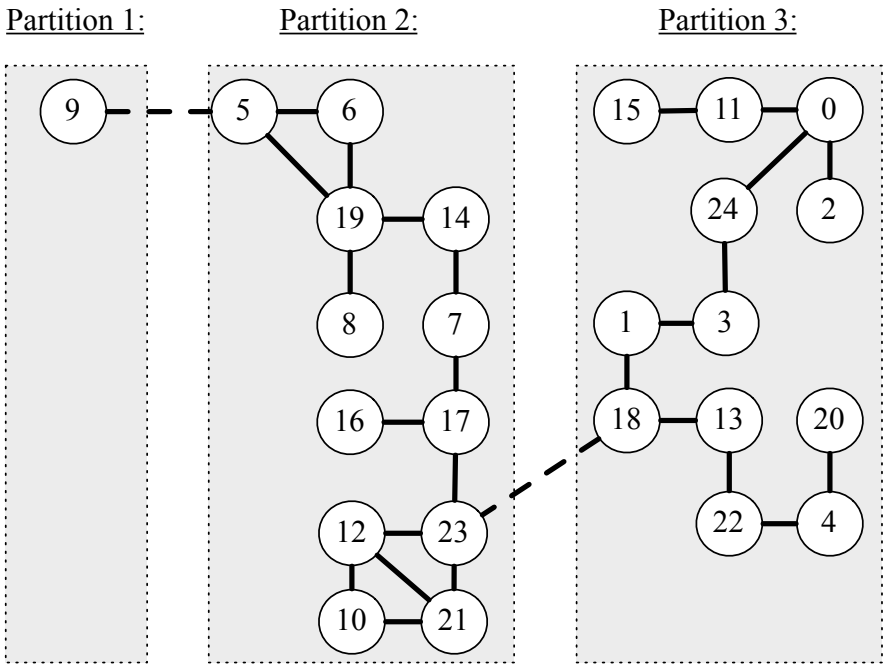


Figure 8.13: Partitioned RDM network that leads to a requirements violation.

In addition, Figure 8.14 plots the mean satisfaction of Goal (F) under the operational contexts in the novelty archive. As this figure shows, Loki introduces adverse environmental conditions such that the RDM network gradually becomes more partitioned as the simulation progresses. As captured in both Figures 8.13 and 8.14, adverse operational contexts caused the RDM network to operate with two to three partitions during the simulation. These RDM network partitions, and their effects upon the data diffusion process, are likely root causes for Goal (A) to become unsatisfied in most trials.

Similarly, Figure 8.15 plots the mean satisfaction of Goal (H) under the operational contexts in the novelty archive. As this figure shows, Goal (H) becomes gradually unsatisfied as the RDM network gets congested with duplicate, dropped, and delayed data message, thereby increasing the amount of time required to diffuse

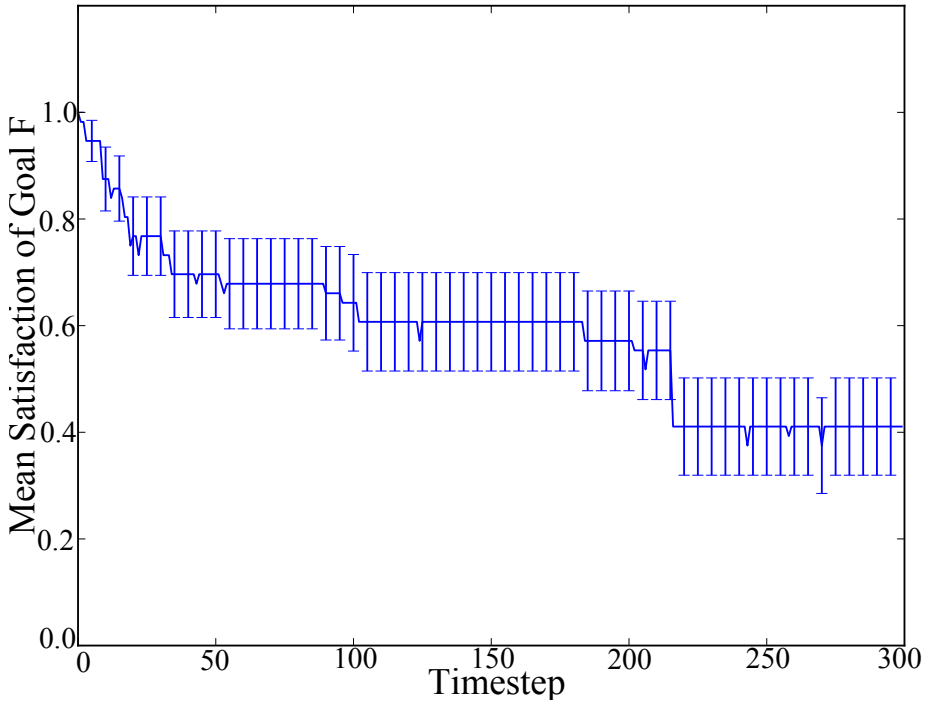


Figure 8.14: Utility values for Goal (F).

data. Together, Figures 8.14 and 8.15 capture the combined effects of adverse environmental uncertainty and how these ultimately prevent the satisfaction of Invariant Goal (A).

While this goal RELAXation seems reasonable, it is insufficiently *strong* to disallow the violation of other invariant goals in the model. Most notably, the weakened satisfaction criteria of Goal (F) affects the satisfaction of Invariant Goal (A) and non-invariant Goal (C) by exposing *new* data to loss and unavailability. Examining the operational contexts associated with this behavior suggests one recurring environmental pattern where network link failures can completely isolate a single data mirror in the RDM network. Unfortunately, *new* data introduced at an isolated data mirror is at extreme risk in the event of a data mirror failure (somewhat unlikely) or a data message corruption (more likely). Moreover, the RDM network *cannot* recover from the loss or corruption of a new data item as it automatically violates the satisfaction of invariant Goal (A).

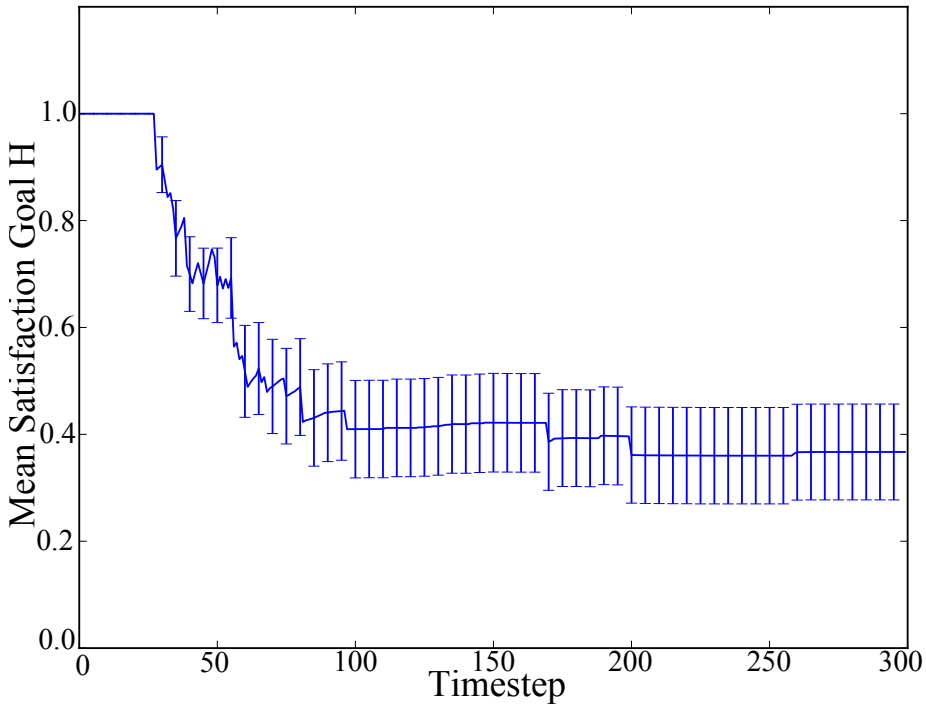


Figure 8.15: Utility values for Goal (H).

8.3 RDM Goal Model Revision

Based on this analysis, we determine that Goal (F) is a prime candidate for goal revision. Although different goal revision strategies exist, such as obstacle prevention and requirement strengthening [107], we apply two uncertainty mitigation strategies for RELAXed goal models previously introduced by Cheng *et al.* [13]. As the elided goal model in Figure 8.16 shows, we introduce another high-level Goal (F'), in this case a sibling to Goal (F), and also introduce a RELAX operator to Goal (F') to account for already identified sources of environmental uncertainty. This specific revision effectively merges the third and second uncertainty mitigation strategies defined for RELAXed goals [13].

Combined, Goals (F) and (F') now state that RDM network partitions can be temporarily tolerated as long as the *order* of each connected component of data mirrors is greater than one. Figure 8.17 presents a sample RDM network that captures this goal model revision. As this figure shows, if a network partition occurs, then

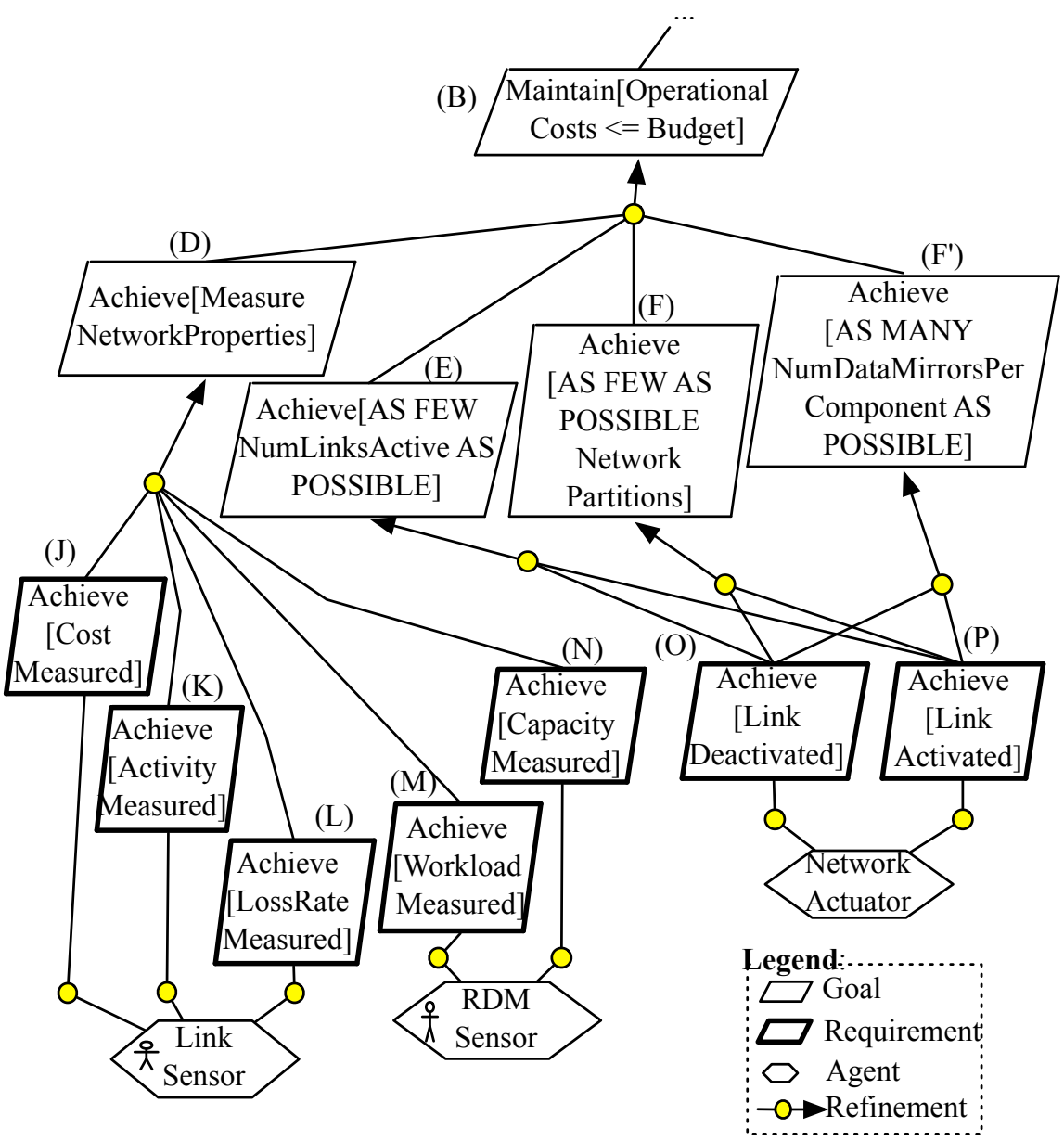


Figure 8.16: Revised goal model with applied uncertainty mitigation strategies for RELAXed goals.

ideally it splits the RDM network into two connected components where the size of each component is approximately half the number of data mirrors in the network. With this change, the RDM network would immediately trigger a self-reconfiguration if a network link or data mirror failure partitions the network such that a data mirror is isolated, regardless of whether the number of partitions exceeds the RELAXed

constraint of Goal (F).

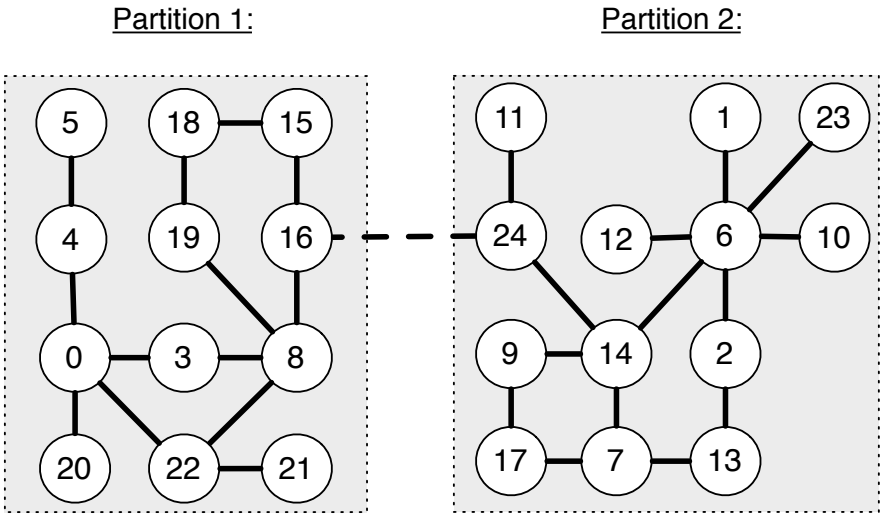


Figure 8.17: Desirable network partition that reduces chances of data loss.

Next, we evaluate the effectiveness of this goal model revision by reusing the operational contexts in the novelty archive as test cases. First, we execute Athena to regenerate utility functions for Goals (F) and (F') that reflect these revisions. We then reuse the same operational contexts that previously caused Goals (F), (C), and (A) to become unsatisfied via isolated data mirrors.

Figure 8.18 plots the satisfaction of Goal (A) after revising the RDM goal model and, subsequently, regenerating its corresponding utility functions for requirements monitoring. As this figure shows, about 80% of the RDM simulations now manage to satisfy all invariant requirements even when subjected to a wide range of adverse operational contexts. Moreover, as Figures 8.12 and 8.18 illustrate, this goal revision alone doubles the number of RDM simulation instances that manage to satisfy Invariant Goal (A).

Likewise, Figure 8.19 plots the satisficement of Goal (F) after the goal model and utility functions were revised. As this figure illustrates, Goal (F) is now satisficed at approximately 80% of its maximum possible value, which corresponds to the RDM network being connected. As such, this satisficement value implies that the RDM

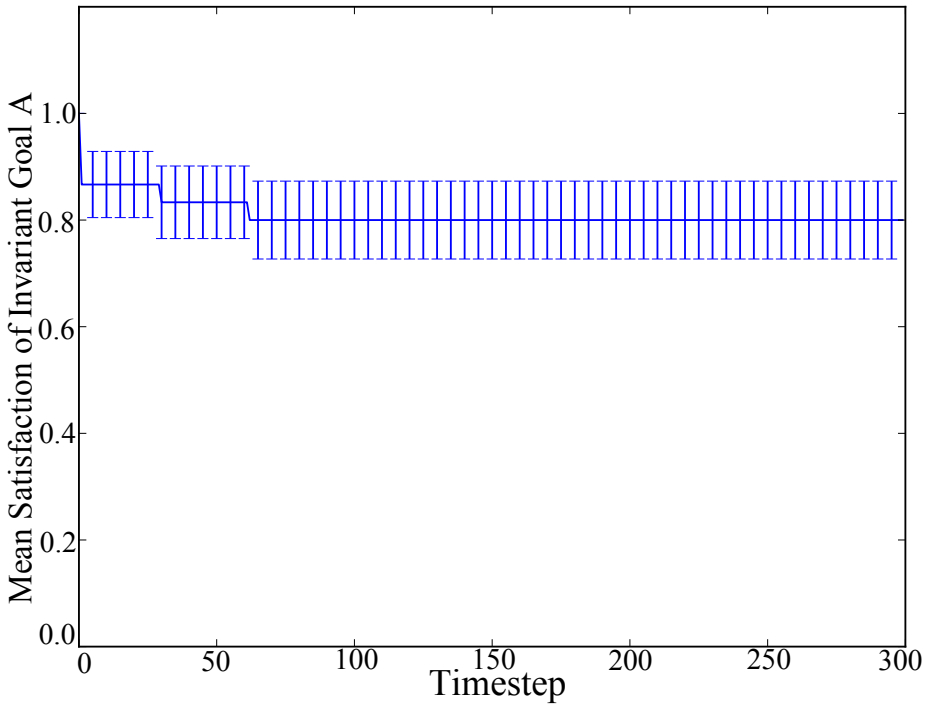


Figure 8.18: Mean satisfaction of Goal (A) under operational contexts in novelty archive after goal model revision.

network was partitioned at most once at any given time. Furthermore, the RDM network does not contain isolated data mirrors at any point during the simulation since it would automatically trigger an adaptation that reconfigures the network.

Simulating this scenario confirms that the RDM network suffers fewer data losses as a result of isolated data mirrors, thus suggesting that our goal revision is effective. Nevertheless, this goal model revision is unable to prevent all data loss as a result of an isolated data mirror. Specifically, one simulation instance still failed to satisfy Goals (A) and (C) because Loki introduced the following events at the same time step *before* the reconfiguration could be triggered: (1) a new data item is introduced at a data mirror, (2) the network link connecting that data mirror to the RDM network fails, and (3) the data mirror also fails thereby wiping its state clean, including the newly inserted data message. Unfortunately, this requirements violation *cannot* be prevented by further revising Goals (F) and (F') nor the RDM goal model itself as there is no mitigation strategy that can prevent the immediate corruption of a new

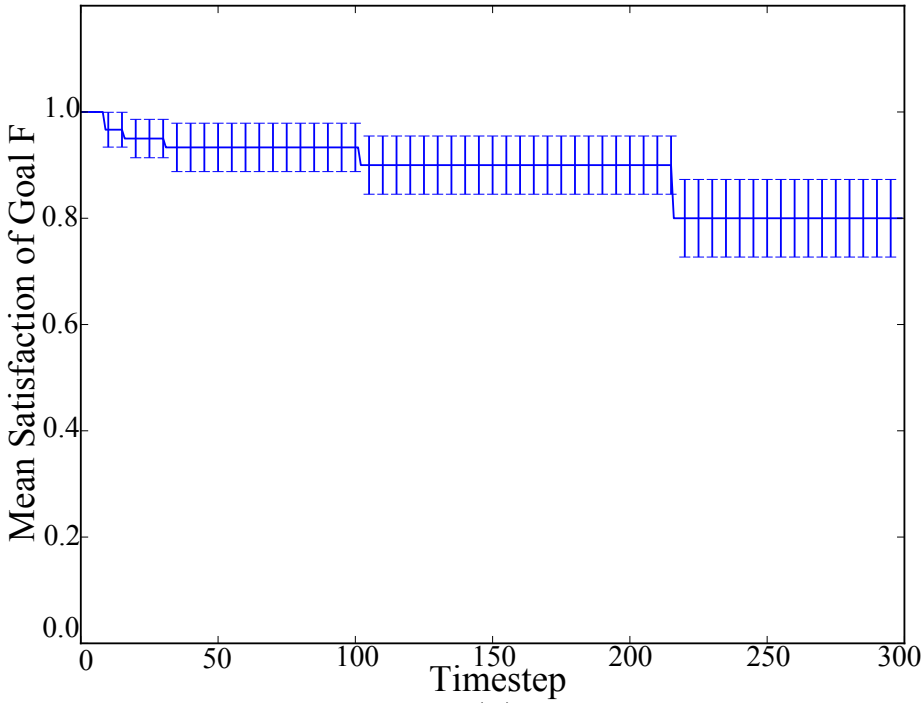


Figure 8.19: Mean satisficement of Goal (F) under operational contexts in novelty archive after goal model revision.

data item that has not been replicated yet.

8.4 Fine-Tuning Utility Functions

The revised RDM goal model and its associated utility functions for requirements monitoring enable the RDM application to better identify and address certain combinations of adverse system and environmental conditions. Nevertheless, system and environmental uncertainty can still prevent the RDM network from fully satisfying its requirements at run time. For example, minor and transient forms of uncertainty may cause the RDM network to unnecessarily self-reconfigure multiple times, thereby diverting resources towards adaptation concerns rather than replicating and distributing data messages. As such, we now continue with our end-to-end example and apply our AutoRELAX technique to the *revised* RDM KAOS goal models, both the unRELAXed and manually RELAXed versions (see Figures 8.1, 8.2, and 8.16). The

resulting fine-tuned non-invariant utility functions should enable the RDM network to better distinguish when an adaptation is warranted, as well as when minor forms of uncertainty can be temporarily tolerated at run time.

Figure 8.20 presents three sets of box plots that capture the fitness values achieved by 1) AutoRELAX-generated models, 2) a manually created RELAXed goal model (Figures 8.2 and 8.16), and 3) an unRELAXed goal model, respectively. As these box plots illustrate, AutoRELAX generates goal models that achieve a statistically significant higher fitness value than unRELAXed and manually RELAXed goal models ($p < 0.001$, Welch Two Sample t-test). These differences in fitness values show how AutoRELAX is capable of refining the initial set of goal RELAXations, and their corresponding utility functions, such that the RDM network satisfies its invariant requirements while minimizing the number of goal RELAXations and the number of triggered adaptations.

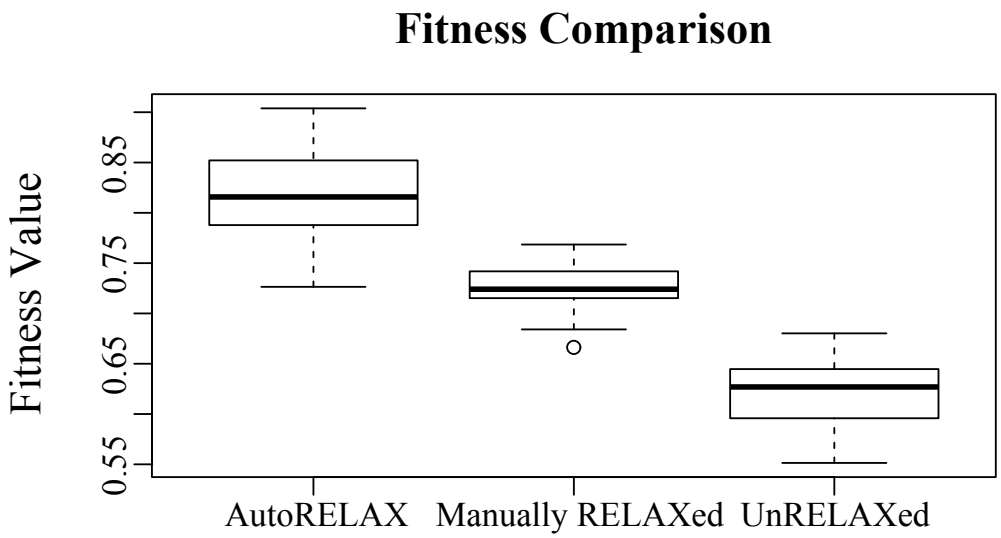


Figure 8.20: Fitness value comparison between AutoRELAX, manually RELAXed and unRELAXed goal models.

Figure 8.21 presents three sets of box plots that capture the adaptation costs incurred by 1) AutoRELAX-generated models, 2) a manually created RELAXed goal model, and 3) an unRELAXed goal model, respectively. Specifically, each set of box

plots measures the amount of time that components in the RDM network spent in active, passive, and quiescent modes *during* reconfigurations (these plots do not include time outside of a reconfiguration). As Figure 8.21 shows, option (1) (AutoRELAX) is preferable because it has less negative impact on overall system functionality. Specifically, by carefully lessening the satisfaction criteria of non-invariant goals, the number of adaptations decrease and so does the cumulative amount of time components spend in passive and quiescent modes during a reconfiguration.

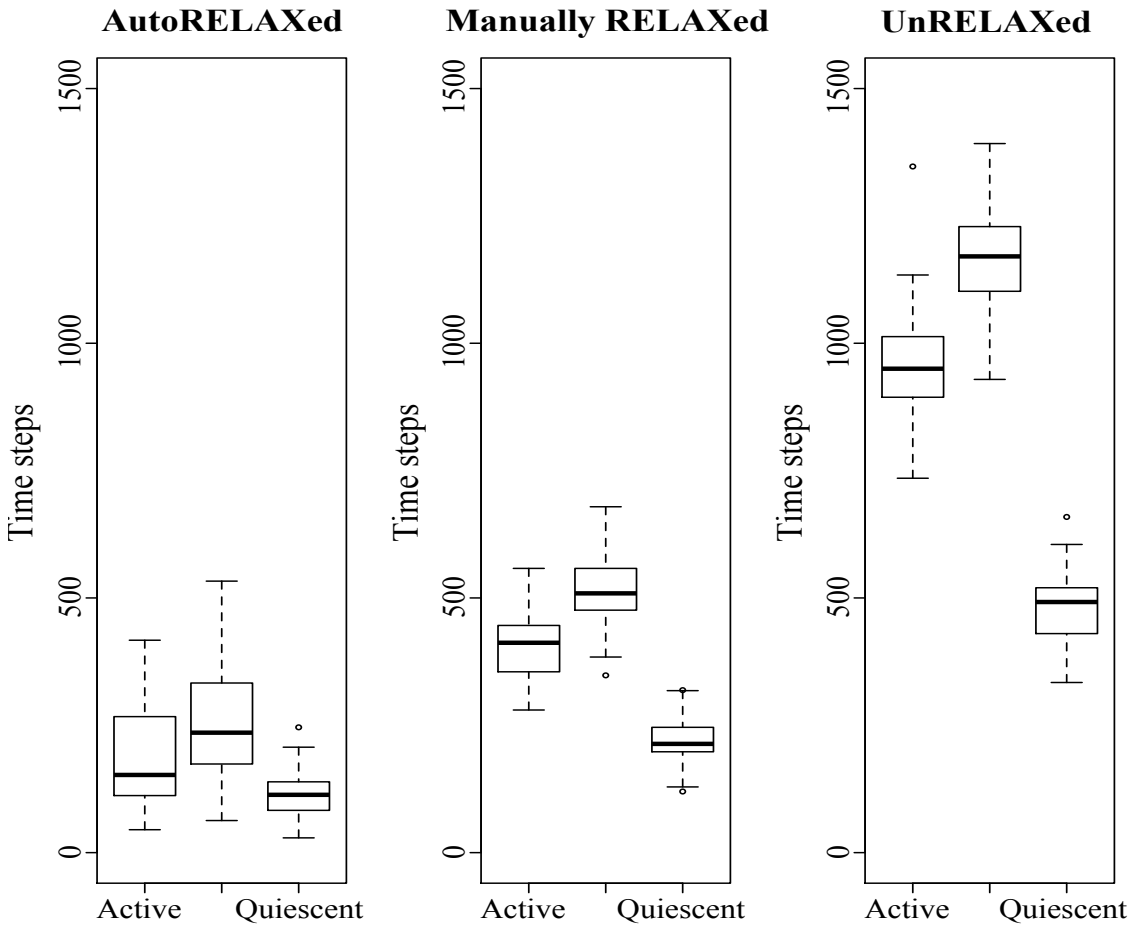


Figure 8.21: Adaptation costs comparison between RELAXed and unRELAXed goal models.

8.5 Dynamic Reconfiguration

Thus far, this chapter has shown how our model-based framework supports the iterative revision of a DAS' goal model. Specifically, we have revised the original RDM goal model (see Figure 8.1) based on the feedback obtained from applying Athena, Loki, and AutoRELAX. Although the revised RDM goal model does improve the reliability of the RDM network by triggering adaptations whenever a data mirror becomes isolated at run time, there are certain additional scenarios that still warrant reconfiguration in order to prevent data loss. This section presents additional experimental results that demonstrate how our model-based framework can leverage Plato and Hermes to dynamically reconfigure the RDM network in response to adverse system and environmental conditions.

Reconfiguration Against Successive Link Failures. As previously described, the RDM goal model revision 8.16 introduces Goal (F') to trigger a self-reconfiguration whenever the RDM network detects an isolated data mirror. We now present how Plato can be applied to our end-to-end RDM case study to dynamically reconfigure an overlay RDM network in real-time in response to multiple link failures. Note that these link failures are likely to disconnect the RDM network *and* isolate data mirrors.

As in Experiment 6.1, Experiment 6.2, and Experiment 6.3, we first ran Plato to produce an initial overlay network design whose primary design objective was to minimize operational costs, i.e., $\alpha_{cost} = 1$, $\alpha_{perf} = 0$, and $\alpha_{rel} = 0$. Figure 8.22 shows the initial overlay network design, essentially a spanning tree. Next, the simulation randomly selects active overlay network links and set their operational status to *faulty*. Since these network link failures tend to disconnect the RDM network and, more importantly, isolate at least one data mirror, Plato was triggered to evolve new target reconfigurations in response to the changing underlying network. This process is repeated several times during the simulation for a maximum of ten consecutive link

failures.

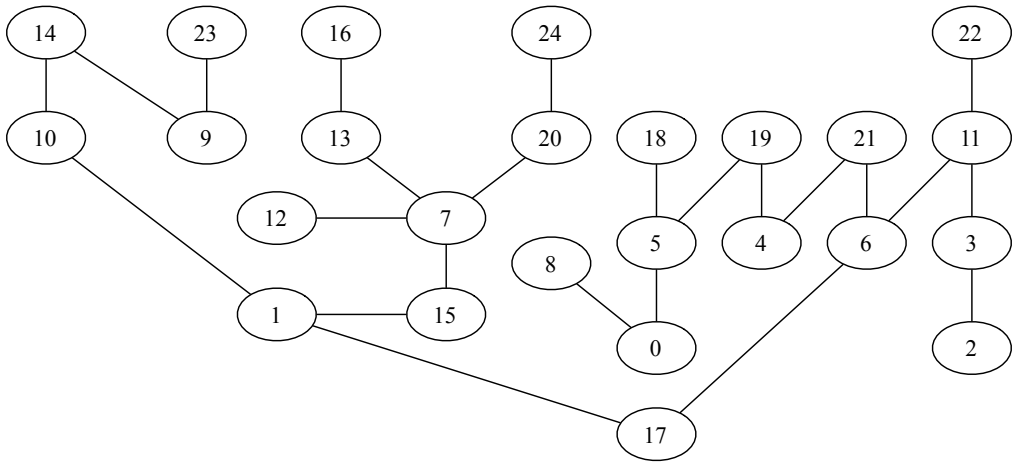


Figure 8.22: Sample RDM network generated by Plato when optimizing for cost.

Figure 8.23 plots the mean maximum fitness achieved by Plato as it evolved overlay network designs. After the initial overlay network design became disconnected as a result of link failures, Plato automatically rescaled reconfiguration priorities to emphasize data reliability, i.e., $\alpha_{cost} = 1$, $\alpha_{perf} = 2$, and $\alpha_{rel} = 2$. This plot illustrates how Plato evolved target reconfigurations that withstood various successive link failures (depicted by the valleys) without the overlay network necessarily becoming disconnected. Specifically, on average, each generated target reconfiguration maintained a fitness value well above -400, which was the numerical penalty assigned to disconnected overlay networks.

Figure 8.24 presents a sample RDM network topology generated in response to adverse environmental conditions, such as network link failures. Plato generated this overlay network such that it comprises 32 active network links, the majority of which use asynchronous propagation methods with either 1 or 5 minute time bounds. Overall, this overlay network provides a combination of performance and reliability while maintaining operational expenses well below the allocated budget.

It is also interesting to observe that during each successive reconfiguration, the first few target reconfigurations generated by Plato were progressively lower in fitness,

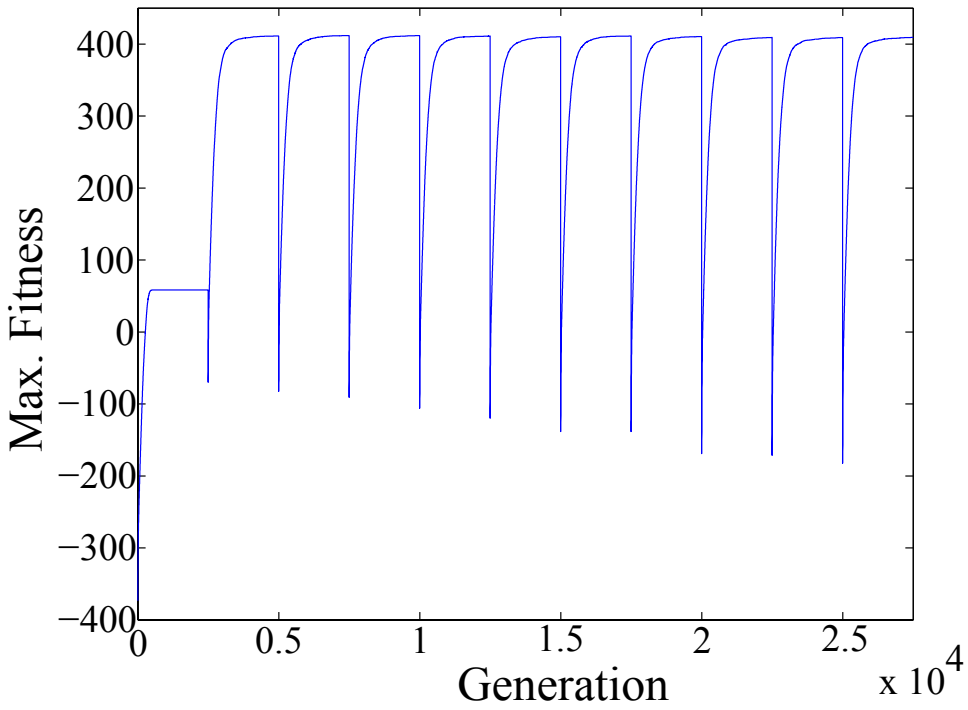


Figure 8.23: Maximum fitness of overlay networks achieved throughout multiple reconfigurations.

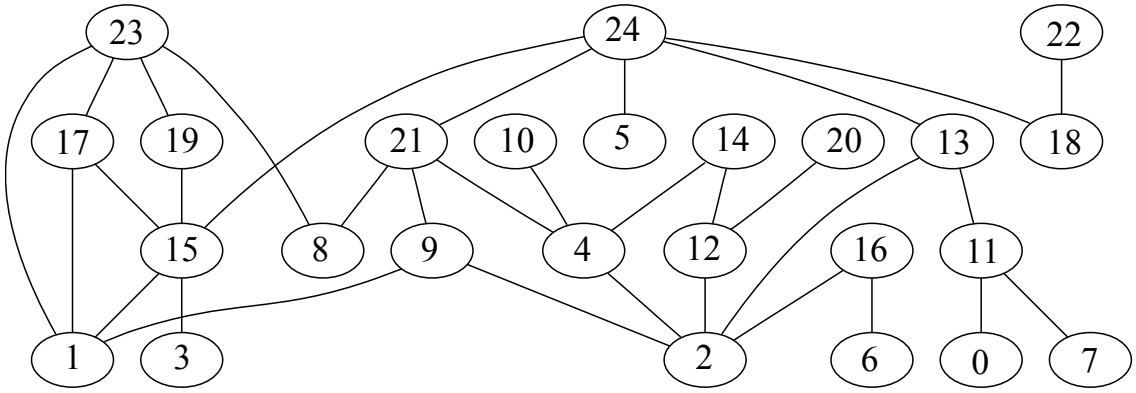


Figure 8.24: Sample RDM network generated by Plato when optimizing for cost, performance, and reliability.

suggesting that as more network links failed in the overlay network, it became more difficult for Plato to find promising areas in the solution space to explore further. Nevertheless, Plato evolved suitable target reconfigurations of the same overall fitness value in response to successive network link failures. Moreover, Plato was able to consistently evolve viable target reconfigurations within 500 generations (approximately

30 seconds).

The plot in Figure 8.25 shows the mean number of active network links in the initial and reconfigured overlay network designs. In the initial overlay network design, Plato reduced the number of active links to form a spanning tree and minimize operational costs. In contrast, throughout each reconfiguration iteration, Plato increased the number of redundant active links to maximize data reliability while still attempting to minimize operational costs. As this plot illustrates, after each successive network link failure, Plato generated target reconfigurations where overlay networks comprised approximately 30 active links.

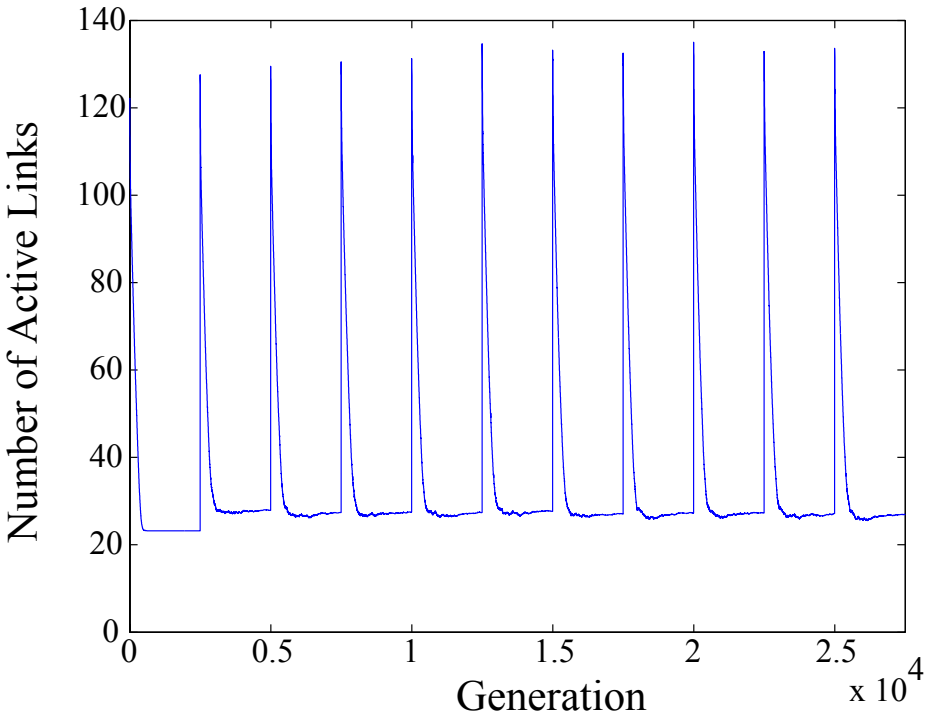


Figure 8.25: Number of active links in overlay network throughout multiple reconfigurations.

Figure 8.25 also provides insights as to how Plato is able to evolve suitable target reconfigurations that balanced maximizing data reliability and minimizing operational costs. Specifically, Plato activated many overlay network links during the first few iterations of the genetic algorithm. Eventually, Plato pruned back the size of the

overlay network by deactivating most redundant network links in order to reduce operational costs. Note that while some redundant network links in the overlay network do increase operational costs, they also improve the robustness of the overlay network design against potential future link failures.

Finally, Figure 8.26 plots the mean potential data loss for a data mirror in the initial and reconfigured overlay network designs. The mean potential data loss measures the amount of data, in gigabytes, that may be lost at a given data mirror as a result of some type of failure. As this plot illustrates, after the initial overlay network design became disconnected, Plato rapidly evolved target reconfigurations where most propagation methods in the overlay network were set to either synchronous mode or asynchronous mode with a 1 or 5 minute time bound. These particular data propagation settings reduced the average potential data loss across the network by providing a higher level of data protection at the expense of degraded network performance.

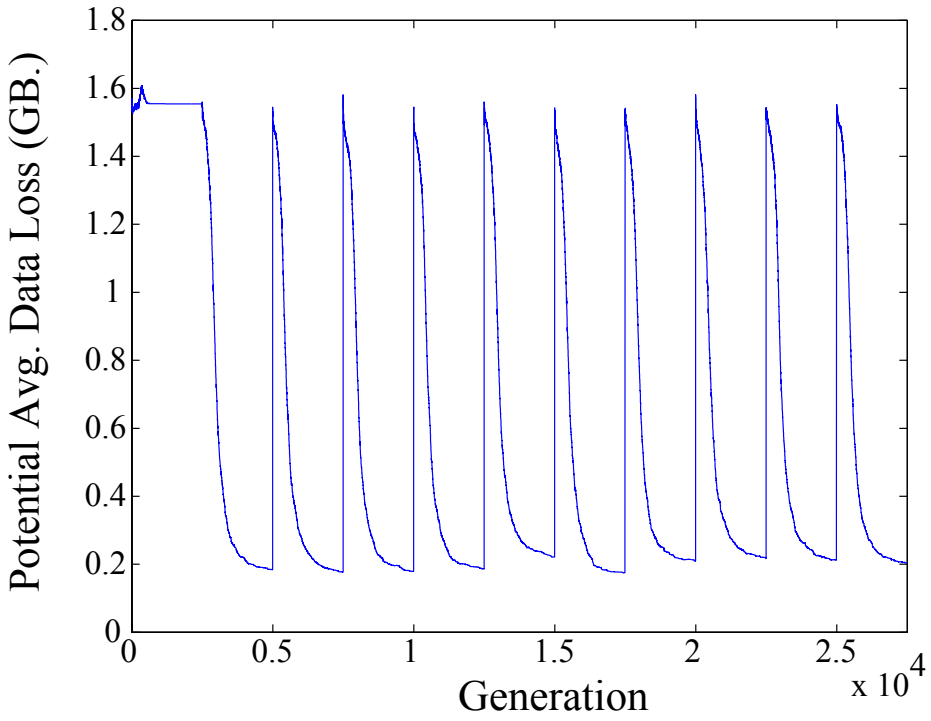


Figure 8.26: Potential average data loss across overlay network throughout multiple reconfigurations.

Optimizing for Reliability. Given the combinations of adverse system and environmental conditions that the RDM network is subjected in this end-to-end case study, we configured Hermes to maximize the reliability of a reconfiguration. Specifically, we set the fitness sub-functions for Hermes as follows: $\alpha_{cost} = 0.2$, $\alpha_{perf} = 0.2$, and $\alpha_{rel} = 0.6$. Such trade-off preferences may arise when the reconfiguration is driven by failures that threaten the functionality of the system rather than by variations in system performance. For example, as Experiment 3.2 and Experiment 4.1 demonstrated, the failure of either a remote data mirror or a connection between remote data mirrors may cause data to be permanently lost. As such, this portion of the end-to-end case study explores scenarios where the cost of losing data is severe.

Figure 8.27 plots the mean maximum fitness values of adaptation paths per generation. As this plot shows, Hermes evolved adaptation paths that achieved an approximate fitness value of 4502. This fitness value represents a 139% improvement over component-dependency analysis whose fitness value is represented by the filled circle plotted at generation 0, before Hermes modifies the initial safe adaptation path.

In general, Hermes achieves higher fitness values by evolving solutions different from the initial adaptation path in two key ways. First, Hermes adds pairs of "*passivate*" and "*activate*" instructions not present in the initial adaptation path. Second, Hermes reorders the sequence of reconfiguration instructions to establish large regions of quiescence throughout most of the reconfiguration. Specifically, passivate instructions are shifted to the beginning of the adaptation path, thereby temporarily pausing most remote data mirrors. Hermes then reconfigures the RDM network before finally reactivating data mirrors. Although this strategy is similar to Kramer and Magee's dynamic change management protocol, Hermes also balances tradeoffs with factors such as performance and cost. Moreover, as Figure 8.27 also illustrates,Hermes achieves large fitness gains within the first 500 generations (< 30 seconds), reaffirming our findings that Hermes can balance competing tradeoffs in a reasonable amount of

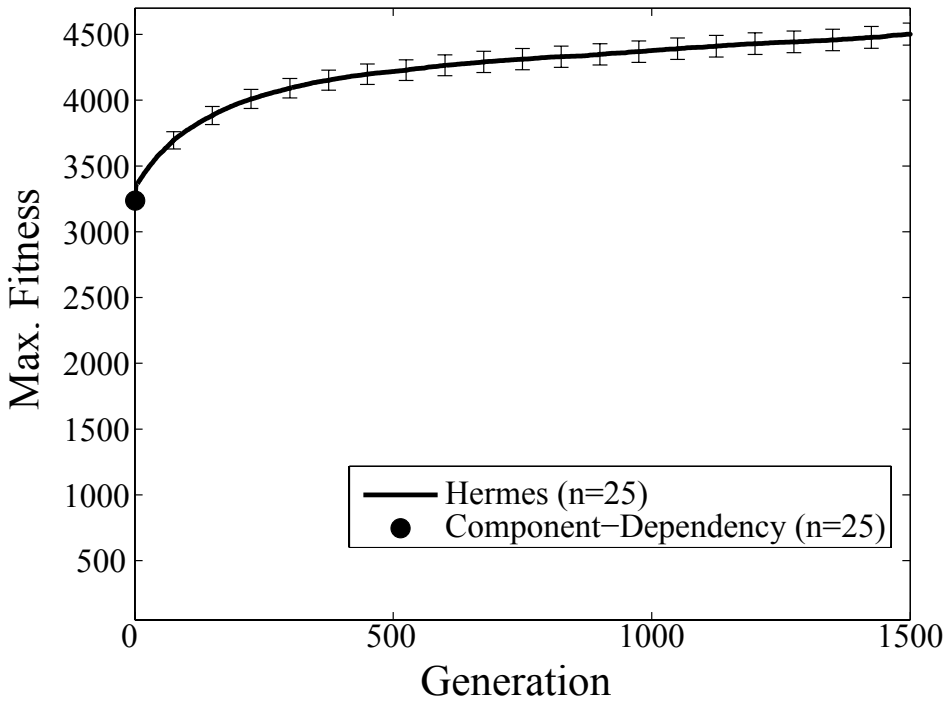


Figure 8.27: Progression of average fitness values when maximizing reconfiguration reliability.

time within the RDM application context.

In contrast to Experiment 7.2, where minimizing reconfiguration disruption was the primary concern, in this end-to-end case study we maximized the reliability of the reconfiguration itself. As Figure 8.28 illustrates, Hermes increased reconfiguration costs by approximate 33 seconds more than the initial adaptation path. This 12% increase in reconfiguration costs is a direct result of the additional passivate and activate instructions inserted by Hermes. Interestingly, these pairs of additional instructions were sometimes applied to data mirrors that were not directly involved in the reconfiguration process. Were it not for these additional instructions, these remote data mirrors would have propagated data during the entire reconfiguration. Thus, by passivating most data mirrors, Hermes provided better data reliability at the expense of higher reconfiguration costs.

Lastly, Figure 8.29 plots the amount of data (in MB) sent and queued by RDMs

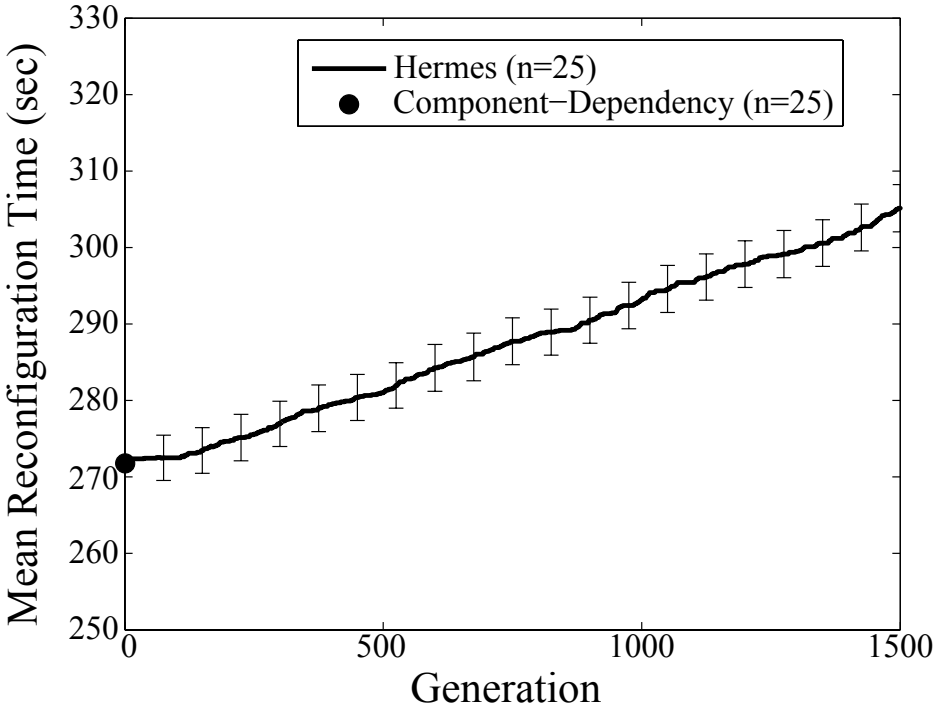


Figure 8.28: Average time required to complete reconfiguration when maximizing reconfiguration reliability.

during reconfiguration. Data sent measures the amount of data RDMs diffused during reconfiguration as a result of being *active*. Likewise, data queued measures the amount of data produced but not diffused by RDMs because their operational status was set to *passive* state. As this plot illustrates, Hermes gradually evolved solutions that queued larger amounts of data than was being diffused throughout the network. Furthermore, this plot also illustrates how reconfiguration time increased as a result of passivating a greater number of data mirrors, thus also increasing the amount of data produced and diffused during this time. Lastly, these results also confirm the tradeoffs observed in Experiment 7.2 between the performance and reliability of adaptation paths. Namely, maximizing reliability typically implies a higher level of system disruption. While it is generally undesirable to disrupt system services by passivating large numbers of remote data mirrors, doing so creates a region of quiescence that better protects data against failures during reconfiguration.

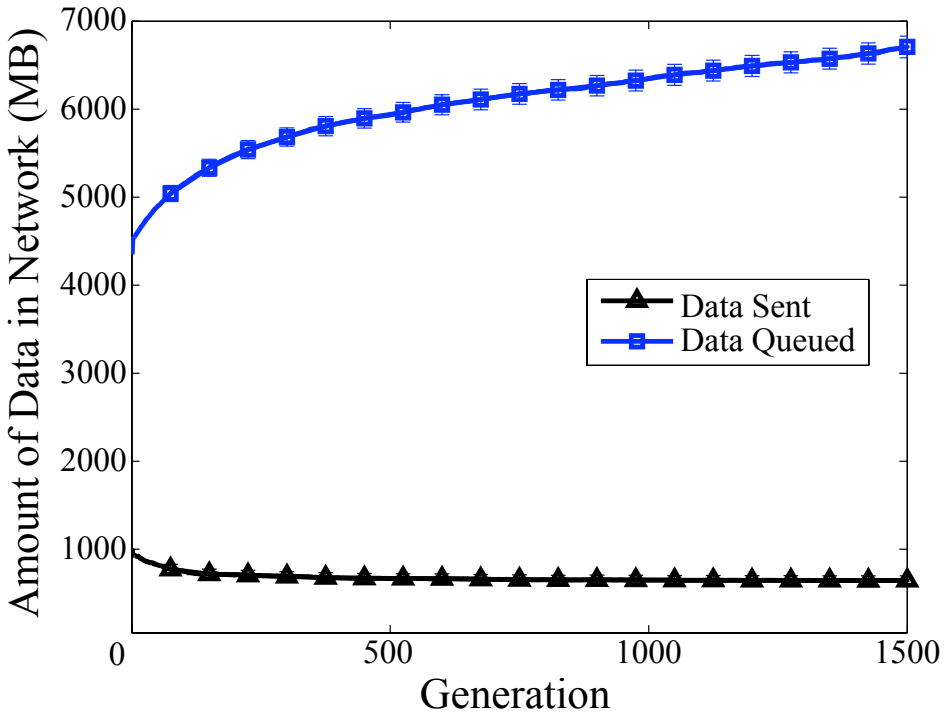


Figure 8.29: Reconfiguration and performance tradeoffs in evolved solutions.

Chapter 9

Related Work

Although DASs have steadily gained attention over the past several years from a wide range of engineering disciplines [15, 65, 83], the concept of dynamic reconfiguration has existed since the early days of computing. Specifically, first attempts at self-modifying code supported run-time program optimization and explicit management of physical memory [73]. Due to the lack of tools and techniques to abstract low-level adaptation details, these programs were often complex and difficult to understand. Eventually, new approaches and techniques for building adaptive systems emerged. For example, error detection and error handling capabilities essentially provided systems with self-adaptive behaviors to prevent or resolve undesirable behaviors or actions [37]. While these approaches demonstrated that adaptation was both possible and powerful, they were often tightly coupled with application-specific source code and applied in *ad hoc* manners. As a result, the first generation of adaptive systems were often difficult to write, debug, and maintain.

This chapter presents related work on model-based approaches for designing and implementing DASs. First, we overview approaches that use goal-oriented requirements models to guide the design and dynamic reconfiguration of a DAS. We then present approaches that use design-level models to guide the development and dy-

namic reconfiguration of a DAS.

9.1 Requirements Models for Adaptive Systems

In this section we present related work that leverages goal-oriented requirements models in the order in which they would most likely be used to guide the design and dynamic self-reconfiguration of a DAS. First, we overview approaches for specifying and modeling the requirements of a DAS. We then describe approaches for identifying and resolving obstacles that can prevent the satisfaction of requirements in a DAS. Next, we overview approaches for identifying, modeling, and mitigating sources of uncertainty that can affect the behavior of a DAS. Lastly, we present approaches for monitoring the satisfaction of requirements at run time in adaptive systems.

9.1.1 Specifying Dynamically Adaptive Systems

Berry *et al.* [7] identified four different levels of requirements engineering for DASs. The first level, similar to traditional requirements engineering for computer-based systems, focuses on eliciting and specifying the functionalities that a DAS must provide. To this end, a requirements engineer must specify how a target program should react in response to each possible input that it can receive. The second level consists of a DAS deciding if, when, and how it should adapt in response to the monitoring information it gathers from its execution environment. The third level focuses on identifying and designing the various adaptation elements needed for a DAS to support the adaptations that will be performed at the second level. At this level, an adaptation engineer must evaluate both the mechanisms and effects of possible adaptations in response to new environmental inputs. Lastly, the fourth level consists of researching different adaptation mechanisms, such as middleware and frameworks, to support dynamic adaptation.

Goldsby and Cheng [38] presented a goal-oriented approach that captured each of the four levels of requirements engineering for DASs using the KAOS specification and modeling language [18]. In their approach, the first level of requirements engineering for DASs uses a goal model to specify the functionality that a DAS must provide to its clients. Similarly, in the second level, a requirements engineer uses a goal model to specify the environmental conditions and appropriate target systems that a DAS can reach by self-adapting. Lastly, the third and fourth levels use goal models to specify and select amongst different adaptation infrastructure elements. In this manner, the goal-oriented approach by Goldsby and Cheng explicitly captures the roles and contributions of each stake holder throughout each of the four levels of the requirements engineering process for DASs.

Although the second level of requirements engineering for DASs focuses on specifying how a DAS should react in response to each possible input that it can receive, achieving this objective is often infeasible due to system and environmental uncertainty [114]. Specifically, it is often infeasible for a human to identify, at design time, all possible combinations of environmental conditions that might arise at run time and how these affect the behavior of the DAS. As such, Whittle *et al.* [114] introduced the RELAX requirements specification language to identify and assess sources of uncertainty in a DAS. The RELAX language, previously described in Chapter 2, uses fuzzy logic-based operators to relax the satisfaction criteria of non-invariant requirements and explicitly address the effects of uncertainty.

The model-based framework presented in this dissertation spans the third and fourth levels of the four levels of requirements engineering for DASs. Specifically, this dissertation presents research on new adaptation mechanisms that form part of the fourth level of requirements engineering for DASs. In turn, these new adaptation mechanisms can be applied to automatically perform several tasks at the third level of requirements engineering for DASs, such as automatically identifying sets of en-

vironmental conditions that might cause a DAS to reconfigure and discovering safe adaptations in response to these inputs.

9.1.2 Requirements Modeling in a Dynamically Adaptive System

Several techniques have been proposed and used to model software requirements in DAS [13, 28, 38, 67, 77, 78]. In general, most of these goal-oriented modeling approaches capture the adaptation capabilities of a DAS as alternate goal operationalizations. For instance, Morandini *et al.* [77, 78] explicitly model system goals, as well as goal achievement conditions and specific recovery actions in the form of alternate goal realizations. Specifically, if goals are unsatisfied, then the DAS switches between different goal realization strategies.

In a different approach, Feather *et al.* [28] used the KAOS goal specification language to model the requirements and constraints of the DAS, as well as the conditions that the DAS should monitor and the possible adaptations that the DAS could apply. At run time, the DAS traces its behavior through these models in order to detect conditions that warrant reconfiguration, such as a requirements violation. Likewise, Lapouchnian *et al.* [67] used hybrid goal models to capture system-wide goals of a DAS. Their approach combined elements from the KAOS and i* [118] modeling languages to specify conditions that might require adaptation in a DAS.

In contrast to these goal modeling approaches, our model-based framework does not explicitly focus on how to specify and model the requirements, constraints, and possible adaptations of a DAS. Instead, our model-based framework uses these goal modeling approaches as enabling technologies to focus on explicitly supporting the automatic identification of obstacles in a DAS and the generation of safe adaptations in response to adverse environmental conditions. The new set of obstacles and adaptations identified by our model-based framework can then be modeled using any of

these goal-based modeling approaches for DASs.

While useful throughout the design and execution of a DAS, the previous set of goal-oriented requirements modeling approaches are static in nature and do not normally support the evolution of models at run time in response to changes in requirements and environmental conditions. As such, Bencomo *et al.* [6, 100] suggested promoting requirements to live run-time entities whose satisfaction can be evaluated in support of adaptation decisions. Similarly, Souza *et al.* [103] introduced a feedback loop-based Awareness Requirements (AwReqs) construct where meta-level requirements capture, monitor, and manage the satisfaction of other system requirements. These *requirements aware* approaches integrate requirements into the decision-making process of a DAS, thus linking adaptation decisions with the requirements they are intended to satisfy.

Our model-based framework also treats requirements as live entities that can evolve and be leveraged at run time. Specifically, through Athena, our model-based framework directly supports run-time requirements monitoring as well as linking DAS self-reconfigurations to specific goals and requirements that must be satisfied. In contrast to the goal-oriented modeling approaches by Feather [28] and Lapouchnian [67], and the requirements-aware approaches by Souza *et al.* and Bencomo *et al.*, our model-based framework supports the management and run-time monitoring of RELAXed requirements to account for system and environmental uncertainty.

9.1.3 Obstacle Identification, Analysis, and Resolution

Obstacle analysis facilitates the systematic identification and resolution of conditions that prevent a system from satisfying its goals and requirements at run time. Letier and van Lamsweerde proposed a set of heuristics, refinement patterns, and formal techniques from for systematically identifying, analyzing, and resolving obstacles from goal specifications [105, 106, 107]. In these approaches, a requirements

engineer first identifies an obstacle by selecting a goal, requirement or assertion that might become unsatisfied at run time and then analyzing its likelihood and criticality. Different mitigation strategies, such as prevention, resolution, or toleration, can be applied to address each identified obstacles.

Although it is ideal to identify and mitigate every possible obstacle that might prevent a DAS from satisfying its goals, achieving this objective for many application domains is infeasible given the large space of potential obstacles that might arise at run time [13, 28, 114]. Our model-based framework complements the obstacle analysis process introduced by Letier and van Lamsweerde by facilitating the automatic discovery of system and environmental conditions that produce diverse behaviors, including requirements violations and latent behaviors. The new set of obstacles identified by Loki can then be analyzed and disallowed via the same obstacle analysis mechanisms that Letier and van Lamsweerde presented.

In a related approach, Letier and van Lamsweerde introduced a framework for specifying partial degrees of goal satisfaction [69], where satisfaction is expressed in terms of probabilities. These probabilities can be obtained directly from stakeholders or derived from actual system usage data. The satisfaction of each goal is then evaluated by applying application-specific refinement equations or objective functions. Several heuristics are proposed for deriving such objective functions in terms of system goals. Furthermore, the authors also provide a top-down and bottom-up approach for propagating satisfaction probabilities throughout the goal model. Ultimately, these probabilities can be used to prioritize the set of goals that should be revised in order to maximize the degree of goal satisfaction.

Our model-based framework and the probabilistic partial goal satisfaction framework presented by Letier and van Lamsweerde share the same objectives of identifying unsatisfied goals and how these may affect the satisfaction of other system goals. Nevertheless, several key differences exist between the two approaches. First, a require-

ments engineer may only apply their framework during the requirements and design phases of the software development cycle. Our model-based framework, on the other hand, can be applied during those same phases of the software development lifecycle, as well as at run time to monitor requirements after the system has been deployed. Second, their framework requires not only a goal model of the system, but also real usage data that captures different instances where goals and requirements have been satisfied and violated. Unfortunately, this information may be unavailable or difficult to acquire for various application domains. In contrast, our model-based framework only requires a goal model to derive utility functions for requirements monitoring.

Notwithstanding these differences, our model-based framework and Letier and van Lamsweerde's partial goal satisfaction framework can complement each other in multiple ways. For instance, our model-based framework can leverage the likelihood and severity of goal violations identified by their probabilistic framework to prioritize how often to compute utility values at run time. Likewise, our model-based framework can output traces of utility values at run time to archive the extent to which goals were satisfied during execution. These utility value traces can then be analyzed to derive and refine the likelihood and effects of goal violations in their partial goal satisfaction framework. Moreover, the obstacles (i.e., system and environmental conditions) identified by Loki can also be reused to validate and refine the probabilistic values assigned to each goal in their partial satisfaction framework, as well as refine the impact of that goal becoming unsatisfied at run time.

Lutz and Mikulski [71] identified mechanisms for discovering and resolving *requirements* errors during the software testing phase. The objective of these mechanisms is to resolve incomplete requirements, unexpected requirements interactions, and requirements confusions by testers. Through Loki, our model based-framework supports these mechanisms for requirements discovery and resolution as it supports the automatic discovery of both incomplete requirements, in the form of latent and

unexpected behaviors, and unexpected requirements interactions, in the form of requirements violations due to missing or inadequate obstacle mitigations.

Glinz *et al.* [101] proposed simulating the specification of a system in order to either validate or localize defects in the requirements of the system-to-be. In their approach, a requirements engineer manually configures the simulation to test the specification. More recently, genetic algorithms have also been applied to generate test cases in simulation-based validations of software specifications where the objective is to exercise as much of the specification as possible [66, 81]. For example, Nguyen *et al.* [81] proposed an approach for testing the behavior of an autonomous agent in response to, among other things, environmental conditions generated by a genetic algorithm. To leverage these approaches, however, developers must extend the genetic algorithm with domain-specific fitness functions that evaluate the quality of generated test cases. Through Loki, our model-based framework supports the same objective of exercising the specification of a DAS. However, in contrast to the approaches by Lajolo and Nguyen, Loki replaces the fitness function of a genetic algorithm with a novelty metric that does not need to be redefined across different application domains.

9.1.4 Identifying and Mitigating Sources of Uncertainty

Researchers are acknowledging the fact that uncertainty permeates the design, implementation, and execution of a DAS [5, 6, 13, 25, 26, 69, 92, 100, 111, 112]. Much recent work has focused on explicitly identifying and documenting sources of uncertainty and evaluating their possible effects upon the DAS. These approaches, and others, attempt to raise awareness about how uncertainty affects a DAS all the way from requirements elicitation to run-time decision-making. In general, these approaches explore the effects of uncertainty in the form of missing requirements, unexpected requirements interactions, and unpredictable or misunderstood system and environmental conditions [71, 112]. For instance, Welsh et al. [111, 112] introduced

the concept of a Claim as a marker of uncertainty in requirements and design decisions baed on possibly incorrect assumptions. Likewise, Esfahani et al. [25, 26] proposed an analytical framework that combines mathematical approaches for modeling and assessing uncertainty in adaptation decisions from a risk-management perspective.

In addition to designing frameworks for documenting and managing sources of uncertainty, researchers have also focused on leveraging fuzzy set theory to represent and analyze the effects of uncertainty in requirements. For instance, Whittle et al. [114] introduced the RELAX requirements specification language to facilitate the identification and analysis of sources of environmental uncertainty in a DAS. Cheng *et al.* [13] extended the RELAX requirements specification language with a goal-oriented modeling process that can be applied to explicitly address uncertainty in a DAS. In their proposed process, a requirements engineer identifies non-invariant goals in a KAOS goal model of the DAS whose satisfaction can be affected due to uncertainty and then selects an appropriate RELAX operator to relax that goal's satisfaction criteria accordingly. In a similar approach, Pasquale et al. [5, 84] introduced FLAGS, a KAOS goal modeling framework that introduces the concept of a fuzzy goal whose satisfaction can also be evaluated through fuzzy logic functions. Both goal-modeling approaches use fuzzy logic-based functions to relax the satisfaction criteria of goals in a goal-oriented model. In contrast to RELAX, however, FLAGS does not focus on identifying sources of uncertainty, but rather evaluating the degree to which a goal is satisfied.

These approaches for documenting, representing, analyzing, and managing uncertainty ultimately depend on a requirements engineer who evaluates the likelihood of various sources of uncertainty arising at run time, as well as their effects upon a DAS. In particular, a requirements engineer using either RELAX, FLAGS, or Claims must currently manually determine which goals may become unsatisfied at run time without disrupting invariant requirements, as well as how much flexibility can be in-

troduced into each non-invariant goal's satisfaction criteria. Our model-based framework complements these approaches in two key ways. First, Loki supports the automatic identification of sources of uncertainty and how these may affect the abilities of a DAS to satisfy its requirements. In addition, AutoRELAX automates the entire identification and assessment process by generating goal RELAXations that specify the extent to which goals may become temporarily unsatisfied at runtime without violating invariant requirements. Although our model-based framework currently focuses on RELAXing functional non-invariant KAOS goals, it can also be extended to automatically identify fuzzy goals in FLAGS, as well as automatically RELAX the satisficement criteria of soft goals [16].

9.1.5 Requirements Monitoring in Adaptive Systems

Feather [28], Fickas [29], and Robinson [97, 98] developed requirements monitoring frameworks intended to detect when an executing system fails to satisfy its requirements. These frameworks support the instrumentation, monitoring, and diagnosis of an executing system. To leverage these frameworks, a requirements engineer first models a DAS with a goal-based modeling language, such as KAOS [18, 105]. Assumptions and constraints that can become violated at run time can then be identified from these goal models. A requirements engineer then creates state-based models of the DAS such that the requirements monitoring framework can trace through them with execution data. At run time, the monitoring framework observes traces of the executing system and logs any unsatisfied requirement or constraint.

The requirements monitoring frameworks by Feather, Fickas, and Robinson share similar objectives with our model-based framework. In particular, their frameworks support the identification of conditions that signal when a DAS has deviated from its expected behavior, such as when a requirement is no longer satisfied. Nevertheless, while their frameworks use state-based models to monitor requirements, our model-

based framework uses a goal-oriented model to automatically generate utility functions that can monitor the satisfaction of functional, non-functional, and RELAXed requirements. In this manner, our model-based framework automates the task of encoding conditions that a DAS should monitor at run time. Additionally, our model-based framework supports the monitoring of RELAXed requirements, which, to the best of our knowledge, are not supported by other requirements monitoring framework. Lastly, unlike their requirements monitoring frameworks, our model-based framework also supports the on-demand generation of safe adaptations in response to current system and environmental conditions.

Recently, utility functions have been applied for self-assessment purposes in DASs [15, 20, 109]. For instance, Walsh *et al.* [109] manually developed and applied utility functions to map monitoring data to a single scalar value representative of how well the system was executing, akin to the concept of a *health* value. Leveraged in this manner, utility functions provide not only an objective and quantitative basis for automated decision-making, but also facilitate the mapping of those decisions to higher-level goals, requirements and concerns. Similarly, utility functions have also been applied within the context of a DAS to guide the selection of self-optimization strategies. For example, Garlan *et al.* [15] applied utility functions to evaluate and select among among different reconfiguration strategies depending on how each strategy satisfied different architectural and performance-based constraints. Although these utility functions provide numerous benefits for decision-making within a DAS, they are usually manually elicited from either a domain expert or an application user [20]. As such, these *ad-hoc* approaches for manually eliciting utility functions may not capture a comprehensive set of utility functions to monitor all application requirements.

In addition to out model-based framework, several automated approaches have been developed for automatically deriving utility functions for requirements monitoring in a DAS. For instance, several statistical regression-based techniques have been

applied to infer utility values that capture the overall performance of a DAS at run time [12, 20]. Likewise, Wong *et al.* [116] developed a genetic programming-based approach to compose various metrics that can detect when a DAS behaves abnormally. While these approaches automate the task of deriving performance-oriented utility functions, their success ultimately depends on the quality of monitoring data gathered by the DAS. Specifically, regressed utility functions may inadvertently miss the detection of anomalous behaviors if the training data is either incomplete or contains undesirable behaviors. Moreover, these approaches tend to delay the integration of utility functions until deployment-time when real execution data is available to drive the regression process. Our model-based framework does not suffer from either drawback as it derives utility functions directly from a goal model.

Requirements monitoring frameworks and self-assessment utility functions can also be extended to leverage the operational contexts discovered, evaluated, and archived by our model-based framework. Specifically, Loki's novelty archive contains a generalized collection of system and environmental conditions that are associated with specific DAS behaviors and how these affect the ability of the DAS to satisfy its requirements. If a requirements monitoring framework determines that the DAS is not executing within any of its archived operational contexts, then it might reflect an unanticipated system and environmental condition. In such scenarios, the DAS can instruct the requirements monitoring framework to collect additional data, diagnose the effects of the new operational context, and if applicable, add it to the archive along with its reconfiguration decision and outcome.

9.2 Design Models for Adaptive Systems

This section presents related work that uses design-level models to guide the implementation and dynamic reconfiguration of a DAS. First, we present approaches

that use design-level models to guide the systematic and rigorous development of a DAS. We then present several frameworks that leverage models, such as architectural models, to guide the dynamic reconfiguration of a DAS.

9.2.1 Model-based Approaches for Developing Adaptive Systems

Zhang and Cheng [120] previously introduced a model-based development process to guide the rigorous development of adaptive programs. A key benefit of their process is that it separates the adaptive and non-adaptive behavior specifications of an adaptive program. Doing so results in models that are easier to specify and more amenable to visual inspection. Starting with high-level goals, their proposed process begins by identifying sets of invariant properties that should always be satisfied by the DAS. Their process then progresses through to design models that specify, for different domains, the environmental conditions under which a specific target program will execute. Next, their process focuses on identifying and verifying the correctness and safety of adaptation transitions responsible for transitioning the DAS between pairs of target programs. Lastly, these models can serve to either generate rapid prototypes or to guide the development of the resulting DAS. In this manner, the focus of their process is the specification of key properties at each of the major development phases.

The model-based development process proposed by Zhang and Cheng is complementary to the model-based framework we present in this dissertation. In particular, our model-based framework supports the automatic identification of different domains, and their corresponding system and environmental conditions, that a DAS may need to execute in. Furthermore, our model-based framework supports the generation of both target system reconfigurations and safe adaptation paths in response to system and environmental changes. Within their model-based development process, these operational domains, target system reconfigurations and safe adaptation

paths correspond to local models applicable under a specific set of environmental conditions and adaptive models that safely transition the executing system from a source program to a target program, respectively.

Software engineering researchers have also developed design patterns for guiding the development of a DAS. In particular, a design pattern is a generic design-level solution that can be reused to address recurring problems in a particular domain [34]. To this end, Gomaa *et al.* [42] developed four design patterns for reconfiguring software architectures at run time. These four design patterns specify the behavior required to dynamically reconfigure specific types of architectures; supported architectures include master/slave, centralized, server/client, and decentralized architectures. For each design pattern, Gomaa *et al.* identified when it is safe to perform a reconfiguration and provided hierarchical UML state diagrams illustrating the necessary behavior. Although these reconfiguration design patterns are helpful to developers implementing a DAS from scratch, their contents are not organized in template format and they do not address safety and assurance concerns. Subsequently, Ramirez and Cheng [85, 88] harvested twelve adaptation-oriented design patterns from more than thirty existing designs of adaptive systems to support the reuse of solutions for recurrent problems in dynamic adaptation. These design patterns support the monitoring, decision-making, and reconfiguration of a DAS. In contrast to the design patterns introduced by Gomaa *et al.*, these design patterns are catalogued and organized using a design pattern template akin to the one developed by Gamma *et al.* [34].

The adaptation-oriented design patterns proposed by Gomaa *et al.* and by Ramirez and Cheng are orthogonal to the model-based framework presented in this dissertation. That is, these design patterns can be leveraged to guide the design and implementation details of a DAS as it relates to specific activities such as monitoring, decision-making, and reconfiguration.

9.2.2 Model-based Frameworks for Dynamically Reconfiguring Adaptive Systems

Research into adaptive software techniques has also focused on designing and developing frameworks for building adaptive systems [11, 35]. A object-oriented framework is a set of cooperating classes that make up a reusable design for a specific class of software [34]. Among other services, a framework dictates the overall architecture of the application and its thread of control. This control thread often leads to an inversion of control where developers write code that gets called by the framework. A major benefit of a framework is that it provides large amounts of reusable code, thereby enabling developers to building applications faster. Nevertheless, to certain degree, creating freedom is lost because many design decisions have already been made by the framework developers [34]. Additionally, framework-based applications are sensitive to changes in the framework's interface.

Separating the adaptive logic from the functional logic often simplifies the development and maintenance of adaptive systems while promoting software reuse. In the same spirit, several model-based architectures have been developed to separate concerns in the design and implementation of a DAS [4, 35, 45, 83]. For instance, Oreizy *et al.* [83] proposed an infrastructure that supported two simultaneous processes in a DAS. While the first process managed the evolution of the system in response to environmental changes, the second process used models to detect changing requirements and environmental conditions and planning responsive modifications. Moreover, other researchers [29, 36, 37] also explored how monitoring and decision-making tasks interact within a DAS, often by exploiting models of the system at design time and at run time. Garlan and Shaw [35, 102] further decomposed the architecture of a DAS by applying control theory approaches. As a result, the most common architecture found in adaptive systems today comprises monitoring, decision-making, and reconfiguration processes.

Although finer-grained models can be used to guide adaptation, software engineering adaptive research focused primarily on using architectural description languages (ADLs) to capture and manage system evolution and system adaptation. Architecture-based approaches for self-adaptive software usually view systems as networks of concurrent components bound together by connectors [83]. In this manner, architectural-based representations of a system shift focus away from source code to coarse-grained components and their interconnections. Within these representations, a component is responsible for providing application functionality and maintaining state information. Connectors, on the other hand, offer transport and routing services for messages and data objects. Thus, in architectural-based approaches, dynamic reconfiguration involves not only adding, removing, and modifying components and their connections, but also managing the evolution of the system and the consistency of the component-connector representations.

Kramer and Magee [65] proposed a three-layer architecture-based model for self-adaptation. The component control layer, which is the lowest layer, is responsible for creating, interconnecting, and deleting components. The change management layer, the middle layer, comprises a predetermined set of reconfiguration plans that can be applied to repair the architecture at run time. Lastly, the goal management layer, the highest layer, is responsible for creating new change management plans as necessary in order to facilitate the overall evolution of the system and its reconfiguration mechanisms.

Other examples of architectural-based approaches for self-adaptive systems include Taylor *et al.*'s C2 [83] and Gorlick's Weave [43]. C2 [83], composes systems as a hierarchy of concurrent components bound together by connectors such that a component within the hierarchy can only be aware of components residing at the same level or beneath it. Weaves, on the other hand, is a dynamic, object-centric architecture targeted towards applications with large volumes of data flow and real-

time constraints [43]. One interesting characteristic of Weaves is that no component in a network knows the sources of its input objects or the destination of its output objects.

Similarly, Garlan *et al.*'s Rainbow [35] is an object-oriented framework that provides reusable code and infrastructure for monitoring and adapting distributed components. Rainbow uses external adaptation mechanisms to avoid intruding upon functional logic and thus facilitate its application to legacy systems. Rainbow's adaptation infrastructure incorporates control theory concepts [10, 102] to monitor and report values to gauges and gauge consumers. These values are then related to properties in an architectural models, where the architecture is analyzed to ensure no constraint is violated. If a constraint is violated, then the architecture is reconfigured by selecting a reconfiguration plan that best addresses monitored conditions and architectural constraints as specified by utility theory-based models.

One significant difference between our model-based framework and these adaptation-oriented frameworks is the level of abstraction at which they operate. That is, our model-based framework uses goal-oriented requirement models to guide the design and dynamic adaptation of a DAS, whereas the other adaptation-oriented frameworks use architectural models. Furthermore, these adaptation-oriented frameworks use a repository of architectural models to determine how the DAS should reconfigure itself at run time. In contrast, our model-based framework can generate safe adaptations at run time to support reconfigurations against unanticipated adaptation scenarios. Ultimately, however, our model-based framework complements these adaptation-oriented frameworks in one important way. Specifically, our model-based framework can automatically explore the space of possible reconfigurations and thus generate architectural models that can then be used to guide the reconfiguration of a DAS.

Fleurey *et al.* [30, 31] developed DiVA, a model-based approach and framework

for constructing and managing a DAS. The primary objective of DiVA was to address the combinatorial explosion of artifacts, such as configurations and adaptation rules, that result from the complexity of a DAS both at design time and at run time. Their approach encodes the parameters and components that a DAS can dynamically reconfigure into sets of variation points. Abstracting possible reconfigurations in this manner enables DiVA to exploit aspect-oriented techniques [59] and thereby represent variability dimensions as aspects that can be woven in and out of the DAS as necessary. As a result, a developer need not consider, at design time, all possible combinations of reconfigurations that a DAS might need to support. At run time, only the selected target configuration needs to be evaluated to ensure it satisfies constraints and domain assumptions.

At run time, DiVA selects reconfigurations by applying utility theory to evaluate all possible configurations that the DAS might reach from its current state and select the optimal one. While DiVA exploits variability points and aspect-oriented techniques to manage the complexity of adaptation policies specified at design time, this approach merely defers this combinatorial complexity until the run time phase. To address this run time optimization problem, DiVA combines both rule-based and optimization-based techniques to provide efficient adaptation validation capabilities. Specifically, DiVA uses both domain-specific modeling languages and optimization techniques, such as value discretization (i.e., qualitative values such as low, medium, or high) to reduce the possible space of adaptations into more manageable sizes that can be evaluated at run time.

Our model-based framework shares a similar set of objectives as with those used by DiVA, yet both achieve these objectives through a complementary set of strategies and techniques. Both frameworks support the on-demand generation of adaptations at run time based on current system and environmental conditions. In this manner, neither approach requires a developer at design time to prescribe specific sets of adap-

tations that the DAS will need to support at run time. Similarly, both approaches are able to efficiently generate adaptations that preserve system consistency before, during, and after reconfiguration. To select optimal adaptations at run time, however, the DiVA framework combines both rule-based and optimization strategies. In contrast, our framework leverages evolutionary algorithms to generate and optimize target system configurations and adaptation paths.

Goldsby *et al.* [39, 41] applied digital evolution [82] to evolve suites of structural and behavioral UML models of adaptive systems. These models specified target system configurations that satisfied functional requirements while balancing trade-offs between non-functional properties. Their approach is similar to our model-based framework in that both apply evolutionary computation techniques to explore broader sets of target system configurations than would normally be considered by a human developer. As with Goldsby's approach, our model-based framework can also be applied at design time to explore the space of possible adaptations and how these satisfy different functional and non-functional properties. Nevertheless, for some application domains, such as remote data mirroring, our framework can also be applied at run time to generate target reconfigurations on-demand in response to current system and environmental conditions. Furthermore, Goldsby *et al.*'s approach does not specify the sequence of reconfiguration instructions required to safely transition the executing system from one target configuration to another. To this end, our model-based framework complements Goldsby's approach by providing the sequence of reconfiguration instructions that can safely transfer the system to its target configuration.

Within their evolutionary computation-based framework, Goldsby *et al.* [40] introduced an approach that applies novelty search to generate properties that specify latent behaviors of a system in UML class and state diagrams. Marple uses novelty search to facilitate the discovery of a set of properties that describe the UML model but are not explicitly stated in the system's requirements. Although our model-based

framework facilitates the discovery of latent behaviors in a DAS, it differs from Marple by focusing on generating operational contexts that violate requirements. Marple, on the other hand, is used to detect (unwanted) latent properties. Furthermore, Marple uses model checking techniques to evaluate models for the existence of latent properties, and Loki uses an executable specification of the system, in the form of a simulation, that can identify system and environmental conditions that produce latent behaviors and requirements violations.

To adapt the DAS, these frameworks often leverage the dynamic change management (DCM) protocol introduced by Kramer and Magee [64]. As stated in Chapter 2, in the DCM protocol, components can reach active, passive, and quiescent states. In an active state, a component can service and initiate transaction requests with other components. In contrast, in a passive state, a component can service but not initiate transaction requests with other components. A quiescent component, on the other hand, can neither service nor initiate transaction requests with other components. Kramer and Magee identify quiescence as a requirement for preserving system consistency during adaptation. However, this requirement may result in significant degradation of system performance because components not involved in an adaptation may need to temporarily reach passive states. Although our framework leverages the dynamic change management protocol established by Kramer and Magee, Hermes supports the reordering of those reconfiguration instructions to minimize service disruption while preserving the safety of the adaptation path.

Vandewoude *et al.* [108] introduced the concept of *tranquility* as a weaker but sufficient condition for preserving system consistency during adaptation. In contrast to quiescence, tranquility does not require neighboring components to reach passive states before a component undergoes a reconfiguration. Specifically, only components involved in a reconfiguration must reach passive states, thus providing tranquility an advantage of being less disruptive than quiescence. However, Vandewoude *et al.* also

showed that reaching tranquility in a bounded time is not guaranteed. For instance, a component undergoing adaptation may receive requests from active neighboring components at any time. As such, reaching tranquility is considerably influenced not only by the order in which requests from neighboring components arrive, but also by whether those requests overlap or not. Although experimental evaluations conducted by Vandewoude *et al.* showed this scenario to be rare, if tranquility is not reachable in bounded time, then the system must regress to a quiescent state. In this manner, our model-based framework provides a complementary approach for safely reconfiguring a DAS while minimizing system disruption in application domains where tranquility may be difficult to reach in bounded time. Specifically, Hermes reorders the sequence in which reconfiguration instructions are applied in order to minimize the time required to reconfigure a DAS. In contrast to Vandewoude's approach, Hermes still guides components towards quiescent states to preserve system consistency during adaptation.

Zhang *et al.* [121] further extended Kramer and Magee's approach by generating graphs of all possible safe adaptations. Each edge in the safe adaptation graph encodes a safe reconfiguration step, and each reconfiguration step in the graph is associated with a relative cost that measures the approximate time required for the reconfiguration step to complete. Once the safe adaptation graph is generated, Zhang *et al.* apply Dijkstra's algorithm [22] to search for a safe reconfiguration that minimizes system disruption. While their approach guarantees globally optimal solutions, the complexity of the algorithm is exponential with respect to the number of components involved in the reconfiguration, which may be impractical if the reconfiguration involves many components. In contrast to the safe adaptation graph approach introduced by Zhang *et al.*, our framework applies evolutionary computation to efficiently generate adaptation paths that preserve system consistency. As our experimental results suggest, Hermes can exploit more opportunities when balancing multiple con-

cerns as the complexity of a reconfiguration increases.

Chapter 10

Conclusions & Future Investigations

Increasingly, software systems need to self-adapt in response to changes in their requirements and execution environment. The self-adaptive and software engineering communities have proposed numerous techniques, approaches, and processes for rigorously designing, implementing, and validating DASs [10, 35, 53, 65, 73, 78, 83, 88, 119, 120]. A few of these techniques and methods have attempted to address assurance issues surrounding how a DAS satisfies its requirements before, during, and after a reconfiguration [38, 40, 64, 119, 120, 121, 122].

An underlying and recurrent motivation throughout this dissertation is the realization that it is often impossible for a human to fully anticipate and understand the effects of all operational contexts that a DAS can encounter. Based on this assumption, this dissertation presented a model-based framework that supports the specification, monitoring, and dynamic reconfiguration of a DAS in order to address sources of system and environmental uncertainty. In particular, our model-based framework comprises five key processes, each of which uses a goal model as a reference point to link adaptation-specific decisions with the requirements they are intended to support

or satisfy. By using goal models as reference points for adaptation, our framework can be leveraged at any point from requirements engineering all the way to run time time.

In our model-based framework, three design-time components automate the discovery and analysis of sources of uncertainty such that a DAS can be hardened to handle their occurrence at run time. Specifically, Athena supports the generation of utility functions that can monitor how system and environmental conditions affect requirements satisfaction at run time. Loki, on the other hand, leverages these utility functions and exposes a DAS to a wide range of operational contexts to discover interesting behaviors that might include requirements violations and latent behaviors. Lastly, AutoRELAX leverages sources of uncertainty discovered by Loki to refine the utility functions derived by Athena such that a DAS can better tolerate certain sources of uncertainty that might not necessarily require an immediate adaptation.

Our model-based framework also recognizes that certain adaptation decisions must be deferred until run time when actual system and environmental conditions can be measured and analyzed. Specifically, even with automated techniques such as Athena, Loki, and AutoRELAX, it is unlikely that a DAS will be fully capable of properly interpreting and handling all operational contexts that it encounters at run time. To this end, two components in our model-based framework, Plato and Hermes, support the on-demand generation of target system reconfigurations and safe adaptation paths. Overall, these key processes should produce more resilient adaptive systems that can better handle uncertainty at run time.

We demonstrated our model-based framework by applying each of its components to an industrial case study that involves the run-time reconfiguration of a remote data mirroring network. Moreover, this end-to-end case study demonstrated how both goal models and evolutionary algorithms can directly support the identification and mitigation of sources of uncertainty at both design time and at run time.

10.1 Summary of Contributions

There were three major objectives for this research:

1. To use lightweight, computationally inexpensive techniques to assess the environment and its impact on system functionality.

2. To maximize the automation for the overall process of specifying, monitoring, and adapting a DAS.

3. To minimize the need to identify a predetermined set of adaptations in response to anticipated reconfiguration scenarios at design time.

Guided by these research objectives, this dissertation makes the following research contributions:

- Defines an overarching model-based framework for specifying, monitoring, and dynamically reconfiguring a DAS that explicitly accounts for the effects of system and environmental uncertainty [72].

- Automatically generates and fine-tunes utility functions for requirements monitoring in a DAS. These utility functions enable a DAS to detect requirements violations and other conditions that either lead to unsatisfied goals or warrant adaptation. In addition, these utility functions explicitly account for the effects of environmental uncertainty in order to reduce false adaptation positives [87, 89, 91].

- Automatically explores and evaluates different combinations of system and environmental conditions that a DAS may encounter throughout its lifetime. This analysis can suggest alternate design choices for resolving obstacles at the requirements level [92].

- Automatically generates adaptations that specify the target system configuration, as well as the series of reconfiguration steps required to safely reach that target configuration from the system's current configuration. These adaptations can be generated by only specifying, at a high-level of abstraction, the general objectives that the DAS should satisfy [86, 90, 93, 94].

10.2 Future Investigations

Several research areas can be explored based on the results presented in this dissertation. In particular, several complementary investigations can be pursued to extend our model-based framework for specifying, monitoring, and dynamically adapting a software system to explicitly address system and environmental uncertainty. We now describe each of these future investigations and relate them to our model-based framework.

Automatic Updates of Utility Functions. Athena currently generates utility function templates for requirements monitoring in a DAS. Although Athena supports partial code generation, these utility functions must ultimately be implemented within the adaptation logic of the DAS. As such, two possible future investigations include: (1) further developing Athena such that it generates and, if applicable, compiles utility functions into application code that can be linked or loaded into a DAS at run time, and (2) extend our model-based framework such that changes to the goal model at run time causes Athena to regenerate utility functions for the affected portion of the goal model. Combined, these two extensions would enable a DAS to regenerate and reload utility functions for requirements monitoring at run time in response to changes in the system's requirements. Moreover, this research line might require introducing new modeling and implementation techniques for introducing and maintaining traceability links between the goal model and its utility functions.

Visualization Techniques. Another interesting line of research based on Athena includes developing tools and techniques for visualizing the satisfaction of functional, non-functional, and RELAXed goals in real time. Specifically, our model-based framework treats goal satisfaction as adaptation triggers that can automatically cause a DAS to self-reconfigure in response to adverse system and environmental conditions. However, some adaptive systems might require humans in the loop to make high-level adaptation decisions. In these systems, humans might benefit from tools and strategies that quickly depict the degree to which a DAS is satisfying its requirements without having to manually inspect goal satisficement values.

Interpreting Loki Conditions. The novelty archive produced by Loki can also serve as an additional information repository that can be leveraged by a DAS at run time to detect when it is executing either in an unanticipated or adverse operational context. Specifically, the DAS could attempt to map current system and environmental conditions to operational contexts stored in its novelty archive. If the DAS successfully matches these conditions to an existing operational context, then it can evaluate and anticipate how well it will be able to satisfy its requirements. However, if the DAS cannot match current conditions to an existing operational context, then it might need to trigger possible adaptations to either collect additional monitoring information and diagnose the effects of its operational context or switch to a safe or protected operational mode until it reaches a more desirable operational context.

Loki Extensions. It may also be possible to apply Loki at different levels of abstraction. For instance, Loki could be extended and applied at the application-code level to discover how operational contexts exercise the application and adaptive logic at run time. To this end, the main novelty search mechanism used by Loki can be directly reused such that it searches for distinct behaviors at the code level as captured by logging statements generated as the DAS executes. Identifying novel

behaviors in this manner, and tracing them to specific code segments, might suggest specific functions or classes that require revisions. Moreover, it may be possible to transparently introduce these logging statements via aspect-oriented programming techniques (AOP) [59, 60], thereby separating Loki from core functionality.

Automatic Generation of Fitness Weighting Schemes. AutoRELAX uses fitness sub-function coefficients to balance competing concerns between minimizing the number of RELAXed goals and minimizing the number of adaptations triggered. Currently, these fitness sub-function coefficients are static and manually derived by a requirements engineer. As such, another future research direction might explore how different optimization algorithms, including evolutionary algorithms such as the stepwise adaptation of weights (SAW) technique [24], might be applied to automatically balance concerns as captured by these fitness sub-function coefficients. This extension would enable requirements engineer to reuse AutoRELAX as a black-box technique without having to empirically determine how fitness sub-function coefficients should be configured.

Decentralizing Adaptive Logic Generation. Both Plato and Hermes are currently implemented as a collection of centralized components and algorithms in our model-based framework. Unfortunately, it can be expensive for centralized components to gather, analyze, and process all monitoring information to determine if a reconfiguration is warranted at run time. Furthermore, centralized implementations of Plato and Hermes can also become single points of failure that prevent a DAS from self-reconfiguring at run time. Based on this reasoning, another interesting line of research could explore the feasibility of executing Plato and Hermes in a distributed manner. Dividing the responsibilities of Plato and Hermes among various independent DAS components might improve both fault tolerance as well as adaptation performance.

Extending Monitoring Capabilities. Currently, Hermes uses static values to

evaluate the effects of a safe adaptation path at run time. These values are determined empirically by an adaptation engineer using our model-based framework and reflect *anticipated* adaptation costs. It might be interesting to explore how the monitoring infrastructure of a DAS can be leveraged to observe not only system and environmental properties, but also actual adaptation costs at run time. In this manner, a DAS could use its monitoring infrastructure to refine adaptation costs at run time based on actual execution data. Refining anticipated adaptation cost values at run time could improve the quality of adaptation paths generated by Hermes as it can evaluate its effects more accurately and precisely.

Lastly, it would be interesting to explore how our model-based framework can be extended to leverage tests at run time in order to detect whether a DAS satisfies, or is even capable of satisfying, its requirements. One possible direction recently proposed by Fredericks, Ramirez, and Cheng [32] extends the traditional monitoring, analysis, planning, and execution (MAPE-K) architecture loop for adaptive and autonomic systems such that it focuses on testing activities. Specifically, within a MAPE-T architecture, a DAS would continuously monitor the satisfaction of test cases and how the satisfaction of these relate to specific requirements. Furthermore, the MAPE-T architecture would also evaluate whether test cases are applicable under the current operational context of the DAS and, if necessary, adapt their expected inputs and outputs to reflect current system and environmental conditions. In addition, the MAPE-T architecture would also determine when tests could be safely run at run time without adversely affecting the DAS.

APPENDICES

Appendix A

Intelligent Vehicle System Case Studies

This appendix presents additional experimental results that illustrate how techniques in our model-based framework can be applied to an intelligent vehicle system (IVS) application. First, we overview the IVS application domain and state the various goals and constraints that it must satisfy. Next, we present and describe a goal model that captures the requirements of the adaptive cruise control and lane keeping features of the IVS. When then present experimental results obtained by leveraging the utility functions derived by Athena to support requirements monitoring in the IVS under different simulation scenarios. Lastly, we present experimental results obtained by applying Loki to the IVS executable specification when exploring sources of uncertainty.

A.1 Intelligent Vehicle Systems

Intelligent transportation systems (ITS) are envisioned to provide safe and efficient transportation of passengers across roadways. Within the ITS domain, an intelligent vehicle system (IVS) provides a human driver with the convenience of au-

tonomous vehicle control through a combination of adaptive cruise control (ACC), lane keeping, and collision avoidance features. As Figure A.1 illustrates, the ACC module is responsible for maintaining a *SafeDistance* between the IVS and any obstacles in front of the IVS, such as a Lead Vehicle. To this end, the ACC module commands the vehicle's engine to achieve and maintain either a *DesiredSpeed* or a *SafeSpeed*, depending on the presence of any obstacles, and thus keep the IVS within the *CoastingZone*. The lane keeping module, on the other hand, detects roadway markings and commands the vehicle's steering mechanisms to maintain the IVS within the center of the lane. Lastly, the collision avoidance module uses a set of cameras and distance sensors to detect nearby obstacles and adjust the vehicle's engine and steering mechanisms in response.

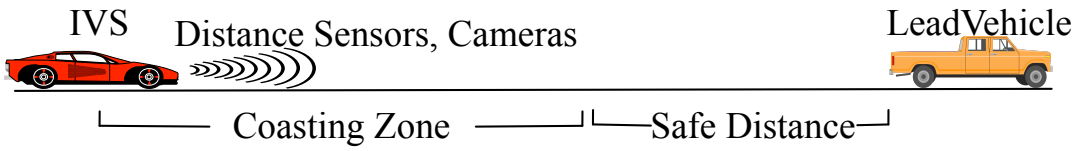


Figure A.1: Overview of an Intelligent Vehicle System.

The Webots simulation platform [74] provides a generic implementation of an IVS vehicle that is capable of cruise control and lane keeping features. This generic IVS model comprises a single GPS unit for computing the vehicle's position and velocity, two cameras for detecting roadway markings, and a single accelerometer to compute acceleration and deceleration rates. For this dissertation research, we extended the basic Webots IVS implementation with a monitoring infrastructure that supports ACC, lane keeping, and collision avoidance features. In particular, we extended the basic Webots IVS to also include a single compass and gyroscope for computing changes in the vehicle's heading and velocity, two additional cameras to detect roadway markings and nearby obstacles, and ten laser- and sonar-based distance sensors that measure the distance between the IVS and any nearby obstacles. For the remainder of this dissertation, the term IVS refers to the extended IVS implementation.

A.2 IVS Goal Model

Figure A.2 presents the first part of a KAOS goal model for the IVS application. Specifically, this part of the goal model presents the adaptive cruise control (ACC) feature of the IVS. This goal model captures the set of goals responsible for computing the IVS's current speed and its distance to nearby vehicles in front of the IVS. As this goal model illustrates, the IVS can use either a GPS unit or a set of wheel sensors to measure and compute its current speed. Likewise, the IVS can use a set of cameras and distance sensors to detect and compute the distance to a nearby obstacle, respectively. These alternative goal refinements enable the IVS to adapt how it senses its environment in response to different system and environmental conditions, such as sensor failures.

Figure A.3 presents the second part of the KAOS goal model for the IVS application. In particular, this goal model captures the objectives and constraints of the lane keeping feature. Note that goal (A) refers to the same goal as in Figure A.2. This goal model captures goal specifications for computing and adjusting the position of the IVS relative to the center of its lane. To achieve these objectives, the IVS uses a set of front-facing cameras to detect road markings and compute any necessary corrections to the vehicle's current steering angle. Specifically, the IVS uses its *L/R Steers* to adjust its heading and or direction according to these steering angle corrections, thereby maintaining the IVS within its lane boundaries. Lastly, this goal model also captures a non-functional goal that specifies that the IVS should minimize its acceleration and deceleration rates in order to assure passenger comfort.

To capture the interactions across the shared boundary between system and environmental agents, we applied the *unmonitorability* refinement pattern [19] to goals (I, L, M), (J, N, O), (K, P, Q), and (T, V, W). This refinement pattern is applicable whenever a system agent is unable to directly measure an environmental property specified in a goal formulation. For instance, the ENV property *VehicleSpeed* in goal

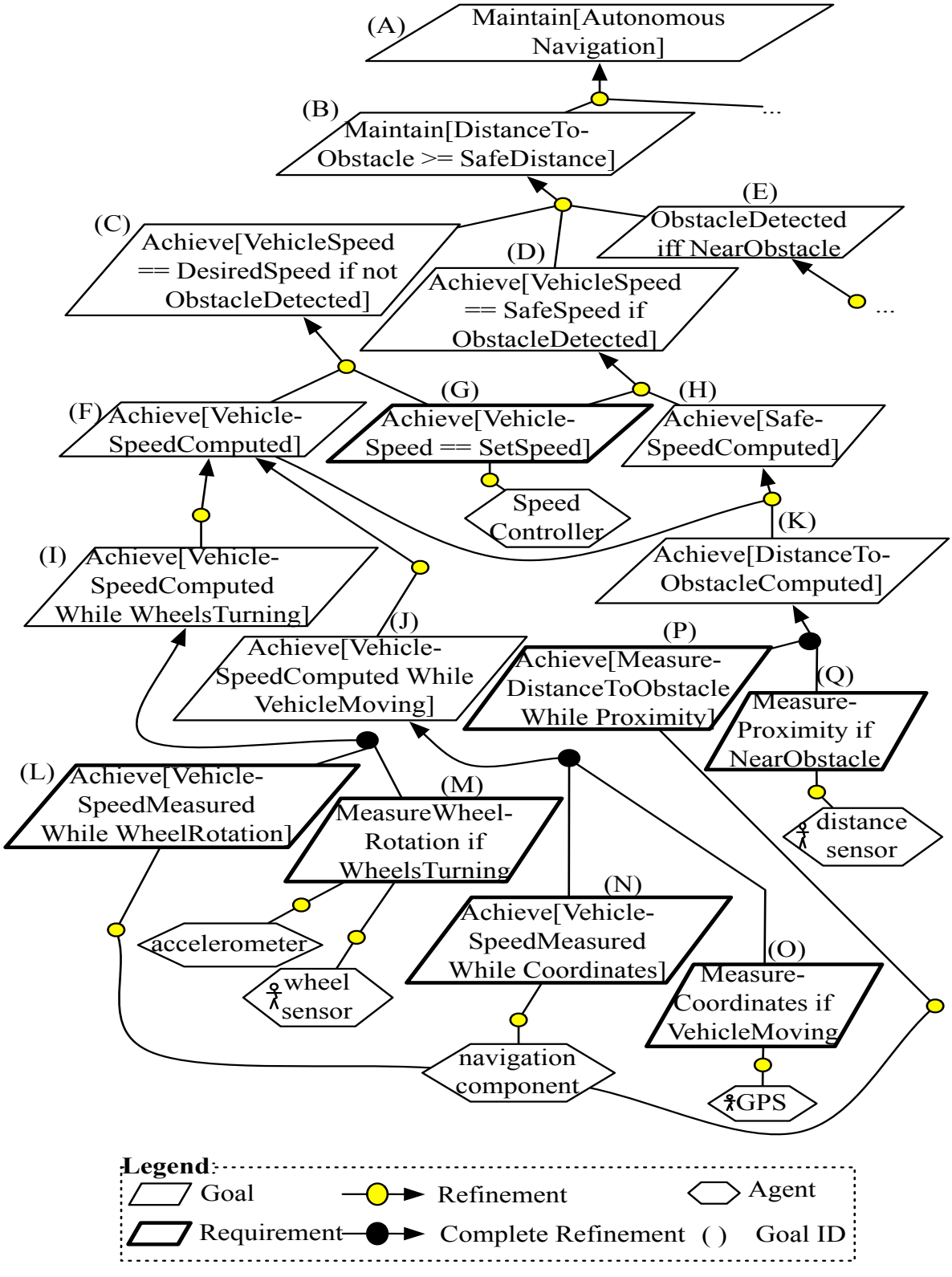


Figure A.2: KAOS goal model for adaptive cruise control in IVS.

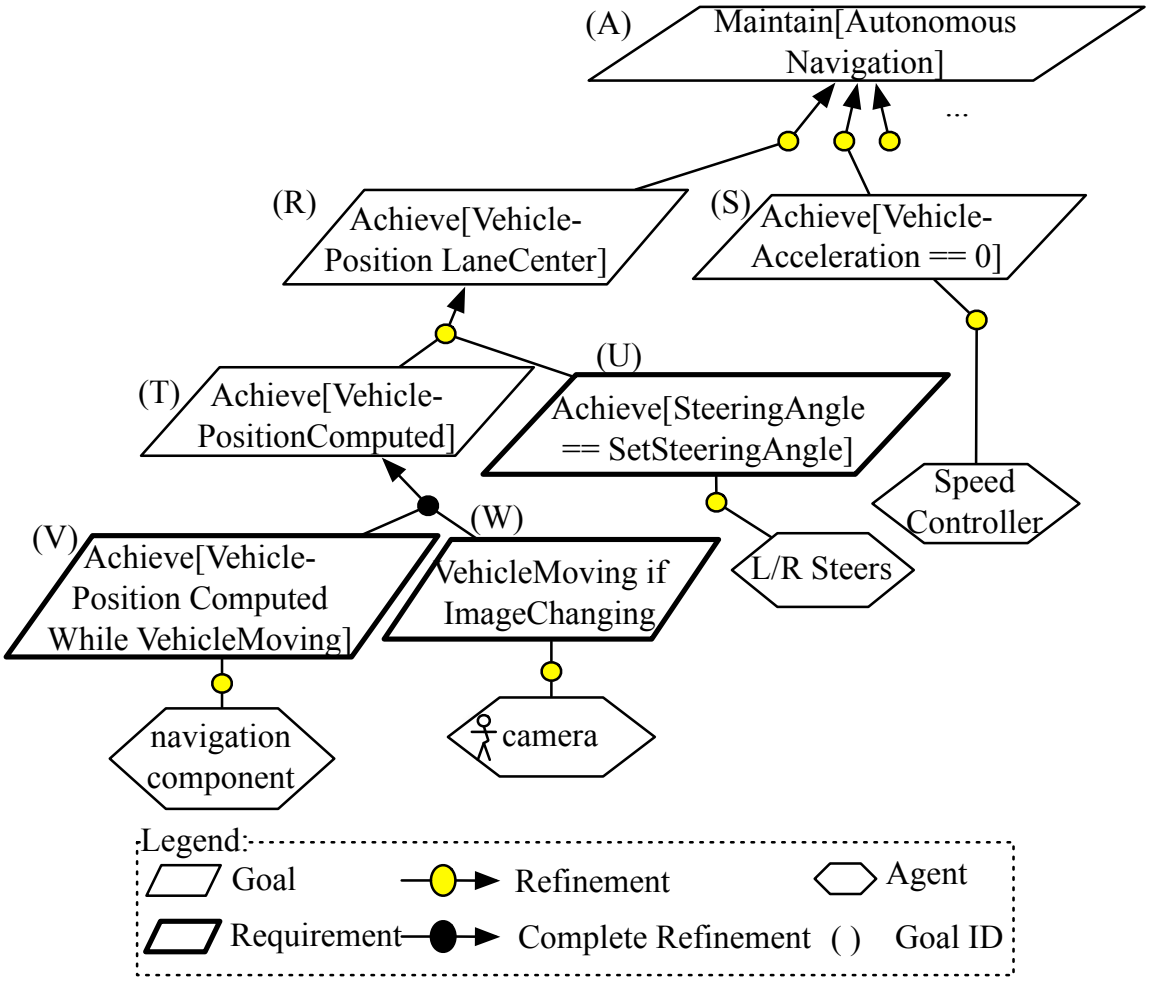


Figure A.3: KAOS Goal model for lane keeping feature in IVS.

(I) cannot be directly measured by system agents. Instead, we applied this refinement pattern to reassign the responsibility of measuring that environmental property to an environmental agent capable of doing so, *WheelSensor*. The measurements by this environmental agent are then processed by the *NavComputer* system agent.

Combined, the goal models in Figure A.2 and A.3 capture the adaptive cruise control and lane keeping features that the IVS must support, respectively. Note that several goals have been RELAXed in these goal models to explicitly address uncertainty. For instance, as these goal models illustrate, the IVS can tolerate minor deviations from its center lane without necessarily causing a collision with another vehicle. Likewise, the IVS can also tolerate minor deviations between its measured

speed and its desired target speed, as long as no obstacles are detected. Lastly, these goal models also capture non-functional goals that specify that the IVS should minimize its acceleration and deceleration rates through its *SpeedController* actuator.

A.3 Derived Utility Functions for Requirements Monitoring

In this section, we apply Athena to the IVS goal model in order to derive utility functions for requirements monitoring. First, we present a subset of the utility functions derived by Athena for monitoring invariant, non-invariant, and RELAXed goals. Next, we present results from applying these utility functions to monitor how well the IVS satisfies its requirements under different simulation scenarios.

A.3.1 Sample Derived Utility Functions

As previously shown in Chapter 3, Athena accepts as input a set of ENV properties, MON elements, and REL relationships for every RELAXed goal in the KAOS goal model. Table A.1 presents a subset of these elements that are relevant to the ACC feature in the IVS goal model. For instance, Row (1) specifies that inGgoal (M), the ENV property WheelRotation can be directly measured by the MON element WheelSensor. Rows where no MON elements are specified are instances of the unmonitorability refinement pattern and explicitly denote that ENV property cannot be directly observed by MON elements. For instance, Row (2) specifies that in Goal (L) the ENV property VehicleSpeed cannot be directly measured by a MON element, and must instead be computed by the corresponding REL relationship.

Overall, Athena derived a total of 11 utility functions for requirements monitoring based on the IVS goal model. Next, we present a subset of these utility functions.

State-based Utility Functions. Figure A.4 presents the state-based utility

Table A.1: Table with ENV, MON, and REL elements for IVS application.

Row	Goal	MON	REL
1	M	WheelSensor	WheelSensor.value
2	L		VehicleSpeed = (IVS.wheel_diameter * 3.14 * WheelRotation)
3	O	GPS	Coordinates = GPS.value
4	N		VehicleSpeed = (NavigationComponent.prev_pos - Coordinates) / GPS.time_unit
5	Q	DistanceSensor	Proximity = DistanceSensor.value
6	P		DistanceToObstacle = (Proximity * DistanceSensor.max_range)
7	H	WheelSensor, GPS	SafeSpeed = VehicleSpeed - 0.1 * VehicleSpeed * (1.0 - SafeDistance / DistaneToObstacle)
8	E	DistanceSensor	ObstacleDetected = Proximity > 0.95
9	B	WheelSensor, GPS	SafeDistance = 2.5 * VehicleSpeed * 1000 / 3600

function that Athena derived for monitoring the satisfaction of Invariant Goal (B), which states that the IVS should maintain a safe distance between itself and nearby obstacles in front. This utility function is based on the template shown in Figure 3.9. In particular, this utility function first computes the distance to an obstacle by measuring the values that the IVS's distance sensors produce. Next, this utility function verifies whether this invariant goal has been previously violated (the satisfied variable stores this history). If the IVS has satisfied this goal, then it computes whether it is currently maintaining a distance that is at least as great as the target_distance threshold. Otherwise, this utility function returns false.

Continuing with the IVS ACC goal model in Figure A.2, Goal (D) is also an invariant goal that refers to both the ENV property `VehicleSpeed` as well as the relative constraint `SafeSpeed`. Since this goal refers to the real-valued ENV property `VehicleSpeed`, Athena generates a metric-based utility function for this invariant goal even though its satisfaction can also be assessed in a crisp fashion. Specifically, Athena measures the satisficement of this goal by deriving a metric-based utility function that measures the degree to which the IVS minimizes the difference between `VehicleSpeed`

```
// Derived Utility Function for Goal B (Safe Distance)
boolean maintainSafeSpeedStateFunction(double proximity,
      String operator, double target_distance) {

   // calculate distance to obstacle:
   double distance_to_obstacle = proximity *
      IVS.Distance_Sensor.max_range;

   // return utility value:
   if(operator.equals("<=")) {
      if(!satisfied) {
         return !(satisfied = distance_to_obstacle
            <= target_distance);
      } else {
         return false;
      }
   } else (...)
}
```

Figure A.4: State-based utility function derived by Athena for goal (B)

and `SafeSpeed`. In this manner, Athena instantiates function templates (3.1) as follows:

$$\mathrm{UT}_{\mathrm{SafeSpeed}} = 1 - \min\ \{\ \frac{|\mathrm{VehicleSpeed} - \mathrm{SafeSpeed}|}{\mathrm{VehicleSpeed}},\ 1\ \} \tag{A.1}$$

This utility function produces a utility value that is inversely proportional to the difference between `VehicleSpeed` and `SafeSpeed`, and may be used to detect conditions conducive to a requirements violation. For instance, a significant drop in this utility value may suggest the IVS is exceeding its `SafeSpeed` constraint, possibly leading to a collision with a nearby obstacle.

Metric-based Utility Functions. Figure A.5 presents the metric-based utility function that Athena derived for non-invariant goal (C), which states that the IVS should achieve and maintain a desired speed. To derive this utility function, Athena instantiated utility function (3.1) with the ENV to MON mappings previously shown in Table A.1. Specifically, this utility function first computes the current speed of the

IVS by using the wheel rotation values that the IVS's `Wheel Sensors` produce. Next, this utility function computes a value proportional to how close the IVS's `Vehicle Speed` is to its `Desired Speed`.

```
// Derived Utility Function for Goal C (Vehicle Speed)
   double desiredSpeedUtilityFunction(double wheel_rotation,
      double desired_speed) {

      // calculate current speed:
      double current_speed = IVS.wheel_diameter * 3.1415 *
            wheel_rotation;
      // return utility value:
      return Math.min(Math.abs(current_speed -
            desired_speed) / current_speed, 1.0);
}
```

Figure A.5: Metric-based utility function derived by Athena for goal (C)

This utility function produces a utility value that is inversely proportional to the difference between `VehicleSpeed` and `DesiredSpeed`. Thus, the primary objective of this utility function is to assess whether the IVS is able to achieve and maintain a desired target speed while no obstacles are detected nearby.

RELAXed Utility Functions. Lastly, Goal (R) in the IVS lane keeping goal model (see Figure A.3) has been RELAXed by applying the "AS CLOSE AS POSSIBLE TO" RELAX operator. As a result, to evaluate the satisficement of goal (R) at run time, Athena applies the triangle-shaped utility function presented in Figure 3.6. Athena instantiates this utility function template by assigning the ENV property `Vehicle Position` to the *Measured Quantity* variable, and the absolute threshold `Lane Center` assigned to the *Desired Quantity* constraint. This utility function measures the satisfaction of RELAXed Goal (R) by explicitly capturing the extent to which the IVS can tolerate temporary deviations in its positioning within the center of the driving lane.

A.3.2 Simulation Results

The following experiments were conducted by executing the IVS model and its controller on the Webots simulation platform [74]. Each simulation executes for 2000 time steps, where each time step is 32 milliseconds of simulated time. At the end of each time step, Webots pauses the simulation, executes one iteration of the controller code, and then continues with the simulation process. As the controller executes, the IVS first probes its set of sensors and then uses this monitoring information to recompute its set of utility values. We performed 30 simulation trials for each scenario that involves environmental noise to establish statistical significance and plot mean values.

Normal Behavior. In this simulation, the IVS is initially placed 400 meters behind the Lead Vehicle in between two driving lanes. Initially, the IVS and the Lead Vehicle are set to achieve and maintain desired target speeds of $65km/h$ and $45km/h$, respectively. As a result, in this scenario the IVS first attempts to satisfy Goal (C) in Figure A.2 by achieving its desired target speed while simultaneously satisfying Goal (R) in Figure A.3 by steering the vehicle towards the center of its lane. However, before the IVS can achieve its desired target speed, it detects the presence of the Lead Vehicle and must therefore adapt its behavior to satisfy Goal (D) in Figure A.2 by achieving and maintaining a safe target speed. After 1200 simulation time steps elapse, the Lead Vehicle sets its desired target speed to $75km/h$. Therefore, the IVS is able to once more achieve and maintain its desired target speed once the Lead Vehicle moves away from its sensors.

Figure A.6 plots the utility values produced by the utility functions that Athena derived for Goal (B) in Figure A.2, which specifies that the IVS should maintain a safe distance between itself and nearby obstacles. Since this goal is an *invariant*, Athena generated a state-based utility function to evaluate whether the goal has always been satisfied throughout the simulation. In addition, since both the ENV property

and constraint referred by this invariant goal are floating-point values, Athena also generated a metric-based utility function to determine the degree to which the IVS satisfices [16] that goal. As this figure illustrates, the metric-based utility function suggests at time step 600 the IVS is approaching the Lead Vehicle. As such, the IVS begins to decelerate until it achieves a safe target speed. The IVS maintained this safe speed until its distance to the Lead Vehicle increased, as the plot shows around time step 800. Moreover, as this figure illustrates, the IVS did not violate Goal (B) as its state-based safe distance metric remained at a value of one (true) throughout the entire simulation.

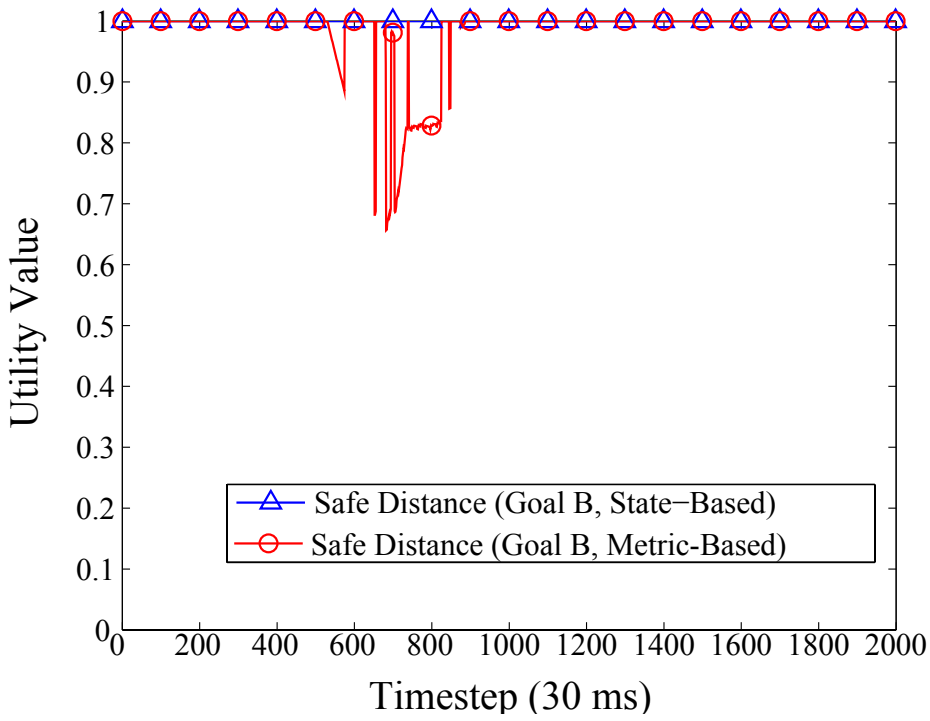


Figure A.6: Utility values for "maintain safe distance" goal.

Figure A.7 plots the metric-based utility functions for Goals (C) and (D) in Figure A.2, which specify that the IVS should achieve and maintain either a desired (denoted by hollow circles) or safe target speed (denoted by triangles) depending on the presence of an obstacle, respectively. From time steps 0 to 150, the IVS begins to accelerate in order to achieve and maintain its desired target speed, eventually

reaching its target speed at approximately time step 500. At this time, the IVS also detects the presence of the Lead Vehicle and adapts its behavior in order to achieve its safe target speed. This figure also captures the tradeoffs between these two goals, as the IVS decelerates from time step 550 to time step 1100 until it achieves its safe speed. Thus, the satisfaction of Goal (D) gradually increases as the satisfaction of Goal (C) decreases. Once the IVS no longer detects the Lead Vehicle, it begins to accelerate once more until it achieves its safe target speed, at which point both Goals (C) and (D) are considered to be satisfied.

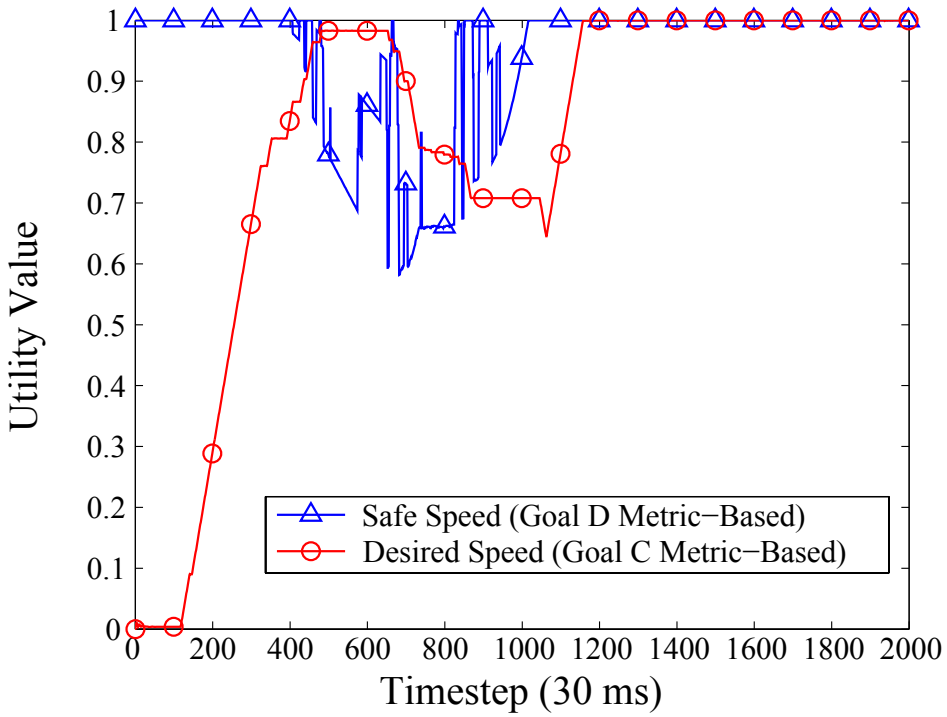


Figure A.7: Utility values for achieve safe and desired speed goals.

Throughout this simulation, the IVS maintained an acceptable acceleration and deceleration rate. Figure A.8 plots the utility values produced by the utility function that Athena derived for measuring the satisficement of non-functional Goal (S) from the goal model in Figure A.3. The IVS maximized the satisficement of this goal as its utility value never dropped beneath a value of 0.85. In addition, this plot also illustrates how this utility function captures the behaviors of the IVS in response to

the detection of a nearby vehicle. That is, whenever the IVS accelerated or decelerated, such as to achieve a desired or safe target speed, the satisficement of this non-functional goal dropped.

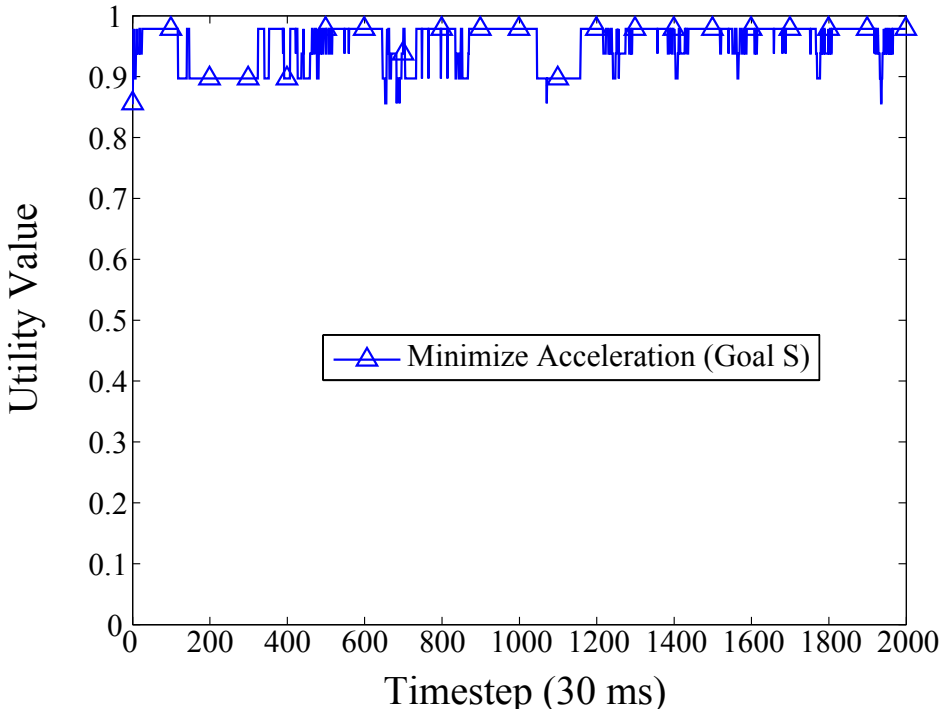


Figure A.8: Utility values for "minimizing acceleration and deceleration rates".

Lastly, Figure A.9 plots the utility values produced by the utility function that Athena derived for measuring the satisfaction of goal (R) ("`maintain center lane`", see Figure A.3). As this figure illustrates, the IVS is initially placed between two lanes. As a result, the initial satisfaction of this goal is approximately 0.15. By time step 200, the IVS manages to steer towards the center of its lane. Thereafter, the IVS maintained the center of that lane until the simulation completed.

Stopped Lead Vehicle. This next simulation is similar to the previous one, with one key difference. Specifically, both the IVS and the Lead Vehicle begin to execute as in the previous simulation. However, instead of the Lead Vehicle slowing down to a target velocity of 45 km/h, it now stops in the highway, in the middle of the two

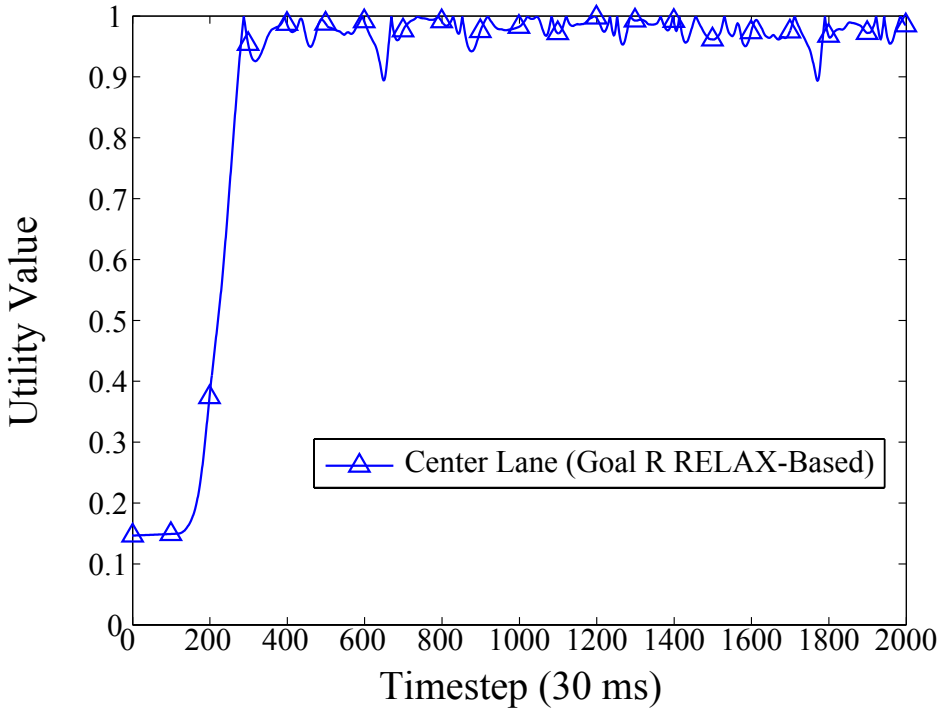


Figure A.9: Utility values for "achieve and maintain center lane goal".

driving lanes, thereby obstructing part of the IVS's center lane. As a result, the IVS must swerve to the side to avoid a collision with the stopped Lead Vehicle and then return to the center of the lane.

As Figure A.10 shows, the IVS successfully maintained a safe distance between itself and the Lead Vehicle throughout the entire simulation. In particular, both the state-based and metric-based utility functions associated with Goal (B) in Figure A.2 reported utility values of 1.0 at each time step. Although the Lead Vehicle partially obstructs the edge of the IVS's lane in this simulation scenario, it does not actually block the lane to the extent where it would cause a collision with the IVS. This can be captured by the IVS's sensors, as the metric-based utility function does not suggest a dip in utility values.

Figure A.11 plots values produced by the utility function responsible for measuring the satisficement of Goal (C) in Figure A.2, which states that the IVS should

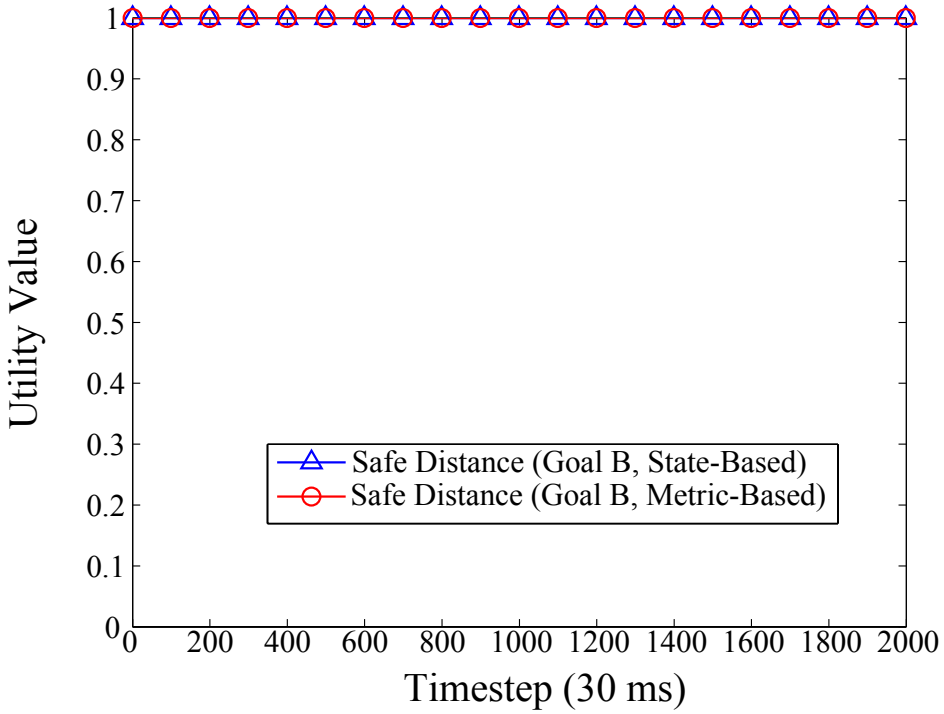


Figure A.10: Utility values for "maintain safe distance" goal.

achieve and maintain a target desired speed. Given that the Lead Vehicle exits the lane where the IVS is driving, the IVS continues to accelerate until it achieves its target desired speed. Since its forward-bearing distance sensors do not detect the presence of the Lead Vehicle, it maintains this velocity throughout the remainder of the simulation.

Figure A.12 plots values produced by the utility function responsible for measuring the satisficement of Goal (S) in Figure A.3, which states that the IVS should minimize acceleration and deceleration rates. This utility function registers a sharp drop in value at approximately time step 600, when the IVS passes the Lead Vehicle. Specifically, at this point in the simulation, one of the left-most distance sensors in the IVS, (LL_DS) registers part of the Lead Vehicle in the IVS's driving lane. Thus, to prevent a collision, the IVS readjusts its steering mechanisms to move the vehicle towards the right portion of the lane. In this manner, this slightly abrupt change in

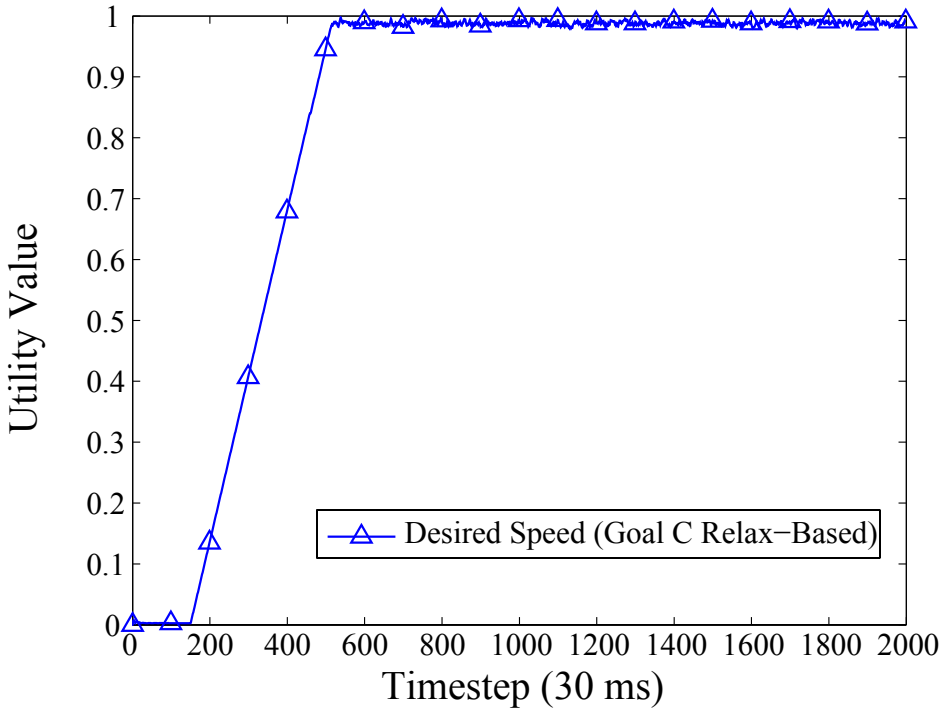


Figure A.11: Utility values for "achieve and maintain desired speed goals".

the IVS's heading is explicitly captured by this utility function.

Lastly, Figure A.13 plots values produced by the utility function responsible for measuring the satisficement of goal (R) in Figure A.3, which states that the IVS should achieve and maintain the center of its driving lane. As with the previous simulation, this utility plot captures how the IVS achieves and maintains the center of its driving lane by time step 400. This plot shows a moderate drop in the utility values produced by this utility function at approximately time step 60, when the IVS slightly swerves to the right in order to avoid a collision with the Lead Vehicle. Thereafter, the IVS re-achieves and maintains the center of its driving lane.

Requirements Violation Produced by Environmental Uncertainty. This next simulation presents a scenario that is similar to the first simulation. Specifically, the IVS is initially placed 400 meters behind the Lead Vehicle, in between two driving lanes. As with the first simulation, both vehicles will attempt to achieve and maintain

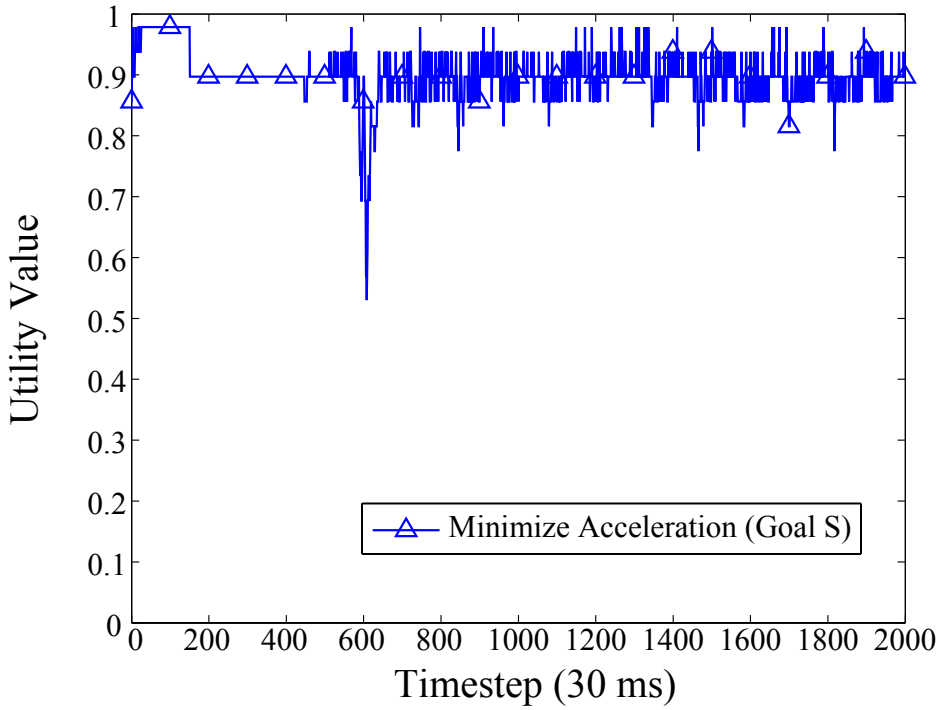


Figure A.12: Utility values for "minimizing acceleration and deceleration rates".

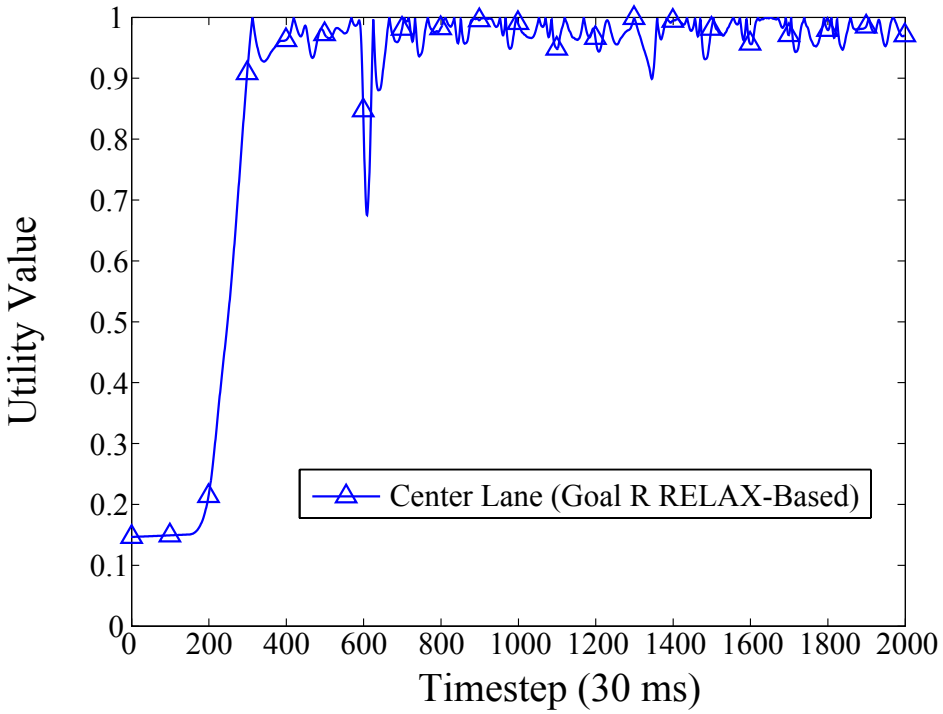


Figure A.13: Utility values for "achieving and maintaining center lane goal".

a desired target speed as well as steer towards the center of the right-most lane. In contrast to the previous simulations, however, the IVS's monitoring infrastructure is now exposed to different severities of static noise (see Table A.2). This uncertainty prevents the IVS from accurately measuring properties about its environment, such as the distance between itself and the Lead Vehicle. As a result of this uncertainty, the measurements and calculations performed by the IVS are unreliable and ultimately produce a requirements violation, in the form of a collision with the Lead Vehicle.

Table A.2: Severity of noise applied to monitoring infrastructure in the IVS

Sensor	Value
Accelerometer	0.05
Camera (Top-right)	0.15
Camera (Top-left)	0.15
Camera (Bottom-right)	0.15
Camera (Bottom-left)	0.15
Compass	0.1
Distance Sensor (Top-right)	0.2
Distance Sensor (Top-left)	0.2
DIstance Sensor (Right)	0.15
Distance Sensor (Mid-right)	0.15
Distance Sensor (Mid-left)	0.15
Distance Sensor (Left)	0.15
GPS	0.005
Gyroscope	0.05
Wheel Sensor	0.15

Figure A.14 plots the utility values produced by the two utility functions responsible for monitoring the satisfaction of Goal (B) in Figure A.2, which states that the IVS should maintain a safe distance to nearby obstacles. As the state-based utility function captures, the IVS failed to maintain a safe distance between itself and the Lead Vehicle. Specifically, the utility values produced by this state-based function changed from true to false at approximately time step 600, when the IVS crossed the minimum safe distance threshold. Since this behavior violated an invariant goal, the value produced by this utility function remains false throughout the remainder of the

simulation. On its own, this state-based utility function is not sufficient to determine whether the IVS collided with the Lead Vehicle. Nevertheless, the metric-based utility function responsible for measuring the satisficement of this goal captures that the IVS did collide with the Lead Vehicle shortly after crossing the minimum safe distance threshold. This collision can be inferred from the utility values that drop to zero.

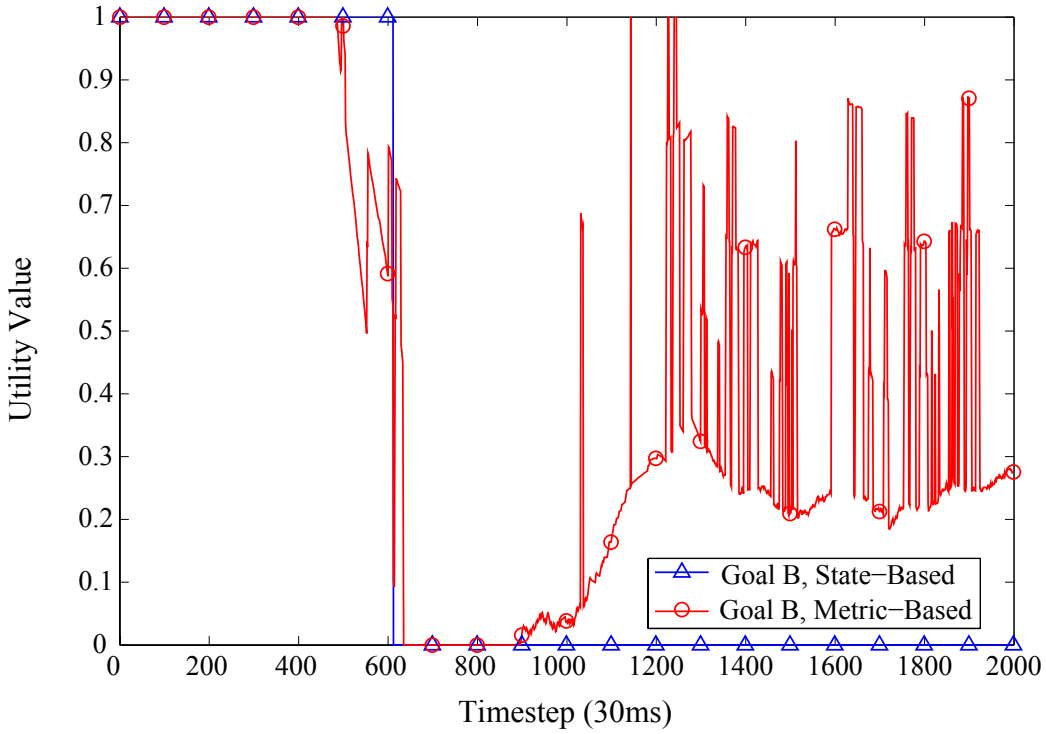


Figure A.14: Utility values for "maintain safe distance goal".

Figure A.15 plots the utility values produced by the metric-based utility function responsible for monitoring the satisfaction of Goal (D) in Figure A.2, which states that the IVS should achieve and maintain a safe speed if an obstacle is detected. As this figure illustrates, the IVS detected the presence of the Lead Vehicle at approximately time step 400. Furthermore, this figure also illustrates how the IVS unsuccessfully attempted to decelerate in order to achieve a safe speed. Lastly, this figure also shows how the IVS was no longer able to achieve a safe speed after time step 700, at it had already crossed the minimum safe distance threshold.

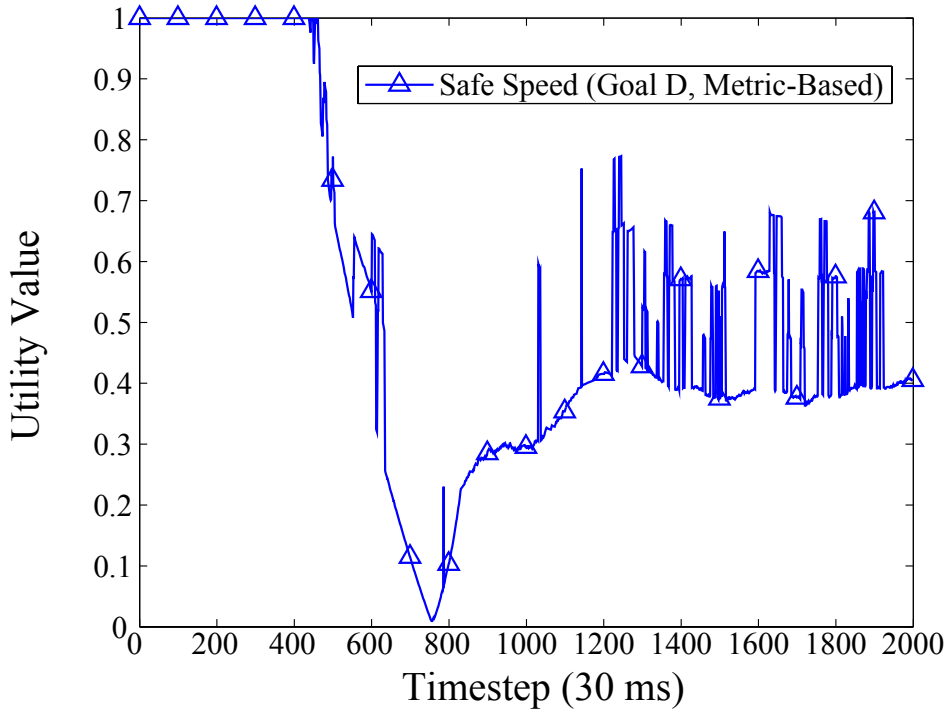


Figure A.15: Utility values for "achieve and maintain a safe speed".

Figure A.16 plots the utility values produced by the metric-based utility function responsible for monitoring the satisficement of non-functional Goal (S) in Figure A.3, which states that the IVS should minimize the acceleration and deceleration rates throughout the simulation when changing its speed or heading. As this figure illustrates, the IVS was able to maintain a degree of satisficement of 0.85 or above throughout the majority of the simulation. Nevertheless, this utility function registered a downward spike in the satisficement of this goal at approximately time step 720, where values suddenly dropped to 0.1. This downward spike captures the rapid deceleration in the IVS's speed produced by the collision between the IVS and the Lead Vehicle.

Lastly, Figure A.17 plots the utility values produced by the metric-based utility function responsible for monitoring the satisficement of RELAXed Goal (R) in Figure A.3, which states that the IVS should achieve and maintain the center of its

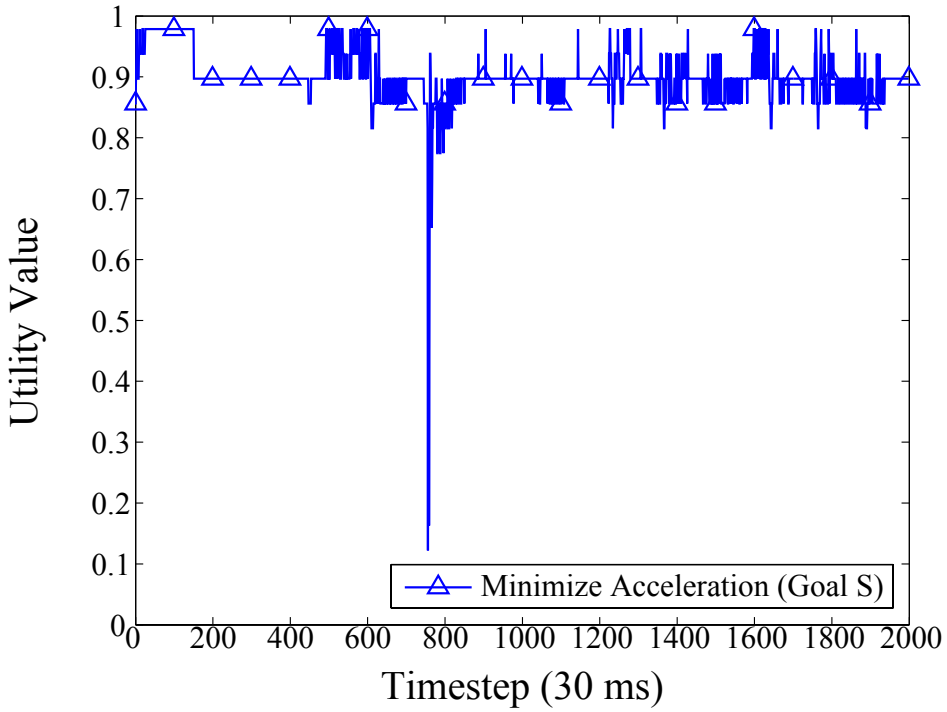


Figure A.16: Utility values for "minimize acceleration and deceleration rates".

lane. As this figure illustrates, the IVS is able to achieve the center of its lane at approximately time step 250. Furthermore, this plot also illustrates how the collision between the two vehicles caused the IVS to slightly shift its heading and thus momentarily depart from the center of the lanes.

A.4 Exploring the Space of Uncertainty

In this section we apply Loki to evaluate how introducing uncertainty at the monitoring infrastructure affects the behavior of the IVS. This case study is intended to explore whether Loki can be applied to a closed commercial simulation platform that does not readily facilitate the automatic modification of fine-grained environmental conditions. To this end, Loki introduces such conditions indirectly via sensory uncertainty. We next describe the simulation and experimental setup, as well as present sample discovered behaviors and a comparison with randomized testing as a baseline.

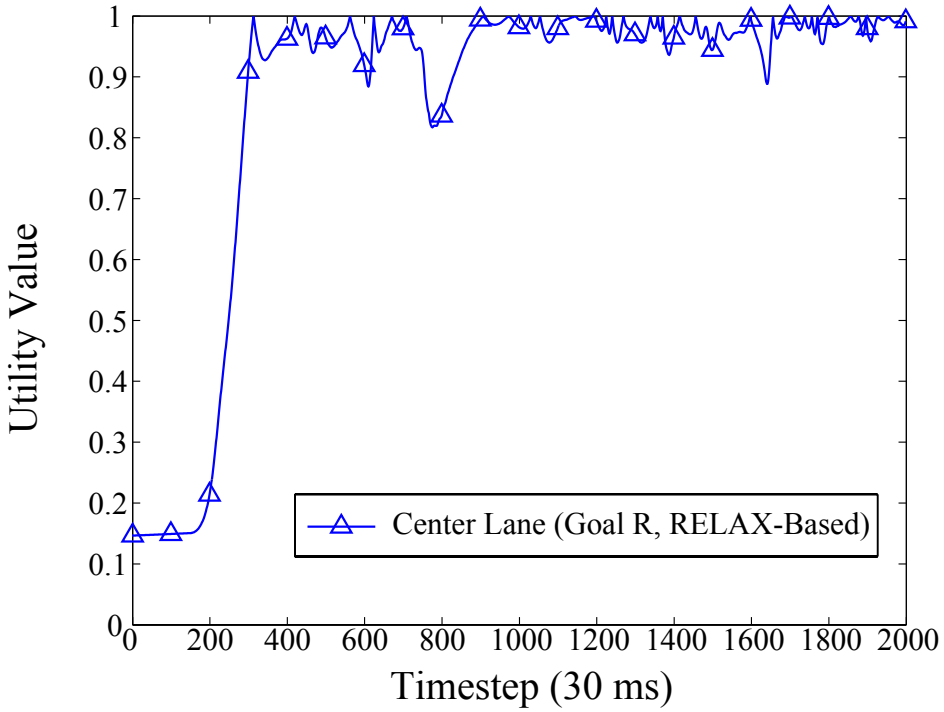


Figure A.17: Utility values for "achieve and maintain center lane".

Simulation and Experimental Setup. For this study, the simulation scenario comprises two autonomous vehicles, an IVS and a Lead Vehicle. Initially, the IVS is positioned 900 meters behind the Lead Vehicle in the same driving lane. Both vehicles begin to accelerate at the same time. While the Lead Vehicle accelerates until it achieves a desired speed of 35 km/h, the IVS continues to accelerate until it reaches a desired speed of 55 km/h. As the IVS approaches the Lead Vehicle from behind, its sensors detect the obstacle and the IVS reconfigures its operational mode to achieve and maintain a safe speed to avoid a potential collision. This safe speed prevents the IVS from crossing into the safe distance zone (see Figure A.1). Shortly thereafter, the Lead Vehicle gradually accelerates until it reaches its new desired speed of 65 km/h, thus increasing its distance from the IVS and enabling the IVS to accelerate, once more, to its desired speed of 55 km/h.

The same scenario is replayed in each simulation throughout this experiment.

However, different operational contexts are applied in each simulation. Note that no sensor is considered to be a single point of failure in this scenario. That is, even after applying the maximum permissible amount of noise to each sensor independently, the IVS is still capable of satisfying its requirements.

Table A.3 specifies the configuration of the genetic and novelty search algorithms for this experiment. With a population of 100 genomes, and a maximum number of ten generations, this particular configuration evaluates exactly 1000 different operational contexts. A Manhattan distance metric is used to compute the distance between utility vectors associated with genomes in the population and novelty archive. After ranking these distances, the novelty of a genome is assigned by computing the mean distance to the seven nearest genomes in the solution space. At the end of each generation, genomes with a novelty value in the top 10% are added to the novelty archive. Lastly, we conducted 25 trials of this experiment for statistical purposes, each with a different seed value that is stored to ensure that results are reproducible.

Table A.3: Loki configuration for IVS experiments.

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

Discovered Behaviors. Loki discovered a mean of 142.42 different behaviors in response to operational contexts out of 1000 evaluations performed in each trial. In addition, for each trial, Loki discovered a minimum and maximum of 77 and 201 different behaviors, respectively. These results suggest that 15% of the operational contexts generated and evaluated by Loki resulted in considerably different behaviors.

Within this set of behaviors, approximately 7.7% of behaviors involved a requirements violation. Upon closer inspection, every requirements violation resulted from impaired self-assessment capabilities in the IVS such that it failed to detect conditions that would indicate a requirement violation. The other 92.3% of behaviors satisfied requirements at run time, though some exhibited latent behaviors. These results confirm the crucial role that system uncertainty plays in determining whether a DAS is able to satisfy its requirements or not.

We analyzed operational contexts that involved requirements violations to determine which combinations of sensor uncertainty had the most impact upon goal satisfaction. As expected, the most detrimental combination of sensor uncertainty involved multiple sensor failures. Although slightly less severe, sensor noise was also detrimental to goal satisfaction. In both cases, a requirements violation occurred because goals were either unfulfilled or their computations unreliable, thereby preventing the IVS from correctly interpreting monitoring data. Results also suggested that noise spikes, where monitoring values were significantly altered for short periods of time, were not as detrimental to goal satisfaction as these had to occur at precise points in time, such as when the IVS makes key decisions, in order to obstruct a goal.

To illustrate the range of behaviors discovered by Loki, we now examine a requirements violation where the IVS was unable to accurately estimate the coasting zone distance due to moderate levels of sensor noise across the GPS and distance sensor agents. In this operational context, the IVS suffered from noise in the GPS, as well as a failed camera and two distance sensors. As a result of this sensory uncertainty, the IVS failed to decelerate in time, crossed into the safe distance zone, collided with the Lead Vehicle, departed from its driving lane temporarily because of the collision and then continued to collide with the Lead Vehicle in order to re-enter the driving lane (in an attempt to satisfy the lane keeping goal), eventually pushing the Lead Vehicle off the road. This behavior captures an undesirable requirements interaction

between the ACC and lane keeping modules.

Careful examination of the interactions between system agents and goals involved in the requirements violation suggest that the root of this failure cascade was the inaccurate computation of the IVS's current velocity. Specifically, although the IVS can choose from two different OR-refinements to estimate its current velocity, if both branches are affected by uncertainty, then the obstacle mitigation is insufficient. In either case, the current velocity estimate is also used to compute the safe speed and safe distance values in the IVS. For example, Figure A.18 plots the real and perceived safe speeds, as calculated based on the current speed of the IVS; the difference between both values is produced by uncertainty in the measurements made by the DAS. As this figure illustrates the IVS consistently calculated a safe speed value that was greater than what the real safe speed value should have been. This difference between perceived and real safe speed values caused the IVS to assume it had a greater stopping distance than it actually had.

Similarly, Figure A.19 plots the actual and perceived distance to obstacles, as measured by distance sensors in the IVS. Again, here the difference between both values is produced by uncertainty in the measurements performed by the IVS's monitoring infrastructure. As this figure illustrates, uncertainty caused the IVS to consistently compute a distance to obstacle that was greater than the real value. Thus, the IVS was unable to accurately compute its safe speed and stopping distance, and thus was also unable to decelerate in a timely manner, resulting in a collision.

Most of the different behaviors discovered by Loki satisfied requirements. However, in several instances Loki also discovered latent behaviors that should be disallowed. For example, in one latent behavior, the IVS decelerated and achieved its safe speed without crossing into the safe distance zone. Thereafter, the IVS began to abruptly accelerate and decelerate in order to maintain its safe speed, causing a jerking motion. In a different latent behavior, the IVS abruptly decelerated just be-

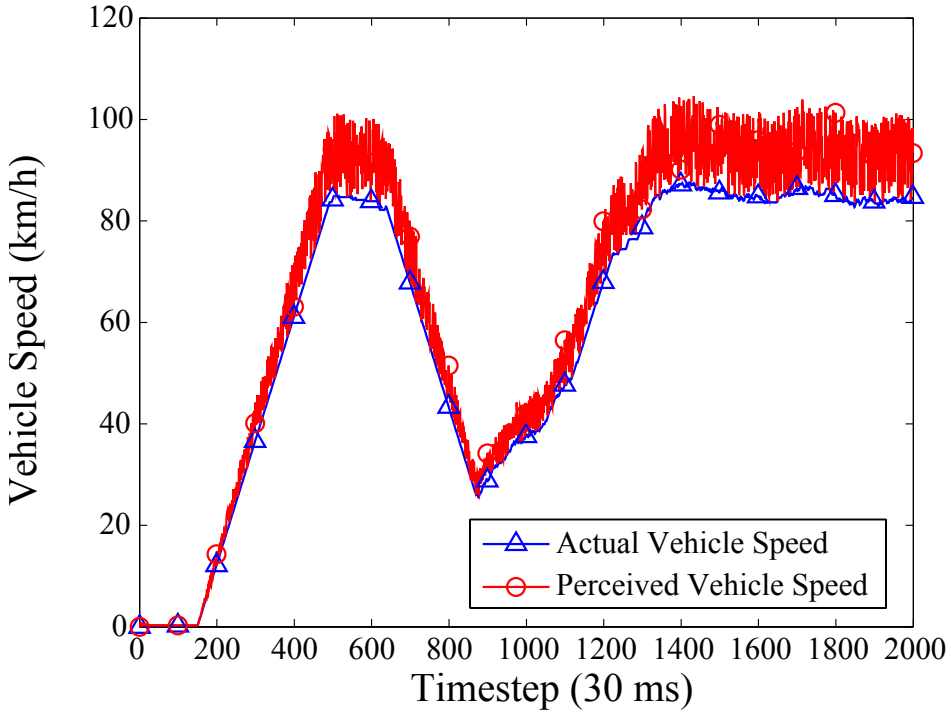


Figure A.18: Effects of uncertainty upon the IVS's current speed self-assessment abilities.

fore crossing into the safe distance zone and, instead of achieving its safe speed as in the previous behavior, the IVS now continued decelerating almost to a complete stop before eventually accelerating. In both cases, such latent behaviors should be prevented as they negatively impact passenger comfort and may cause a collision with other vehicles trailing the IVS, respectively.

Randomized Search Comparison. We now compare Loki with a randomized search algorithm as a baseline comparison. Randomized testing is an effective method for testing software units when no additional information is available to guide the testing process [2]. In this experiment, we randomly generated 1000 operational contexts and evaluated their effects upon the IVS. Since each simulation was executed sequentially and independently, the randomized search algorithm had no knowledge about what areas of the solution space had been explored. Once all simulations completed, we proceeded to compute the differences between the utility values for

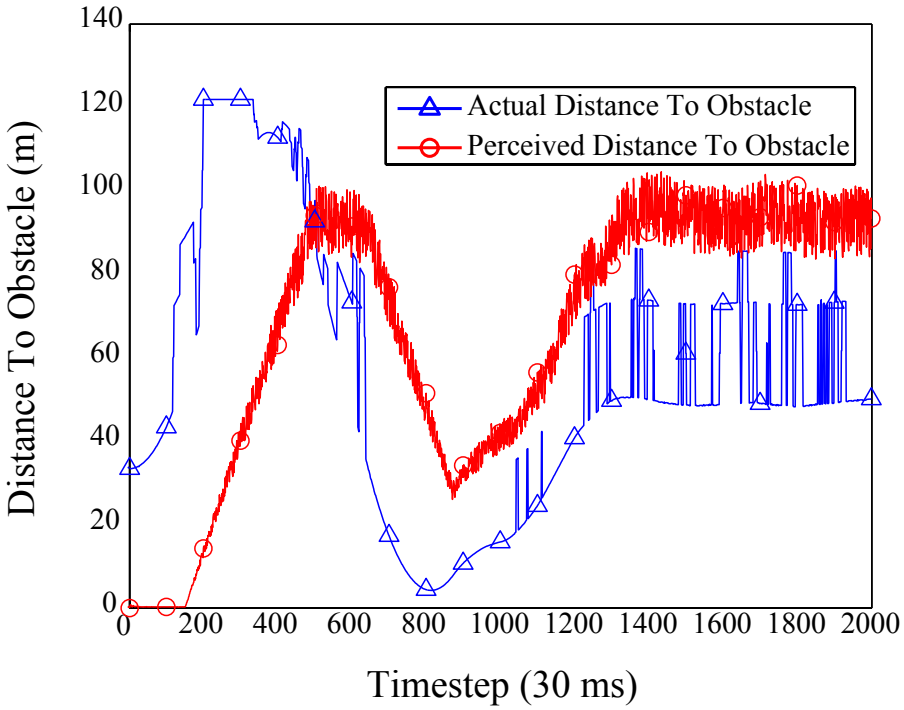


Figure A.19: Effects of uncertainty upon the IVS's safe distance self-assessment abilities.

each sensor configuration (i.e., the novelty value) by using the k-nearest neighbors novelty metric. Once we computed novelty values for each behavior produced by the randomly generated sensor configuration, we compared them with the results generated by Loki. Intuitively, the approach that produced the larger set of novelty values managed to cover a larger portion of the solution space.

In general, randomized search also discovered operational contexts that produced requirements violations. Out of 1000 evaluations per trial, randomized search discovered a mean of 194.1 behaviors that involved requirements violations. These results confirm that randomized search is a valuable tool for discovering test cases that trigger failures in the system-to-be. Although it seemed that randomized search discovered more requirements than Loki, upon closer inspection, most of the randomly discovered behaviors were similar to each other. As the box plot in Figure A.20 shows, we found statistically significant differences in the novelty values between Loki and randomized

search (Wilcoxon ranksum test, $p < 0.001$), where Loki had a median value of 14.7 and randomized search had a median value of 12.3. As this box plot also captures, Loki was able to consistently find behaviors with larger novelty values than randomized search. That is, in every trial, Loki discovered several behaviors with novelty values greater than 40, which are considerably larger than the mean and median values obtained by both approaches. Often, the magnitude of the novelty value correlated with the severity of the latent behavior and requirements violation.

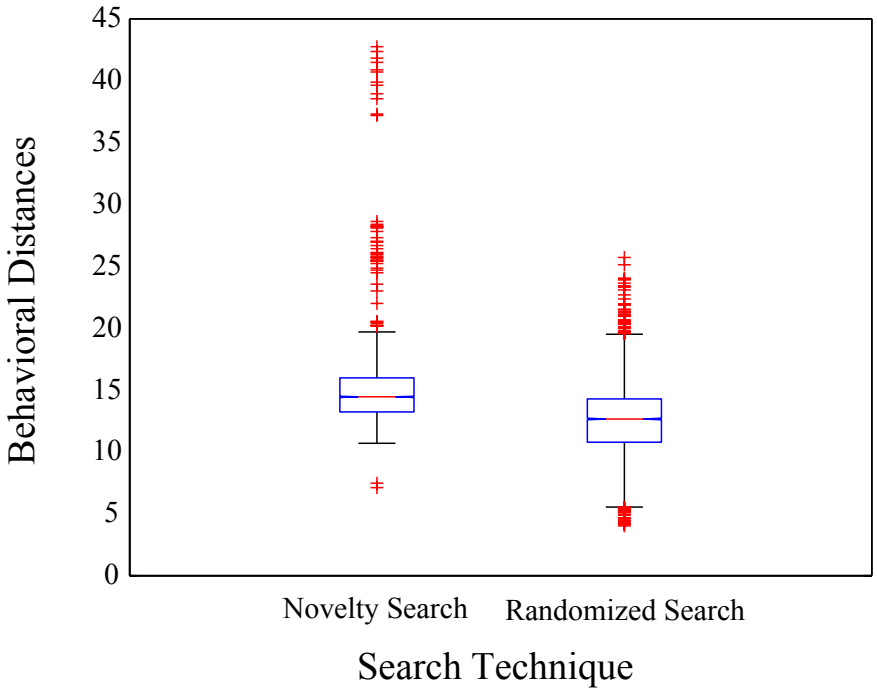


Figure A.20: Box plot comparison of discovered behaviors between Loki and randomized search.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions Compututing Systems*, 19(4):483–518, 2001.

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

[3] J. W. Backus. The syntax and semantics of the proposed international algebraic language of zurich. In *Proceedings of the International Conference on Information Processing*, ICIP, pages 125–132. UNESCO, 1959.

[4] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards decentralized self-adaptive component-based systems. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'08, pages 57–64, Leipzig, Germany, May 2008. ACM.

[5] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy goals for requirements-driven adaptation. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 125–134, Sydney, Australia, October 2010. IEEE.

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

[7] Dan M. Berry, Betty H.C. Cheng, and Ji Zhang. The four levels of requirements engineering for and in dynamic adaptive systems. In *Proceedings of the 11th International Workshop on Requirements Engineering Foundation for Software Quality*, REFSQ, 2005.

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

[9] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[10] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Muller, Mauro Pezze, and Mary Shaw. Engineering self-adaptive systems through feedback loops. In Betty H.C. Cheng, Rogerio

de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, pages 48–70. Springer-Verlag, Berlin, Heidelberg, 2009.

[11] Javier Cámara, Carlos Canal, Javier Cubo, and Juan Manuel Murillo. An aspect-oriented adaptation framework for dynamic component evolution. *Electron. Notes Theor. Comput. Sci.*, 189:21–34, 2007.

[12] Urszula Chajewska, Daphne Koller, and Dirk Ormoneit. Learning an agent's utility function by observing behavior. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML, pages 35–42, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

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

[14] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software engineering for self-adaptive systems: A research roadmap. In Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap. Springer-Verlag, 2009.

[15] 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, Shanghai, China, 2006. ACM.

[16] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[17] Geoff Coulson, Paul Grace, Gordon Blair, Wei Cai, Chris Cooper, David Duce, Laurent Mathy, Wai Kit Yeung, Barry Porter, Musbah Sagar, and Wei Li. A component-based middleware framework for configurable and reconfigurable grid computing. *Concurr. Comput. : Pract. Exper.*, 18(8):865–874, 2006.

[18] Anne Dardenne, Axel van Lamsweerde, and Stephen Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[19] Robert Darimont and Axel van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *SIGSOFT Software Engineering Notes*, 21(6):179–190, October 1996.

[20] Paul de Grandis and Giuseppe Valetto. Elicitation and utilization of application-level utility functions. In *In the Proceedings of the Sixth International Conference on Autonomic Computing (ICAC'09)*, pages 107–116, Barcelona, Spain, June 2009. ACM.

[21] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

[22] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[23] Marco Dorigo and Luca Maria Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1:53–66, 1996.

[24] J. Eggermont and J. I. van Hemert. Stepwise adaptation of weights for symbolic regression with genetic programming. In *Proceedings of the Twelfth Belgium/Netherlands Conference on Artificial Intelligence*, pages 259–266, January 2008.

[25] Naeem Esfahani. A framework for managing uncertainty in self-adaptive software systems. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, Lawrence, Kansas, USA, November 2011.

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

[27] Ramon Fabregat, yezid Donoso, Bejamin Baran, Fernando Solano, and Jose L. Marzo. Multi-objective optimization scheme for multicast flows: A survey, a model and a MOEA solution. In *Proceedings of the 3rd International IFIP/ACM Latin American Conference on Networking*, pages 73–86, New York, NY, USA, 2005. ACM.

[28] M. S. Feather, S. Fickas, A. van Lamsweerde, and C. Ponsard. Reconciling system requirements and runtime behavior. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 50–59, Washington, DC, USA, 1998. IEEE Computer Society.

[29] Stephen Fickas and Martin S. Feather. Requirements monitoring in dynamic environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering*, pages 140–147, Washington, DC, USA, 1995. IEEE Computer Society.

[30] Franck Fleurey and Arnor Solberg. A domain specific modeling language supporting specification, simulation and execution of dynamic adaptive systems. In *Proceedings of the 2009 International Conference on Model Driven Engineering Languages and Systems (Models '09)*, volume 5795 of *Lecture Notes in Computer Science*, pages 606–621, Denver, Colorado, USA, October 2009. Springer.

[31] Frank Fleury, V. Dehlen, Nelly Bencomo, Brice Morin, and Jean Marc Jezequel. Modeling and validating dynamic adaptation. In *Proceedings of International Workshop on Models at Run Time*, Toulousse, France, October 2008.

[32] Erik M. Fredericks, Andres J. Ramirez, and Betty H.C. Cheng. Towards runtime testing of dynamic adaptive systems. In *To Appear in the Proceedings of the 2013 International Symposium on Software Engineering for Self-Adaptive Systems*, San Francisco, CA, USA, May 2013.

[33] Christian Gagne and Marc Parizeau. Genericity in evolutionary computation software tools: Principles and case-study. *International Journal on Artificial Intelligence Tools*, 15(2):173–194, 2006.

[34] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 195.

[35] 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.

[36] David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In *In the Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, Brisbane, Australia, December 2001.

[37] Debanjan Ghosh, Raj Sharman, Rao H. Raghav, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, January 2007.

[38] Heather J. Goldsby and Betty H.C. Cheng. Goal-oriented modeling of requirements engineering for dynamically adaptive systems. In *Proceedings of the 14th International Conference on Requirements Engineering*, RE, pages 345–346, Minneapolis, Minnesota, USA, September 2006. IEEE Computer Society.

[39] Heather J. Goldsby and Betty H.C. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, pages 568–583, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Heather J. Goldsby and Betty H.C. Cheng. Automatically discovering properties that specify the latent behavior of uml models. In *Proceedings of the 13th*

International Conference on Model Driven Engineering Languages and Systems, pages 316–330, Oslo, Norway, October 2010. Springer-Verlag.

[41] Heather J. Goldsby, Betty H.C. Cheng, Philip K. McKinley, David B. Knoester, and Charles A. Ofria. Digital evolution of behavioral models for autonomic systems. In *Proceedings of the Fifth IEEE International Conference on Autonomic Computing*, pages 87–96, Chicago, Illinois, 2008. IEEE Computer Society.

[42] Hasaan Gomaa and Mohamed Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *WICSA'04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 79, Washington, DC, USA, 2004.

[43] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*, ICSE'91, pages 23–34, Austin, Texas, United States, May 1991. IEEE Computer Society Press.

[44] Mark Harman. The current state and future of search based software engineering. In *IEEE International Conference on Software Engineering 2007, Future of Software Engineering*, pages 342–357, Minneapolis, Minnesota, 2007. IEEE Computer Society.

[45] Matthew J. Hawthorne and Dewayne E. Perry. Exploiting architectural prescriptions for self-managing, self-adaptive systems: A position paper. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Managed Systems*, WOSS'04, pages 75–79, Newport Beach, California, 2004. ACM.

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

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

[48] Divyesh Jadav and A. Choudhary. Designing and implementing high-performance media-on-demand servers. *IEEE Parallel Distributed Technology: Systems Applications*, 3(2):29–39, 1995.

[49] Adam C. Jensen and Betty H.C. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In *Proceedings of the 2010 Genetic and Evolutionary Computation Conference (GECCO 2010)*, Portland, OR, USA, 2010. ACM.

[50] 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.

[51] Keneth A. De Jong. *Evolutionary Computation, A Unified Approach*. The MIT Press, March 2002.

[52] Ivan J. Jureta, Stephane Faulkner, and Pierre-Yves Schobbens. Achieving, satisficing, and excelling. In *Proceedings of the 2007 Conference on Advances in Conceptual Modeling: Foundations and Applications*, ER'07, pages 286–295, Auckland, New Zealand, 2007. Springer-Verlag.

[53] Gail Kaiser, P. Gross, G. Kc, and J. Parekh. An approach for autonomizing legacy systems. In *Proceedings of the First Workshop on Self-Healing, Adaptive, and Self-MANaged Systems*, 2002.

[54] Kimberly Keeton, Dirk Beyer, Ernesto Brau, and Arif Merchant. On the road to recovery: Restoring data after disasters. *SIGOPS Operating Systems Review*, 40(4):235–248, April 2006.

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

[56] Kimerbly Keeton and Arif Merchant. Challenges in managing dependable data systems. *SIGMETRICS Performance Evaluation Review*, 33(4):4–10, 2006.

[57] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the 1995 IEEE International Conference on Neural Networks*, pages 1942–1948, Perth, WA, Australia, November 1995.

[58] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[59] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, page 313, Vienna, Austria, 2001. ACM.

[60] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *In the Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[61] David B. Knoester, Andres J. Ramirez, Betty H.C. Cheng, and Philip K. McKinley. Evolution of robust data distribution among digital organisms. In *Proceedings of the 11th annual conference on Genetic and Evolutionary Computation (GECCO '09)*, pages 137–144 (Nominated for Best Paper), Montreal, Canada, July 2009.

[62] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Claudio Magalh, and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware'00: IFIP/ACM Intl. Conf. on Dist. Sys. Platforms*, pages 121–143. Springer-Verlag, 2000.

[63] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992.

[64] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. on Soft. Eng.*, 16(11):1293–1306, 1990.

[65] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering 2007*, pages 259–268, Minneapolis, Minnesota, May 2007. IEEE Computer Society.

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

[67] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards requirements-driven autonomic systems design. *SIGSOFT Software Engineering Notes*, 30(4):1–7, May 2005.

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

[69] Emmanuel Letier and Axel van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 53–62, Newport Beach, California, 2004. ACM.

[70] Thanasis Loukopoulos and Ishfaq Ahmad. Static and adaptive distributed data replication using genetic algorithms. *Journal Parallel Distributed Computing*, 64(11):1270–1285, 2004.

[71] Robyn R. Lutz and Ines Carmen Mikulski. Requirements discovery during the testing of safety-critical software. In *Proceedings of the 25th International Conference on Software Engineering*, pages 578–583, Portland, OR, USA, 2003. IEEE Computer Society.

[72] Philip K. McKinley, Betty H.C. Cheng, Andres J. Ramirez, and Adam C. Jensen. Applying evolutionary computation to mitigate uncertainty in dynamically-adaptive, high-assurance middlware. *Journal of Internet Services and Applications Special Issue on the Future of Middleware*, 3(1):51–58, November 2011.

[73] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H.C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[74] O. Michel. Webots: Professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[75] Marius Mikalsen, Nearchos Paspallis, Jacqueline Floch, Erlend Stav, George A. Papadopoulos, and Akis Chimaris. Distributed context management in a mobility and adaptation enabling middleware. In *Proceedings of the 2006 ACM symposium on Applied Computing*, pages 733–734, New York, NY, USA, 2006. ACM.

[76] David Montana, Talib Hussain, and Tushar Saxena. Adaptive reconfiguration of data networks using genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1141–1149, San Francisco, CA, USA, 2002.

[77] Mirko Morandini, Loris Penserini, and Anna Perini. Modelling self-adaptivity: A goal-oriented approach. In *Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 469–470, Venice, Italy, October 2008. IEEE Computer Society.

[78] Mirko Morandini, Loris Penserini, and Anna Perini. Towards goal-oriented development of self-adaptive systems. In *In the Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 9–16, Leipzig, Germany, May 2008. ACM.

[79] Hausi Muller, Mauro Pezze, and Mary Shaw. Visibility of control in adaptive systems. In *Proceedings of the Second International Workshop on Ultra-Large-Scale Software-Intensive Systems*, ULSSIS'08, pages 23–26, Leipzig, Germany, May 2008. ACM.

[80] Mohammad A. Munawar, Michael Jiang, and Paul A.S. Ward. Monitoring multi-tier clustered systems with invariant metric relationships. In *In the Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 73–80, Leipzig, Germany, May 2008. ACM.

[81] Cu D. Nguyen, Anna Perini, Paolo Tonella, Simon Miles, Mark Harman, and Michael Luck. Evolutionary testing of autonomous software agents. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 521–528, Budapest, Hungary, May 2009. International Foundation for Autonomous Agents and Multiagent Systems.

[82] Charles A. Ofria and Claus O. Wilke. Avida: A software platform for research in computational evolutionary biology. *Artificial Life*, pages 191–229, 2004.

[83] Peyman Oreizy, Michael Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[84] Liliana Pasquale and Paola Spoletini. Monitoring fuzzy temporal requirements for service compositions: Motivations, challenges, and experimental results. In *Proceedings of the 2011 International Workshop on Requirements Engineering for Systems, Services and Systems of Systems*, RESS, pages 63–69, Trento, Italy, August 2011. IEEE.

[85] Andres J. Ramirez. Design patterns for developing dynamically adaptive systems. Master's thesis, Michigan State University, East Lansing, MI 48823, 2008.

[86] Andres J. Ramirez and Betty H.C. Cheng. Evolving models at run time to address functional and non-functional adaptation requirements. In *Proceedings of the Fourth Workshop on Models at Run Time*, volume 509, pages 31–40, Denver, Colorado, USA, October 2009. ACM.

[87] Andres J. Ramirez and Betty H.C. Cheng. Adaptive monitoring of software requirements. In *Proceedings of the 2010 Workshop on Requirements at Run Time*, RE@RunTime, pages 41–50, Sydney, Australia, September 2010. IEEE Computer Society.

[88] Andres J. Ramirez and Betty H.C. Cheng. Design patterns for developing dynamically adaptive systems. In *Proceedings of the Workshop on Software Engineering for Adaptive and Self-Managed Systems (SEAMS)*, Capetown, South Africa, May 2010.

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

[90] Andres J. Ramirez, Betty H.C. Cheng, Philip K. McKinley, and Benjamin E. Beckmann. Automatically generating adaptive logic to balance non-functional tradeoffs during reconfiguration. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC, pages 225–234, Washington, DC, USA, June 2010. ACM.

[91] Andres J. Ramirez, Erik M. Fredericks, Adam C. Jensen, and Betty H.C. Cheng. Automatically relaxing a goal model to cope with uncertainty. In *Proceedings of the Fourth Inernational Symposium on Search Based Software Engineering*, volume 7515, pages 198–212, Riva del Garda, Italy, September 2012. Lecture Notes in Computer Science.

[92] Andres J. Ramirez, Adam C. Jensen, Betty H.C. Cheng, and David B. Knoester. Automatically exploring how uncertainty impacts behavior of dynamically

adaptive systems. In *Proceedings of the 2011 International Conference on Automatic Software Engineering*, ASE'11, pages 568–571, Lawrence, Kansas, USA, November 2011.

[93] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the Sixth International Conference on Autonomic Computing*, pages 97–106, Barcelona, Spain, June 2009.

[94] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. Plato: A genetic algorithm approach to run-time reconfiguration of autonomic computing systems. *Journal of Cluster Computing*, 14(3):229–244, September 2011.

[95] Ingo Rechenberg. *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* PhD thesis, Technical University of Berlin, 1973.

[96] Sebastian Risi, Sandy D. Vanderbleek, Charles E. Hughes, and Kenneth O. Stanley. How novelty search escapes the deceptive trap of learning to learn. In *Proceedings of the Eleventh Annual Conference on Genetic and Evolutionary Computation*, pages 153–160, Montreal, Quebec, Canada, July 2009. ACM.

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

[98] William N. Robinson. A requirements monitoring framework for enterprise systems. *Requirements Engineering*, 11(1):17–41, March 2006.

[99] S. Masoud Sadjadi, Philip K. McKinley, and Betty H.C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. In *DEAS'05: Proc. of the 2005 workshop on Design and Evolution of Autonomic Application Software*, New York, NY, USA, 2005. ACM.

[100] Pete Sawyer, Nelly Bencomo, Emmanuel Letier, and Anthony Finkelstein. Requirements-aware systems: A research agenda for re self-adaptive systems. In *Proceedings of the 18th IEEE International Requirements Engineering Conference*, pages 95–103, Sydney, Australia, September 2010.

[101] Christian Seybold, Martin Glinz, and Silvio Meier. Simulation-based validation and defect localization for evolving, semi-formal requirements models. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 1–10. IEEE Computer Society, 2005.

[102] Mary Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Software Engineering Notes*, 20(1):27–38, 1995.

[103] Vitor E. Silva Souza and John Mylopoulos. From awareness requirements to adaptive systems: A control-theoretic approach. In *Proceedings of the Second International Workshop on Requirements at Run Time*, pages 9–15, Trento, Italy, August 2011. IEEE Computer Society.

[104] Lee Spector and A. Robinson. Genetic programing and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, 2002.

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

[106] Axel van Lamsweerde and Emmanuel Letier. Integrating obstacles in goal-driven requirements engineering. In *Proceedings of the 20th International Conference on Software Engineering*, pages 53–62, Kyoto, Japan, 1998. IEEE Computer Society.

[107] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26(10):978–1005, October 2000.

[108] Yves Vandewoude and Peter Ebraert. Tranquillity: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, December 2007.

[109] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *Proceedings of the First IEEE International Conference on Autonomic Computing*, pages 70–77, New York, NY, USA, 2004. IEEE Computer Society.

[110] Kristopher Welsh and Pete Sawyer. When to adapt? identification of problem domains for adaptive systems. In *Proceedings of the 14th International Conference on Requirements Engineering: Foundations for Software Quality*, pages 198–203, Montpellier, France, 2008. Springer-Verlag.

[111] Kristopher Welsh, Pete Sawyer, and Nelly Bencomo. Towards requirements aware systems: Run-time resolution of design-time assumptions. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 560–563, Lawrence, Kansas, USA, November 2011. IEEE Computer Society.

[112] Kristopher Welsh and Peter Sawyer. Understanding the scope of uncertainty in dynamically adaptive systems. In *Proceedings of the Sixteenth International Working Conference on Requirements Engineering: Foundation for Software Quality*, volume 6182, pages 2–16, Essen, Germany, June 2010. Springer.

[113] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *Proceedings*

of the First International Conference on Autonomic Computing, ICAC, pages 2–9, New York, NY, USA, May 2004. ACM.

[114] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Jean-Michel Bruel. RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In *Proceedings of the 17th International Requirements Engineering Conference (RE '09)*, pages 79–88, Atlanta, Georgia, USA, September 2009. IEEE Computer Society.

[115] R. Witty and D. Scott. Disaster recovery plans and systems are essential. Technical Report FT-14-5021, Gartner Research, September 2001.

[116] Sunny Wong, Melissa Aaron, Jeffrey Segall, Kevin Lynch, and Spiros Mancoridis. Reverse engineering utility functions using genetic programming to detect anomalous behavior in software. In *Proceedings of the 17th Working Conference on Reverse Engineering*, WCRE, pages 141–149, Beverly, Massachussetts, USA, October 2010. IEEE.

[117] Z. Yang, Betty H.C. Cheng, R. E. Kurt Stirewalt, J. Sowell, S. M. Sadjadi, and Philip K. McKinley. An aspect-oriented approach to dynamic adaptation. In *Proceedings of the First Workshop on Self-Healing Systems*, pages 85–92, New York, NY, USA, 2002. ACM.

[118] Eric S.K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering*, pages 226–235, Annapolis, MD, USA, January 1997. IEEE Computer Society.

[119] Ji Zhang and Betty H.C. Cheng. Specifying adaptation semantics. In *WADS'05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1–7, New York, NY, USA, 2005. ACM.

[120] 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, New York, NY, USA, 2006. ACM.

[121] Ji Zhang, Betty H.C. Cheng, Zhenxiao Yang, and Philip K. McKinley. *Enabling safe dynamic component-based software adaptation*, volume 3549 of *Lecture Notes in Computer Science*, pages 194–211. Springer Berlin / Heidelberg, 2005.

[122] Ji Zhang, Heather J. Goldsby, and Betty H.C. Cheng. Modular verification of dynamically adaptive systems. In *Proceedings of the Eighth International Conference on Aspect-Oriented Software Development*, 2009.