Meta-Management of Collections of Autonomic Systems

Thomas J. Glazier

CMU-S3D-23-110

December 2023

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Garlan, Chair
Bradley Schmerl
Fei Fang
Betty H.C. Cheng (Michigan State University)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Software Engineering.

This work was supported by the following funding: Navy N000141612961, NSF CNS1116848, ONR-N000141612961, Stevens Institute of Technology RT119CMU20140623 and NSA Lablet H98230-18-D-0008.

Keywords: Autonomic Systems, Meta-Management

To Bradley and Benjamin
The journey to your goals is long, winding, broken,
full of obstacles, and worth more than the destination.

To Chrissy
Anything with you and nothing without you

Abstract

To meet the demands of high availability and optimal performance in dynamic environments, modern systems deploy autonomic or self-adaptation mechanisms. However, increasingly today's enterprise systems are compositions of many subsystems, each an adaptive system. Currently, each autonomic manager operates to maintain locally defined quality-of-service (QoS) objectives, but their independent actions often lead to globally sub-optimal results. Commonly, human administrators handle situations in which the collection of autonomic systems is behaving sub-optimally. However, generating a plan to change the configurations of the constituent autonomic managers is a complex and challenging task in the management of a single autonomous system, but the challenge is exacerbated where there may be tradeoffs in how to balance configuration options across the collection of autonomic subsystems.

These challenges can be addressed by introducing an automated approach, referred to as *meta-management*, that provides a formal basis for reasoning about changes to the configurations of autonomic subsystems. The automated approach to meta-management is then established as part of a framework that can be used to instantiate a higher level autonomic manager, referred to as a meta-manager, that provides assurance about, and improves the performance of a collection of autonomic systems. This approach and framework includes a MAPE-K control loop specialized to the needs of meta-management, a domain specific language, SEAM, that enables the practical specification of adaptation policies, and a taxonomy of strategy synthesis techniques. The practicality, effectiveness, and applicability of the approach are then evaluated against three case studies.

The first is an AWS Shopping Cart system in which a meta-manager is established to manage a collection of autonomic system represented by a front end user interface, a middleware services tier, and a database services tier. This case study was selected to evaluate the ability of the meta-manager to improve the homeostatic operations of the collection of autonomic systems on popular architectural pattern, code base, and operations platform that is in wide industrial use.

The second is the Google Control plane in which a meta-manager was established to manage a collection of autonomic systems that suffered a significant outage. This case study was selected because it presented a well documented and specific failure scenario that occurred during the period of the research of this thesis that cause of which was, partially, a result of human-centric management of a collection of autonomic systems.

Finally, the third is a simulation of an electrical grid cascade failure that represents the Northeast Blackout of 2003. This case study was selected because it presents an example of a failure of human-centric management of a collection of autonomic systems that was exhaustively documented that occurred in a context outside of information technology and/or cloud based providers. This provides credibility to the applicability claim of the thesis

Acknowledgments

I would first like to thank my advisor, David Garlan, for taking on, working with, teaching, and, on some days, tolerating a less than conventional PhD student. The support, advice, knowledge, and skill he generously shared are invaluable and the efforts he took to ensure my success will not be forgotten.

I would like to thank all the members of the ABLE group through out the years especially Bradley Schmerl and Javier Cámara. Each and every one of them gave the advice and support that was needed at the time, even if it was not what was being asked for.

I would also like to thank the members of my committee, Bradley Schmerl, Fei Fang, and Betty H.C. Cheng. Their efforts to assist on a PhD committee, especially one so long delayed from the proposal, are greatly appreciated.

I would like to thank Bill Wilson, Al and Darren Simpson, and Vincent Dell'Anno, each of which invested in me and provided the necessary assistance throughout the years to ensure that one day a goal like this could be achieved.

I would like to thank Justin Lowe and his family, Sheila, Jax, and Zoe, for always providing the helping hand, understanding ear, and support to my family when it was needed most.

I would like to thank my mom, Christine Glazier, for imparting in me a life long love of learning and her efforts to help me find and make my own path have been a model for everything I do. I would also like to thank Norm Kerth, Randy Kerth, and those who are no longer with us, Bradley Kerth and Dr. Leroy Kerth, each of which, in their own way, provided the guidance, inspiration, and curiosity to look beyond what the world presents to figure out how it works.

I would like to thank Bradley and Benjamin for their patience over the years for the occasions I could not be with them or had to delay doing something with them. Their understanding, compassion, and generosity with their support is an inspiration to me.

Finally, and most importantly, I would like to thank my wife Chrissy. This journey would simply have not been possible without her unwavering support from the first trip crammed into the back of a car while 7 months pregnant, to the temporary relocation to Pittsburgh with a newborn, to my long absences, to everything else that I have either forgotten about or did not know about through out the years. For all of her sacrifices, love, support, and encouragement through out the years, I am forever grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To meet the demands of high availability and optimal performance in dynamic environments, modern systems deploy autonomic or self-adaptation mechanisms. These autonomic mechanisms are responsible for continuously monitoring operating conditions and effecting changes in the system to ensure defined quality objectives are achieved. For example, during cyber-Monday and other periods of extreme traffic, a large web system like Amazon.com will add server capacity to guarantee acceptable performance for all users. When the traffic returns to normal, the autonomic manager will remove the extra server capacity to prevent unnecessary costs.

However, increasingly today's enterprise systems are compositions of many subsystems, each an adaptive system. Each subsystem has its own defined objectives, reasoning methods, and adaptation tactics. Additionally, they are often built by different vendors, hosted on multiple platforms, and have different implementations. For example, a large web system will have different autonomic managers to oversee the product catalog and video playing user interfaces, another for the middle tier common services, and potentially a fourth for the distributed database system.

Currently, each autonomic manager operates to maintain locally defined quality-of-service (QoS) objectives, but their independent actions often lead to globally sub-optimal results. For example, the autonomic manager of an n-tiered enterprise system could be scaling up the capacity of the middle tier while the manager for the database tier is scaling down: at least one of them is likely to be inconsistent with the best global action. These globally sub-optimal behaviors are the result of the subsystems having incomplete information about the current and future state of their environment, interdependency between systems propagating detrimental behavior, and changes in the global definition of optimal behavior due to shifting organizational priorities. Some of these sub-optimal results can be potentially catastrophic to the collective system. For example, the Northeast Blackout of 2003 was the result of a fault in a specific electrical grid, an autonomic system, which eventually cascaded to over 100 power plants and affected 10 million people in Ontario, Canada, and 45 million people in 8 US states with an estimated economic impact of $6.4 billion [54].

Commonly, human administrators handle situations in which the collection of autonomic systems is behaving sub-optimally. To do so, human administrators evaluate the current state of their individual autonomic systems and the environments to determine if the global quality objectives are likely to be met. When the administrator concludes that intervention is appropriate,

they generate a plan of changes to the configuration of the individual autonomic systems with the goal of improving the collection's performance against the global quality objectives.

However, generating a plan to change the configurations of the constituent autonomic managers is a complex and challenging task. First, the human needs to analyze the current state information for the local environments, the managed systems, and each autonomic manager and predict the potential future states of each to determine the best course of action. Prediction of future states is challenging with multiple dimensions of uncertainty, including how potential future states affect the systems under management, how each autonomic manager will respond, and the likely effect those adaptations will have on the individual systems. Understanding these uncertainties is critical to the human administrator, as their comprehension of this information will directly influence the effectiveness of their planned configuration changes. Second, the human needs to consider multiple quality objectives for the system. These quality objectives often have multiple competing dimensions (e.g., cost and response time) and the relative priority of those dimensions may change over time due to changes in organizational priorities. Third, the human needs to select an appropriate set of configuration changes from a combinatorially large set across the subsystems. Failure to properly consider the space of configuration options has a direct impact on the effectiveness of the configuration changes. Finally, the decisions that humans need to make are often time critical to prevent further degradation in the collection's ability to meet global quality objectives.

These challenges are acute for a human in the management of a single autonomous system, but the challenge is exacerbated where there may be tradeoffs in how to balance configuration options across the collection of autonomic subsystems. The NERC report on the 2003 Northeast Blackout highlights these human limitations [54, p. 94-96] by concluding that the human administrators of the primary power grid involved in the Northeast Blackout were only able to determine that a critical system had failed after 40 minutes and were unable to determine the total extent of the failure or the potential consequences of it despite having another 69 minutes to respond before the 'point of no return'. Consequently, the complex and ad-hoc nature of a human generating a plan to change the configuration of constituent autonomic managers is unable to provide strong assurances (e.g., regarding optimality of the plan or minimization of risk).

These challenges can be addressed by introducing an automated approach, referred to as *meta-management*, that provides a formal basis for reasoning about changes to the configurations of autonomic subsystems. This automated approach is enabled by the fact that while each subsystem can vary considerably in functional purpose, each of them is autonomic which provides three key advantages over collections of human-adapted systems that can be exploited to enable an automated approach to meta-management: (1) the simplification of the state space, (2) a reduction in the variance of the results of adaptation, and (3) the abstraction of the underlying system. Each will be briefly discussed here, but see chapter 4 for additional detail.

First, as is common with the introduction of a control system, the administrator must identify the key properties and quality of service objectives (QoS) that define the simplified model of the system that will be used to reason about the current state of the system and any adaptations that are needed. For many systems this manifests as an architectural model of the managed system combined with the QoS objectives [25, 39, 120]. This simplified representation of the managed system partially enables the computational scalability of analyzing changes to the configuration of the autonomic subsystems.

Second, enabled by the simplified representation of the system, the introduction of an autonomic manager provides a degree of predictability about the resulting state of the system and the impact to the QoS properties as a result of taking adaptive actions. This predictability reduces the number of potential outcomes that need to be specified which allows for the practical creation of a specification of the autonomic behavior of a subsystem, referred to as an adaptation policy. This partially enables an automated approach to meta-management by providing a simplified model of the adaptive behavior of the autonomic subsystem that allows a meta-manager to reason about potential changes.

Finally, the abstraction of the implementation details of the individual managed subsystems by treating each of them as a black box autonomic system allows for two key assumptions.

First, we will assume that each autonomic subsystem has a set of configuration parameters that can be used to tune the behavior of the subsystem to within a specified range of behaviors. The autonomic configuration options are the interface by which human administrators establish the organizational and business preferences/tradeoffs and constrain the behaviors of the autonomic manager for each subsystem.

Second, an adaptation policy can be specified for the autonomic subsystem in which the state(s) that could be the result of an adaptive action are dependent upon the state of the environment, the state of the managed system, and the the configuration parameters of the autonomic manager. Due to the desired predictability of the results of adaptation gained with the introduction of an autonomic manager, it would be expected that if the same set of conditions were to occur on two different occasions that the autonomic manager would select the same adaptation option(s) in both circumstances. Therefore, if one is able to elaborate the states of the managed system and the states of the environment, it is also possible to predetermine which adaptation option(s) the autonomic manager would deploy and determine what the potential resulting states of the managed system would be.

An automated approach to meta-management can be established as part of a framework [1] that can be used to instantiate a higher level autonomic manager, referred to as a meta-manager, that provides assurance about, and improves the performance of a collection of autonomic systems. To simplify and enable the creation of the framework, the meta-manager will implement a specialized MAPE-K control loop.

The MAPE-K control loop [57] is a commonly used architecture for the implementation of autonomic managers which control the adaptive behaviors of a specific system. It is composed of four elements, monitoring(M), analysis(A), planning(P), and execution(E), with shared knowledge(K) that enables the process. For the purposes of a meta-manager, a specialized version of the control loop, referred to as a Meta-MAPE-K loop, will be established. While each component of the MAPE-K loop has specializations for the purpose of meta-management, the knowledge and planning components are the most significantly impacted.

The Meta-Knowledge component of the Meta-MAPE-K control loop is responsible for at least three kinds of knowledge. The first is the specification of the adaptive behavior of each autonomic subsystem (i.e., the adaptation policies). The second kind of required information is a global utility function that provides the definition of 'good' for the QoS objectives for the collection of autonomic systems. The third is contextual information about the collection of au-

[1] By framework we mean a library of reusable code, a set of tools, and design principles

tonomic systems that is unknown or only partially known to the autonomic subsystems, referred to as global knowledge. There are at least two types of global knowledge: (1) information on the interrelationships between individual subsystems, local environments, and global properties and (2) constraints on individual subsystems and global state. For example, often collections of autonomic systems are composed together according to a plan referred to as a system architecture. The system architecture creates a set of dependencies and correlations between the autonomic subsystems that can be used to better predict the most likely state of the managed systems in the future. Additionally, the context in which the collection of autonomic systems is operating has constraints that must be adhered to (e.g., maximum operating cost).

The Meta-Planning component of the Meta-MAPE-K loop is responsible for synthesizing a plan of changes, referred to as a meta-strategy, to the configurations of the autonomic subsystems to improve performance of the collection of autonomic systems against the global QoS objective as defined by the global utility function. However, the choice of synthesis technique (e.g., Stochastic multi-player game or Monte Carlo Analysis) used to generate adaptation strategies is characterized by a tradeoff between timeliness, assurance, and computational scalability. Each strategy synthesis technique has a profile for the timeliness of the technique to generate an adaptation plan, the level of assurance that adaptation plan can provide, and the computational scalability of the technique to handle systems of realistic size. Consequently, the selection of a strategy synthesis technique for meta-management is a critical task that must carefully consider the timeliness, assurance, and scalability requirements of the context in which the collection of autonomic systems is operating to ensure the effectiveness of the meta-manager.

1.1 Thesis

An automated approach to the meta-management of collections of autonomic systems, as implemented by a meta-manager, can address the ad-hoc and error-prone and costly process of human-centric administration. Therefore, this thesis investigates the following claim:

Thesis Statement

We can provide engineers the ability to establish an automated solution to the autonomic control of a significant subset of collections of autonomic systems that is effective, applicable in a variety of contexts, and is practical to implement and maintain using the following elements:

1. An automated approach to the management of collections of autonomic systems.
2. A domain specific language used to abstract and represent the adaptation behavior for each autonomic subsystem.
3. Guidance to determine the appropriate strategy synthesis technique for the context in which the collection of autonomic systems is operating.
4. A reusable software framework that simplifies the development of a meta-manager.

The key claims in this thesis relate to practicality, effectiveness, and applicability. Let us

expand on each of these individual claims.

Practicality: The framework will be practical with respect to:

1. *Ease of Use.* The framework will allow individuals with standard state-of-the-practice knowledge in software engineering to use it to instantiate an automated solution to manage an applicable collection of autonomic systems.
2. *Human Feasible Configuration.* The framework will provide methods that will allow a human administrator to specialize a meta-manager to a particular collection of autonomic systems including the specification of adaptive behavior for each subsystem, referred to as an adaptation policy.
3. *Scalability.* The framework will be capable of scaling to handle systems of practical industrial size.

Effectiveness: The framework will be effective with respect to:

1. *Improved Performance.* A collection of autonomic systems managed by an autonomic manager will experience improved performance against defined global quality objectives over human-based management.
2. *Timeliness and Assurance.* Because the framework and approach do not mandate a specific synthesis technique, an engineer implementing the framework can select a synthesis technique that best fits the level of timeliness and assurance required for the context.

Applicability: The framework will be applicable to a significant subset of collections of autonomic systems with the following characteristics:

- Each subsystem is non-adversarial[2] in nature.
- Each of the subsystems provides an interface to adjust the configuration parameters of the autonomic managers.
- The adaptive behavior that each autonomic subsystem will employ for a given state of the environment under a set of configuration parameters can be specified.

1.2 Case Studies

The evidence for many of the arguments that substantiate the claims of this thesis is primarily based on three case studies: (1) Amazon Shopping Cart Web System, chapter 7, (2) Google Control Plane, chapter 8, and (3) Electrical Grid Cascde Failure, chapter 9. Therefore, the level of realism represented in each of case studies and collections of autonomic systems under study is important to understand. Therefore, this section will highlight the key properties and features of each of the case studies and the realism present in each.

The comparison of the case studies across key dimensions is presented in table 1.1. The following are descriptions of each of the key properties:

[2]Meaning that the individual autonomic subsystems have no motivation to act contrary to the global objectives and accept the authority of the meta-manager.

	AWS Shopping Cart	Google Control Plane	Power Grid
Functional Area	IT Consumer	IT Infrastructure	Industrial
Instability	No	Yes	Yes
Homeostatic	Yes	Yes	No
Global Knowledge	Yes	No	Yes
Actual or Simulation	Actual	Actual	Simulation
Autonomic Manager	ElasticBeanStalk & DAX Autoscaling	GCP Managed Instance Groups	SCADA
Synthesis Method	DTMC with Simulation	Stochastic Multiplayer Game	DTMC with Monte Carlo
Synthesis Toolset	PRISM	PRISM-Games	Matlab
Mimic Admin Action	No	Yes	Yes

Table 1.1: Case Study Comparison

- **Functional Area** - The enterprise context in which the collection of autonomic systems described in the case study are operating.
- **Instability & Homeostatic** - As discussed in chapter 4, the automated approach to meta-management is capable of addressing both situations in which the goal is to continuously improve the operations of the collections of autonomic systems, referred to as homeostatic operations [93, 111], and also improving the response of the collection of autonomic systems to rare edge case events which causes instabilities which can have severe consequences.
- **Global Knowledge** - Whether the case study included the use of global knowledge or information that was not available or only partially available to the individual subsystems.
- **Actual or Simulation** - Whether the case study was performed on real-world systems or the collections of autonomic systems was simulated.
- **Autonomic Manager** - The type of autonomic control system(s) responsible for managing the autonomic behaviors of the subsystems.
- **Synthesis Method** - The technique used to synthesize the meta-strategy for the use case.
- **Synthesis Toolset** - The program used to perform the analysis and synthesize the meta-strategy.
- **Mimic Administrator Action** - If the actions of the meta-manager can be directly compared to the actual or recommended actions of human administrators.

AWS Shopping Cart

The Amazon Web Services(AWS) Shopping Cart case study uses an actual system, not a representative system. It uses the actual code base, architectural models, and products and services which power countless production grade shopping cart systems for AWS customers. Specifically, it is based upon an open source and publicly available shopping cart system designed and

maintained by AWS [108]. The exemplar shopping cart is designed to have three architectural tiers: (1) a front end user interface (UI), (2) a middleware services tier, and (3) a data services tier. While all three tiers are controlled by autonomic management systems, the autonomic manager(s) for the middleware services tier do not provide an interface to update their configuration parameters. Therefore, the middleware services are not eligible for the automated approach to meta-management presented in this thesis. However, the other two tiers are controlled by two different, but related, autonomic managers in ElasticBeanStalk [103] for the FrontEndUI and DAX [100] Autoscaling for the data services tier. The code base, architectural pattern, and the products and services in-use are all realistic of thousands of shopping cart in use in production grade situations.

This case study was selected to evaluate the ability of the meta-manager to improve the homeostatic operations of the collection of autonomic systems on popular architectural pattern, code base, and operations platform that is in wide industrial use.

Google Control Plane

The Google Control Plane case study is based upon an actual incident documented by Google in [52] in which the system that processes changes in the networking configuration for Google Cloud customers failed. While the technical details of the control plane system are not publicly available, the experimental platform was instantiated using information from the Google incident report using the same or similar services offered publicly to GCP customers. The most relevant of which are the Managed Instance Groups (MIGs) which provide autonomic management capabilities to clusters of server instances. By autoscaling individual clusters of application server instances, the MIGs are responsible for ensuring there is sufficient processing capacity available to process the network change requests. As each of the clusters are designed to be practically identical, there is very little diversity in the QoS objectives or autonomic configuration options of each cluster. Additionally, while the control plane is an IT centric system, it does not handle the same volume of requests as a front end website might and, due to the specific nature of its function, its environment has only moderate variation in the number of requests sent for a given period of time. The system used in this case study is, to the extent possible, using the same architectural pattern, products and services as the actual Google Control Plane system.

This case study was selected because it presented a well documented and specific failure scenario that occurred during the period of the research of this thesis that cause of which was, partially, a result of human-centric management of a collection of autonomic systems.

Power Grid Cascade Failure

The exemplar system used in the power grid case is a simulated system as it is impractical to perform research experiments on actual electrical grids. Additionally, because the behavior of electrical grids is highly dependent on the topology of their physical components, there is no pre-built simulation of specific events like the Northeast Blackout of 2003. The analysis of a specific event is of limited value as, in the power grid research community, it is difficult to draw meaningful conclusions that would generalize to other power grids. Therefore, the state-of-the-practice in electrical grid research is to use one of several standard models of representative

power grids and, since the behavior of electrical grids is well described by sets of differential equations, simulate the events of interest depending on the type of research being performed. To perform such a simulation, two components are required: the simulation environment and the model of an electrical grid.

In this case study, the simulation environment is established in Matlab 2023a [60] using the MatPower [81, 128] set of open source libraries for electrical grid optimization and simulation which has been cited in over 4000 papers [126]. Additionally, the AC-CFM libraries [86, 87], which augment the MatPower libraries, simulate an electrical grid cascade failure. The electrical grid model is based on the IEEE 39 Bus system model [5, 33] and has been cited over 750 times [125]. The interconnection of the individual electrical grids to compose a larger electrical grid is unique to this thesis, but follows a similar approach used in [66]. Therefore, while the exemplar system presented in this case study is simulated, it follows the best practices of the electrical grid research community with appropriate modifications necessary to test the automated approach to meta-management presented in this thesis.

This case study was selected because it presents an example of a failure of human-centric management of a collection of autonomic systems that was exhaustively documented that occurred in a context outside of information technology and/or cloud based providers. This provides credibility to the applicability claim of the thesis.

1.3 Contributions

The contributions of this work are:

- **Automated Approach:** An automated approach to the meta-management for collections of autonomic systems that provides a formal basis for reasoning about the changes to the configuration of the constituent autonomic systems that improves the performance on a time scale appropriate to the context.
- **Strategy Synthesis Taxonomy:** A taxonomy of strategy analysis and synthesis techniques that provides guidance on each of their expected timeliness and assurance capabilities and the types of contexts for which those techniques might be best suited.
- **SEAM - A DSL for Meta-Management:** A language that addresses the challenges of representing the behavior of autonomic subsystems independent of the adaptation plan synthesis techniques, represents the global objective of the collection of autonomic systems, and the global knowledge about the interdependencies between the individual autonomic subsystems and/or the local environments and constraints on the operations of the collection of autonomic systems.
- **Implementation Framework:** A reusable software framework that implements the common components and functional requirements for the implementation of a meta-manager.

1.4 Thesis Layout

The remainder of this thesis is organized as follows:

- Chapter 2 presents a motivating example of the management of collections of autonomic systems and the research challenges it presents.
- Chapter 3 discusses the previous research related to this work.
- Chapter 4 presents the automated approach to meta-management and how that is implemented in a specialized Meta-MAPE-K loop.
- Chapter 5 presented the details of the SEAM language specialized for the needs of meta-management.
- Chapter 6 presents the taxonomy of strategy synthesis techniques
- Chapter 7 presents a case study based on an AWS Shopping Cart web system
- Chapter 8 presents a case study based on the Google Control Plane
- Chapter 9 presents a case study based on an electrical grid cascade failure
- Chapter 10 discussed how the claims presented in chapter 1 are validated.
- Chapter 11 discusses this approach and provides details on its applicability and its limitations and provides direction for future work.
- Chapter 12 presents the conclusions of the thesis.

Chapter 2

Exemplar Scenario and Research Challenges

This chapter presents a representative system that will be used throughout the remainder of the thesis to illustrate and provide examples for many of the key concepts and motivate a set of research questions that will frame the research carried out in this thesis.

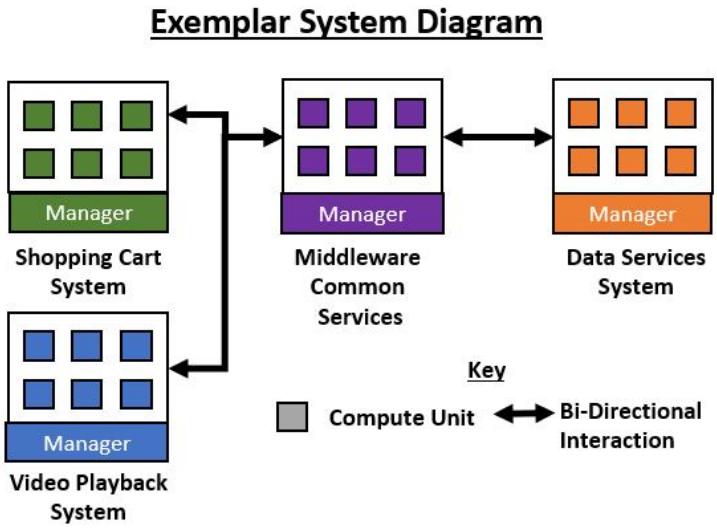


Figure 2.1: Exemplar System Diagram

Consider a large scale multipurpose system that is built to handle a variety of functional use cases and because each use case has a different set of quality objectives, the global system is subdivided into subsystems that are built for a specific use case. For example, an e-commerce platform, like amazon.com, might be subdivided into a shopping cart subsystem, to manage the display and purchase of individual products, and a video playback subsystem. Additionally, each of these user facing systems is dependent upon a common set of services that are necessary to provide functionality to users. For example, both shopping cart and video playback subsystems would need functionality related to individual users like their purchase history and security and access authorizations. This results in the need for a middleware common services subsystem to provide the necessary functionality. Further, the raw information about important entities such as the users and products will need to be stored and retrieved on-demand requiring a data services subsystem. When a multipurpose system is subdivided into

System	Primary Users	Adaptation Tactics	Utility Dimensions	Config Parameters
Video Playback System	Internet Users	Add Server, Change Fidelity	Response Time, Runtime Cost	Capacity Buffer
Shopping Cart System	Internet Users	Add Server, Change Fidelity	Response Time, Runtime Cost	Capacity Buffer
Middleware Services	Front End Systems	Add Server	Response Time, Runtime Cost	Capacity Buffer
Data Services	Middle Tier Systems	Add Server, Change Replication	Response Time, Runtime Cost	Capacity Buffer

Table 2.1: Sub System Properties

functional units it is common to organize it into layers and tiers in a manner consistent with an n-tier architecture pattern [37]. See figure 2.1 for a diagram of the exemplar system.

To ensure that each subsystem maintains its specific quality objectives, it is common to implement an autonomic manager to monitor the state of the system under management and make changes to the configuration of the system as necessary in response to various environmental stimuli. However, autonomic managers do not typically react to the environmental stimuli directly. Instead, they track the values of key system metrics that directly relate to the desired quality of service (QoS) objectives (see table 2.1), which are dependent upon both the environment and the architectural configuration of the managed system. For example, focusing specifically on the shopping cart web system, the environment establishes the user load for the system, the environmental stimuli, and the autonomic manager tracks the average web page response time, which is influenced by both the user load on the system and various architectural properties such as the number of servers in use and the fidelity of the content being presented.

When the autonomic manager determines that a QoS objective (e.g., average page response time) is either not currently being met, referred to as reactive adaptation, or is unlikely to be met within a particular time horizon, referred to as proactive adaptation, then the autonomic manager examines potential alternative configurations for the architectural properties of the managed system. For example, adding servers or lowering the fidelity of the content, or both, are potential alternative architectural reconfigurations of the managed system that will influence the average page response time.

However, typically, an autonomic manager does not simply select *any* course of action that will improve performance against a quality of service objective; rather it attempts to select the 'best' course of action, referred to as an adaptation strategy, under some constraints and preference conditions. For example, the local environment of the shopping cart system might increase the user load causing the average page response time to rise above a preconfigured acceptable level. This condition triggers a response from the autonomic manager which determines what the 'best' adaptation strategy is under the current set of configuration options and the current state of the managed system. For the shopping cart system, the autonomic manager has the choice of adding servers, which will raise the operating cost of the system or lowering the fidelity of the

content which will decrease the quality of the users' interactions with the system. The organization's preference for which of these choices should be deployed is commonly established in the form of a utility function, which in this case might, for example, prioritize adding servers up to a preconfigured maximum before reducing the fidelity of the content. The complete set of adaptation strategies the autonomic manager takes in response to the combinations of the environmental conditions and states of the system is referred to as an adaptation policy and an example adaptation policy for the shopping cart system is presented in figure 2.2.

Avg. Page Response Time	Server Count	Content Fidelity	Deployed Adaptation	New Server Count	Buffer Servers	New Fidelity	Expected Cost
< 2.5 s.	10	High	[None]	10	1	High	$1100
	15	High	[None]	15	2	High	$1700
	20	High	[None]	20	2	High	$2200
2.5 s. – 3.5 s.	10	High	Add Server	11	1	High	$1200
	15	High	Add Server	16	2	High	$1800
	20	High	Reduce Fidelity	20	0	Low	$2000
3.5 s. – 4.5 s.	10	High	Add Server	13	1	High	$1400
	15	High	Add Server	18	2	High	$2000
	20	High	Reduce Fidelity	20	0	Low	$2000
> 4.5s	10	High	Add Server	20	0	High	$2000
	15	High	Add Server	20	0	High	$2000
	20	High	Reduce Fidelity	20	0	Low	$2000

Autonomic Manager Configuration: Maximum Cost: $2000 Capacity Buffer: 10%

Environmental Constraints: Cost per Server per Unit Time: $100

Quality Objective (Utility Function): < 2.5s = 1, 2.5s – 3.5s = 0.6, 3.5s – 4.5s = 0.3, > 4.5s = 0

Figure 2.2: Exemplar of Autonomic Behavior

The following are the descriptions of the individual columns in table 2.2:

1. **Avg. Page Response Time** - The average page response time for the managed system. This is the monitored metric of the managed system.
2. **Server Count** - The current number of servers deployed for the solution.
3. **Fidelity** - The current setting for the content fidelity of the system.
4. **Deployed Adaptation** - The adaptation tactic that was deployed.
5. **New Server Count** - The new server count after the deployed adaptation tactic.
6. **Buffer Servers** - The number of servers that are added to account for the capacity buffer.
7. **New Fidelity** - The new value for the content fidelity after the deployed adaptation tactic.
8. **Expected Cost** - The expected run time cost of the servers.

As can be observed in figure 2.2, the local environment for the shopping cart system establishes the user load. However, the user load cannot be directly observed by the autonomic manager, so the administrators have determined that the observable QoS property is the 'Average Page Response Time' which is dependent upon the user load of the local environment. When the 'Average Page Response Time' is below the target threshold of 2.5 seconds, the adaptation manager deploys no adaptation strategy. However, when the 'Average Page Response Time' is above the target threshold at 3.0 seconds or 4.0 seconds the adaptation manager chooses to deploy an adaptation strategy and make changes to its architectural configuration. Importantly, the selection of which adaptation strategy to deploy is dependent upon the current state of the managed system. For example, if the system is running below the maximum cost, $2000, then the autonomic manager adds additional server capacity. However, if the system is running at the maximum cost, then adding additional server capacity is no longer an available adaptation tactic as part of an adaptation strategy and the system instead reduces the content fidelity.

However, the adaptation strategy the adaptation manager selects is dependent upon the set of configuration options established in the autonomic manager (e.g., maximum cost and capacity buffer). If the configuration options change, then the choices of which adaptation strategy to deploy might also change.

Avg. Page Response Time	Server Count	Content Fidelity	Deployed Adaptation	New Server Count	Buffer Servers	New Fidelity	Expected Cost
< 2.5 s.	10	High	[None]	10	1	High	$1100
	15	High	[None]	15	2	High	$1700
	20	High	[None]	20	2	High	$2200
2.5 s. – 3.5 s.	10	High	Add Server	11	1	High	$1200
	15	High	Add Server	16	2	High	$1800
	20	High	Add Server	20	2	High	$2000
3.5 s. – 4.5 s.	10	High	Add Server	13	1	High	$1400
	15	High	Add Server	18	2	High	$2000
	20	High	Add Server	20	2	High	$2000
> 4.5s	10	High	Add Server	20	0	High	$2000
	15	High	Add Server	20	0	High	$2000
	20	High	Reduce Fidelity	20	0	Low	$2000

Autonomic Manager Configuration: Maximum Cost: $2500 Capacity Buffer: 10%

Environmental Constraints: Cost per Server per Unit Time: $100

Quality Objective (Utility Function): < 2.5s = 1, 2.5s – 3.5s = 0.6, 3.5s – 4.5s = 0.3, > 4.5s = 0

Figure 2.3: Exemplar of Autonomic Behavior, New Configuration

In figure 2.3, the 'maximum cost' configuration option has been modified from $2000 to $2500 which has resulted in changes to the adaptation decisions of the autonomic manager, high-

lighted in red. Specifically, because additional resources are available, the autonomic manager chooses to deploy the 'Add Server' tactic as part of a strategy instead of 'Reduce Fidelity'.

In addition to the configured constraints (e.g., maximum cost) and preferences (e.g., adding servers vs. decreasing content fidelity) that define the tradeoff space for the autonomic manager, there is potentially an additional set of configuration options available to the autonomic manager that serve as architectural guidelines that should be adhered to, but can be temporarily violated or ignored as needed. For example, the autonomic manager for the shopping cart system might have a 'capacity buffer' setting that sets the guideline for how much spare processing capacity the web system should have available to handle small fluctuations in user load. In the event that the shopping cart system is running near its maximum cost then this 'capacity buffer' setting might be ignored in order to comply with that defined constraint. Finally, there are also defined constants that are a result of something in the operating context of the autonomic manager and managed system and cannot be changed. For example, the cost per server per unit time for the shopping cart system is a constant defined by the context and potentially required for the autonomic manager to make appropriate adaptation decisions.

While the autonomic managers for the managed systems in this exemplar are distinct with different quality objectives, implementations, architectural properties, preferences, constraints, constants, and guidelines they will all function similarly to the shopping cart example as described. Table 2.1 outlines the similarities and differences between the subsystems in this exemplar.

However, while each subsystem serves a specific functional purpose, it is only when the individual subsystems are composed into a collective system that the combination of them provides the functional capabilities for which the global multipurpose system was designed. This global system has its own set of quality objectives and constraints that must be met. For example, the organization can define that a single user interaction with the shopping cart system should not take longer that 3 seconds to respond. This interaction is dependent upon not just the shopping cart system but also on the performance of the middleware common services and the data services systems. If any one of the systems in the interaction chain fails to meet its individual quality objectives, then the global quality objective is unlikely to be met. As such, the autonomic managers are individually configured to try and maintain a level of service that will give the best chance for the global objectives to be met. The individual autonomic managers can provide a level of assurance about the performance of the systems they manage, but there is little assurance about the collective performance.

> **Research Question 1:** How to provide assurance on the behavior of the collection of autonomic systems?

Human administrators address this lack of assurance by trying to configure the autonomic managers for each of the subsystems to give the best chance for the global quality objectives to be met. However, managing the performance of a collection of autonomic systems is a challenging task for human administrators that is complicated by several factors.

First, the autonomic behavior of the individual subsystems is dependent upon at least three factors: the state of the environment, the state of the system under management, and the current configuration of the autonomic manager. The behavior of the autonomic manager for a single

subsystem is often only partially known by the human administrators due to the combinatorial complexity in understanding how the dependencies between the environment, managed system, and the configuration of the autonomic manager interact. This complexity is further increased by uncertainty in both the current and future states of the local environment and the system under management. This combinatorial complexity also provides a barrier to the practical specification of the expected autonomic behaviors that could be useful to a human administrator.

> **Research Question 2:** How to enable the practical analysis of adaptation policies given the uncertainty in the future state of the managed system?
> **Research Question 3:** How to enable the practical specification of adaptation policies for individual autonomic subsystems?

Second, the global quality objectives and constraints are dependent upon the interactions and dependencies between the individual subsystems. Changes in the configuration of one subsystem (e.g., the maximum cost constraint) can have a global impact beyond that individual subsystem. The complex web of dependencies between changes to the configuration of individual autonomic managers and the impacts of those changes on both the global quality objectives and on other dependent subsystems is only partially understood by human administrators. In some scenarios, human administrators will have to continuously make adjustments to the configurations of autonomic subsystems to try and improve the performance of the collection of subsystems without violating the constraints (e.g., the maximum cost of the autonomic manager configuration).

> **Research Question 4:** How to synthesize a plan of changes to the configurations of the autonomic subsystems that improves the performance of the collection of autonomic systems?

Third, both the local quality objectives for the individual subsystems and the global quality objectives for the composed collection are likely to have multiple quality objectives with often competing dimensions that are subject to change over time due to changes in organizational priorities. Developing a plan of changes to optimize the configurations of each autonomic subsystem to balance these competing and layered objectives is subject to significant error by human administrators often resulting in sub-optimal results and sometimes in catastrophic outcomes.

> **Research Question 5:** How to synthesize a plan of changes that balances competing organizational priorities?

Finally, the process of synthesizing an adaptation strategy to update the configurations of the autonomic subsystems cannot take an unknown or unrestricted amount of time to complete. The context in which each collection of autonomic subsystems operates will dictate the amount of time in which the changes must be made or else the conditions will change which is likely to invalidate the synthesized adaptation strategy. The amount of time required for a human to understand the complexity of the adaptation policies and dependencies between the individual systems can be prohibitive and is not a generally viable option for the effective management for collections of autonomic systems in non-trivial systems.

Research Question 6: How to synthesize a plan of changes on a time scale appropriate to the context?

However, humans do have an advantage in the management of collections of autonomic systems. Because collections of autonomic systems are often composed according to well established patterns (e.g., an N-Tier architecture), the human administrators have knowledge of how the subsystems should interact with each other and the characteristics of those interactions. Additionally, because the systems were designed and created by human engineers for a specific functional purpose, the environments in which the subsystems operate must be at least partially understood to ensure the systems fulfill their intended purpose. Human administrators can leverage this knowledge to better understand the current state of the collection of autonomic systems and, potentially, better predict the future state of individual autonomic subsystems.

Research Question 7: How to leverage the knowledge about the structure of the system and environments to improve the effectiveness of managing a collection of autonomic systems?

These research questions define the primary challenges in developing an automated approach to the management of collections of autonomic systems. Fortunately, there are several areas of current research that can partially address some of these research questions. The following chapter presents the current state of research in autonomic systems, collections of adaptive systems, strategy synthesis and assurance in autonomic systems, and control theory for autonomic systems.

Chapter 3

Related Work

There are four key areas that were reviewed to see what potential options they offered in addressing the research questions provided in chapter 2: (1) autonomic systems, (2) collections of adaptive systems, (3) strategy synthesis and assurance in autonomic systems, and (4) control theory for autonomic systems. This chapter will explore each of these areas and elaborate on how each relates to the research questions.

3.1 Autonomic Systems

Several of the research questions are at least partially addressed by existing work in autonomic systems, see [71, 94] for surveys of the field. A key component of the approaches to autonomic systems is that of a feedback loop that monitors the state of the system and the environment and adapts the system as necessary to improve performance against defined quality-of-service (QoS) objectives [14]. The predominant approach to the feedback loop is a common architectural pattern referred to as the MAPE-K loop [57]. The MAPE-K architecture pattern establishes a closed control feedback loop and is defined in [67]. MAPE-K includes five distinct components:

1. **Monitor** - Components gather and pre-process relevant context information from entities in the execution environment.
2. **Analyze** - Supports decision making on the necessity of the adaptation.
3. **Planning** - Generates actions to affect the target system.
4. **Execution** - Implements the plan of action generated by the planner with the goal of adapting the managed system.
5. **Knowledge** - Enables data sharing and communication among the components of the feedback loop.

This type or architectural based approach offers several benefits, including the ability to abstract the adaptive operations of the subsystems to the right level of detail to facilitate analysis as opposed to lower level algorithmic details [70]. Additionally, architecture based adaptation is beneficial because it allows for the potential specialization or reuse of existing architecture

analyses [59, 69, 116] to asses the impact of potential of changing the configuration of the autonomic subsystems. The architecture based approach to adaptation, combined with an appropriate strategy synthesis technique, has been shown to provide assurance on the behavior of individual systems, see [24].

However, as popular as the MAPE-K approach is for the management and control of individual managed systems, it is not specifically designed for the more specialized task of providing assurance on the behavior of a collection of autonomic systems (RQ1). Therefore, it is necessary to define a specialized MAPE-K architectural pattern that can provide the required assurance. One of the requirements for such a specialized pattern is the management of specific types of knowledge including: (1) the adaptation policies for each of the autonomic subsystems which relates to RQ2 and RQ3, (2) models for each of the autonomic subsystems, global system, and local environments which relates to RQ2 and RQ3, (3) the global utility function which relates to RQ5, and (4) the knowledge about the structure of the system and environments which relates to RQ7. These models and information need to be maintained at run-time and used to reason about the changes that should be made to the configuration parameters of the autonomic subsystems which relates to RQ4.

3.2 Collections of Autonomic Systems

As previously discussed, to meet the complex functional objectives of organizations, autonomic systems are often composed together into an ensemble. The most common architectural approach to composing individual autonomic systems into an ensemble is an agent based approach as described in [64]. The primary benefit of this approach is that it is an effective way of partitioning the problem space often in congruence with the organizational structure which allows for the effective decomposition of complex problems [63]. These individual systems interact with each other to achieve the functional objectives of the ensemble system. Agent based approaches have been successfully implemented in a number of functional domains including manufacturing [41], bioinformatics and health care [26], and robotics [28] to name a few. However, as noted in [63], agent based software systems suffer from a significant drawback: the behavior of the overall system is unpredictable because of the strong possibility of emergent behavior. This is problematic in contexts which require high degrees of assurance and predictability in the future states of the system.

However, there is recent work in multi-agent self adaptive systems. Specifically, [90] and [32] details an approach to the coordination of adaptive activities among the individual autonomic subsystems. While this is interesting in it own right, this does not provide guidance on any of the research questions highlighted in chapter 2. This is because the goal of this thesis is to modify the configurations of the autonomic subsystems to improve their performance against the global objective, not coordinate their individual adaptive actions. Additionally, [3] proposes a model that abstracts the autoscaling services of multiple cloud to provide a method about reasoning about adaptive changes. While this method relates to RQs 2 and 3, the model that it provides abstracts a specific set of services and would be impractical to support collections of autonomic systems with different adaptive behaviors even if the authors do abstract the different cloud vendors.

Further, [38] explores the ability of a collection of autonomic systems to learn by sharing information amongst the individual autonomic subsystems, [29] explores the dynamic composition of collections of autonomic systems, and [4] proposes a domain specific language to describe the activities that take place during a MAPE-K loop. While each of these methods tangentially addresses elements of the RQs, none of them provide directly applicable approaches.

3.3 Strategy Synthesis and Assurance in Autonomic Systems

As discussed in section 3.1, the meta-analyze and meta-planning components of the proposed meta-management approach will achieve two primary objectives: (1) improve the performance of the collection of autonomic systems against a defined global objective (RQ5) and (2) provide assurance about the expected outcomes of the adaptations (RQ1). Various analysis and synthesis techniques have been successfully applied to address these functional objectives in the context of a single autonomic system.

In [18], the authors detail an approach that uses stochastic multi-player games, to analyze the potential variations in designs for the autonomous managers to determine which is most likely to be effective. For example, some of the design time considerations the authors call out are the balance between reactive and proactive adaptation, whether decision making should be centralized or decentralized, and the differences and effect of concurrent execution of adaptations versus sequential execution. In net effect, the authors approach synthesized a design time strategy for the creation of the autonomic manager. The results demonstrated potential improvements ranging between 8% and as high as 38% for the scenarios under consideration.

In [61], the authors use a stochastic multi-player game (SMG) to evaluate uncertainty in the sensing of the current state of the environment at runtime. Their results demonstrated that the use of a SMG to account for the uncertainty outperformed the baseline, which was uncertainty-agnostic, and continued to do so to ever greater degrees as the amount of uncertainty, the standard deviation on the sensed information, increased. This demonstrates that approaches in probabilistic model checking can provide some measure of assurance about the expected outcomes of adaptations even in the presence of uncertainty.

Probabilistic model checking has provided encouraging initial results in improving the performance of and providing assurances about the outcomes of individual autonomic systems. Other work has focused on the challenges, frameworks, benchmarks, and approaches to providing these assurances at run-time including [121] and [23]. However, most of the existing work focuses on the use of probabilistic model checking in the context of a single autonomic system. There is some limited work in the probabilistic model checking of collections or ensembles of autonomic systems including [8], [114], and [20]. This work focuses on verifying individual properties about the communication or negotiation protocols amongst agent based systems, not on their control or mitigation of globally undesirable behaviors.

One area in which no literature was found, and is a topic of this thesis, see chapter 6, is to understand which probabilistic model checking and, potentially, game theory based techniques would be most appropriate for analyzing the various concerns in collections of autonomic systems. For example, a stochastic multi-player game might be appropriate for examining one potential concern (e.g., the reallocation of resources), but other potential techniques like non-zero

sum games, partially observable Markov decision processes (POMDP), discrete time Markov chains, and several others might be more suited for other concerns (e.g., faulted or rogue system analysis).

3.4 Control Theory for Autonomic Systems

Several of the research questions presented in chapter 2 address problems in hierarchical control in which there is an extensive body of work.

Control theory has established a common approach to the creation of hierarchical control systems which decomposes the complex behavior into individual units to divide the decision making responsibility. Each unit of the hierarchy is linked to a node in the tree and commands, tasks, and goals to be achieved flow down the tree from superior nodes, whereas sensations and commands results flow up the tree [2, 31]. Each of the individual units can communicate directly with their peers. There are two distinguishing features of hierarchical control systems related to its layers: (1) each higher layer of the tree operates with a longer time interval of planning and execution time than its immediately lower layer and (2) the lower layers have local tasks, goals, and objectives which are planned and coordinated by higher layers which issue generally dictatorial decisions. This approach is commonly referred to as a subsumption architecture [2, 31] and has been successfully implemented in many functional domains including airplanes and automobiles, manufacturing, and robotics.

This well established model for hierarchical control certainly serves as a guideline for the creation of a meta-manager for the collection of autonomic systems. However, the subsumption architecture approach would not be appropriate for collections of autonomic systems. The control theory approach to hierarchical control is dependent upon the ability to specify, typically in the form of differential equations, the dynamics of the system under control. This approach would be generally impractical, if not impossible, for collections of autonomic systems.

While the approach of control theory to hierarchical management would be challenged in the context of autonomic systems, there are emerging efforts to adapt and adopt various control theory techniques to self-adaptive systems. In [30] the authors present a control design process which enables the analysis and synthesis of an autonomic manager (i.e., controller) that is guaranteed to have the desired properties and behavior. In [40], the authors address two principal topics in the application of control theory to autonomic systems: automating control switching with high level guarantees and bridging the gap between control and self-adaptive system properties. The work to bridge the gap between control theory and self-adaptive systems is nascent and emerging, but it currently does not currently address the specific challenge of hierarchical control of autonomic systems. However, it is expected that as this work progresses many of the approaches to improve the individual autonomic controllers can also present advantages to the proposed meta-management approach. For example, a meta-manager could potentially have a tactic to swap out autonomic controllers for a specific constituent system and guarantees about the behavior of the individual managers will aid the analysis and synthesis of a meta-strategy.

While the available literature in autonomic systems, collections of adaptive systems, strategy synthesis and assurance in autonomic systems, and control theory for autonomic systems partially address some of the research questions, it is still necessary to develop a automated

approach to the management of collections of autonomic systems that enables the practical specification and analysis of the adaptation policies of the individual autonomic subsystems. The subsequent chapter presents an approach that addresses these challenges.

Chapter 4

An Automated Approach to Meta-Management

This chapter presents an automated approach to meta-management that partially addresses the research questions highlighted in the exemplar scenario presented in chapter 2. Specifically, this approach provides assurance on the behavior of the collection of autonomic systems (RQ1), enables the practical analysis of the adaptation policies (RQ2), and enables the synthesis of a plan of changes to the autonomic configuration that improves the performance of the collection of autonomic systems (RQ4) that also balances organizational priorities (RQ5). This chapter is organized as follows: (1) details of the differences between autonomic and non-autonomic systems that enables the possibility and tractability of an automated approach to meta-management, (2) the definition of an adaptation policy and how it can be used to analyze the behavior of autonomic subsystems, (3) how the analysis can be enhanced with additional knowledge not available to the subsystems, and (4) the definition of a meta-manager and how it improves the performance of a collection of autonomic systems.

As discussed earlier, in a non-autonomic system, human administrators handle situations in which the system is behaving sub-optimally. To do so, human administrators evaluate the current state of the system and the environment to determine if the quality-of-service (QoS) objectives are likely to be met. When the administrator concludes that intervention is appropriate, they generate a plan of changes to the system with the goal of improving the performance against the QoS objectives. Once the changes are complete, the human administrator evaluates the new state of the system to determine if additional changes are required. This process continues until the QoS objectives are likely to be met.

However, as a result of the complex combinatorial dependency between the system and the environment in which it operates, human administrators are able to provide only minimal assurance about what the new state of the system will be and the impact to the QoS objectives. Consequently, in practice human administrators configure and deploy an autonomic manager to handle this complexity and provide a higher degree of assurance. As noted in the introduction, and expanded upon here, instantiating an autonomic manager, a type of control system, provides three key advantages over non-autonomic systems that can be exploited to enable an automated approach to meta-management: the simplification of the state space, reduction of variance in the outcomes of adaptive actions, and the abstraction of the underlying system.

Simplification of State Space

As is necessary with any control system, the implementation of an autonomic manager requires the administrator to define the QoS objectives of the managed system that the autonomic manager is intended to achieve. Additionally, the system properties influence the performance of the managed system and can be modified by the adaptation strategies of the autonomic manager. The combination of the system properties and the QoS objectives define the state of the managed system and is a significant simplification of the managed system.

For example, in the shopping cart system presented in chapter 2, the 'Average Page Response Time' is the QoS property that the autonomic manager is attempting to maintain and improve and the 'Server Count' and 'Content Fidelity' are the system properties of the managed system that influence the 'Average Page Response Time'. The combination of the values of these properties define a system state in the autonomic control system. For example, in figure 2.2 the 'Average Page Response Time' having a value of 2.5 seconds with a 'Server Count' of 10 and high 'Content Fidelity' represents a different state of the system than having the same value for 'Server Count' and 'Content Fidelity' with a 'Average Page Response Time' of 3.5 seconds. These three properties are a simplification of the complete state of the managed system that could include properties like the amount of processing speed and available memory available in each compute unit which, to some degree, can influence the QoS properties but are determined to be not as significant as the identified QoS or system properties.

The implementation of an autonomic manager also allows human administrators to define the state of the environment in which the managed system operates. Identifying the observable elements and properties of the environment is an additional simplification compared to all of the possible values of all of possible elements of the environment. For example, the video playback system presented in the exemplar scenario may be able to observe the number of user generated requests for video playback. Similar to the state of the managed system, the state of the local environment is characterised by the observable properties, and their associated values, identified by human administrators as important to the management of the autonomic system and, as with the system's state, these are a subset of all of the possible observable properties that could be used to characterize the subsystem's environment.

The simplified representations of the state of the managed system and the local environment, combined with the level of resolution for each metric, establishes the size of the state space and partially establishes the effectiveness of the analysis through the fidelity of the model to reality and the computational tractability of the analysis.

Reduction of Variance in the Outcomes of Adaptation Actions

When either a human or an autonomic manager adapts the managed system, the goal of the adaptation plan is to achieve a state of the managed system that will maximize the performance of the managed system against the QoS objectives. However, due to circumstances outside of the control of the autonomic manager, the adaptation plan might fail to achieve the target state due to uncertainties inherent to the adaptation process.

There are multiple sources of uncertainty that can impact the outcomes of adaptive actions. One is the uncertainty in the results of adaptive actions. For example, in the shopping cart system

the autonomic manager might attempt to deploy a new server and have that request fail due to an unexpected error in the underlying infrastructure.

Another source of uncertainty is due to circumstances in the environment and/or managed system that might also inhibit the expected impact to the performance of the system against the QoS objectives. For example, network traffic unrelated to any system presented in the exemplar might inflate the 'Average Page Response Time' regardless of any adaptation action taken. Therefore, the state intended by any entity responsible for adaptation is a desired target state, but is not the only state, and may not even be the most likely state, that could be the result of the deployment of an adaptation strategy.

Other sources of uncertainty include the time it will take to deploy the adaptive actions, see [82], errors in the measurements of the environment and/or managed system, and several more. Therefore, when either a human or autonomic manager deploys an adaptation strategy there is uncertainty in the outcome of the state of the managed system and, consequently, the impact to the QoS objectives. However, while the outcome of deploying an adaptation strategy experiences the same uncertainty in the outcome regardless of the entity responsible for adaptation, there is a significant difference in how predictable the result will be between human based adaptation and autonomic adaptation.

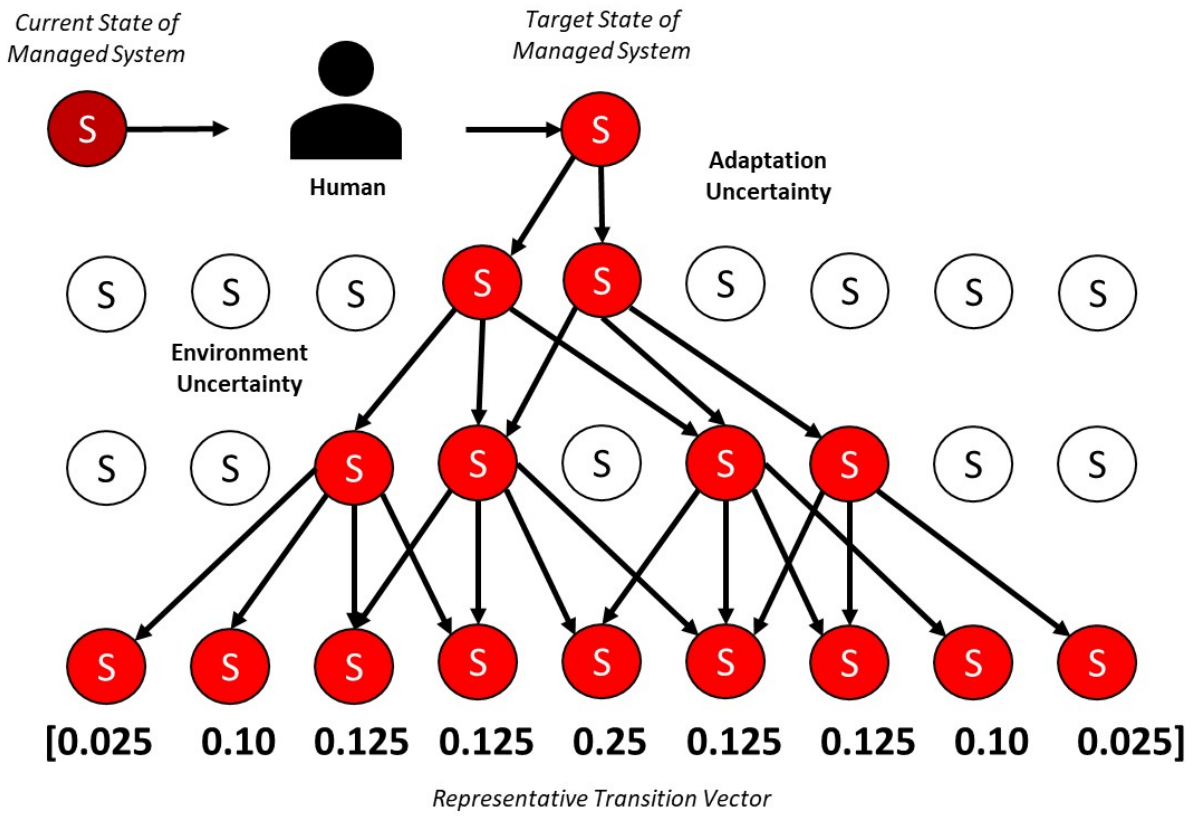


Figure 4.1: Exemplar Representation of Uncertainty in Human-Centric Adaptation

When a human administrator is responsible for performing the adaptive actions it is impractical to expect a human administrator to consider each of the relevant dimensions of both the

environment and the managed system, all of the options for each of those dimensions, and evaluate them in the same manner with the same weights of importance each time they are considered. Therefore, the adaptive actions of a human administrator present a degree of uncertainty that is not present when the managed system is adapted by an autonomic manager as represented in figure 4.1. Eliminating this source of uncertainty is a motivating factor in the deployment of autonomic management systems, see [24] as is represented in figure 4.2. However, eliminating this source of uncertainty, and the resulting increase in predictability, can be exploited to partially enable an automated approach to meta-management. To understand how this can be exploited, it is first necessary to characterize the process of adapting a managed system.

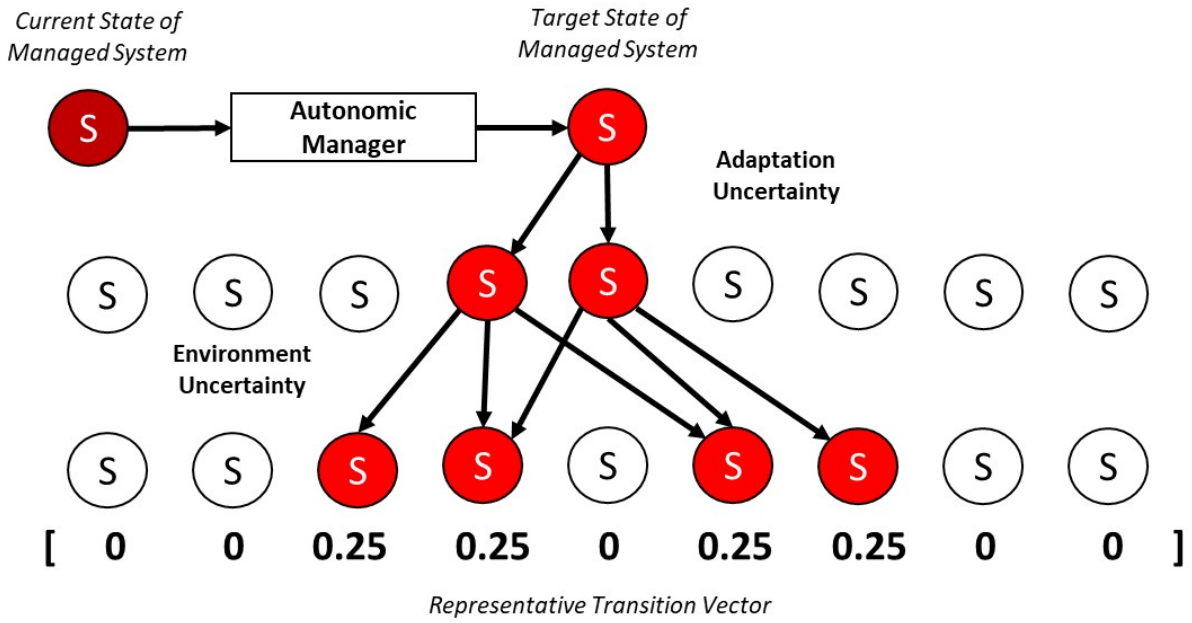


Figure 4.2: Exemplar Representation of Uncertainty in Autonomic Manager-based Adaptation

One method of characterizing the process of adapting a managed system with the results subject to uncertainty is by representing the process of transitioning the system from one state to another as a Discrete Time Markov Chain (DTMC). A DTMC is a stochastic process $(Z_n)_{n\in\mathbb{N}}$ taking values from the finite set of all possible states of the system, S. This process has the property that for all $n \geq 1$ the probability distribution of Z_{n+1} is determined by the state Z_n of the process at time n, and does not depend on the past values of Z_k for $k \leq n-1$, known as the *Markov property*. If $i, j \in S$ and E is the finite set of all possible states of the environment and $e \in E$. Then the probability of moving from state i to state j is defined as:

$$P_{i,j} := \mathbb{P}(Z_{n+1} = j | Z_n = i, e_n) \tag{4.1}$$

The result of equation 4.1 can be encoded into a matrix indexed by $S^2 = S \times S$ which is referred to as the *transition matrix*:

$$[P_{i,j}]_{i,j\in S} = [\mathbb{P}(Z_{n+1} = j | Z_n = i, e_n)]_{i,j\in S^2, e\in E} \tag{4.2}$$

Each row of the *transition matrix* has the property that $\sum_{j\in S} P_{i,j} = 1$.

Representative Transition Matrix

0.025	0.100	0.125	0.125	0.250	0.125	0.125	0.100	0.025
0.010	0.140	0.200	0.200	0.300	0.125	0.050	0.050	0.050
0.030	0.050	0.250	0.500	0.070	0.025	0.025	0.025	0.025
0.250	0.250	0.125	0.125	0.100	0.100	0.050	0.000	0.000
0.000	0.125	0.125	0.125	0.125	0.125	0.125	0.250	0.250
0.000	0.333	0.334	0.333	0.000	0.000	0.000	0.000	0.000
0.110	0.110	0.110	0.110	0.120	0.110	0.110	0.110	0.110
0.250	0.250	0.250	0.250	0.000	0.000	0.000	0.000	0.000
0.050	0.050	0.100	0.150	0.300	0.150	0.100	0.050	0.050

Figure 4.3: Exemplar Transition Matrix for Human-based Adaptation

While unexpected conditions can affect the adaptive outcomes of any entity responsible for adaptation actions, human based adaptation results in the variance or dispersion of potential adaptation outcomes in the *transition matrix* to be higher than if an autonomic manager is deployed, as represented in figure 4.3. This is because the primary goal of deploying an autonomic manager is to increase the predictability in the adaptation results by eliminating the human's inability to consistently consider each of the relevant dimensions and options with the same weight of importance. This increase in predictability results in the lower variance amongst the possible outcomes of adaptation actions and is a key difference in the predictability between human-centric and autonomic manager based adaptation.

Representative Transition Matrix

0.000	0.000	0.250	0.250	0.000	0.250	0.250	0.000	0.000
0.000	0.000	0.250	0.250	0.000	0.250	0.250	0.000	0.000
0.000	0.000	0.250	0.250	0.000	0.250	0.250	0.000	0.000
0.333	0.667	0.333	0.000	0.000	0.000	0.000	0.000	0.000
0.333	0.667	0.333	0.000	0.000	0.000	0.000	0.000	0.000
0.333	0.667	0.333	0.000	0.000	0.000	0.000	0.000	0.000
0.000	0.000	0.100	0.200	0.400	0.200	0.100	0.000	0.000
0.000	0.000	0.100	0.200	0.400	0.200	0.100	0.000	0.000
0.000	0.000	0.100	0.200	0.400	0.200	0.100	0.000	0.000

Figure 4.4: Exemplar Transition Matrix for Manager-based Adaptation

This reduction in the variance of the potential outcomes of adaptive behavior can be leveraged to partially enable an automated approach to meta-management by creating a smaller set of potential resulting states that need to be specified as a result of actions taken by the autonomic manager, as represented in figure 4.4 and highlighted by the red boxes. This enables the practical

creation of a specification of the autonomic behavior of a subsystem, referred to as an adaptation policy by allowing for specification of only the groups of non-zero entries as opposed to the potentially complete set of potential probabilities. See section 5.1 for additional information.

Abstraction of the Underlying Managed Systems

The introduction of the autonomic control system abstracts the managed system and, while the functional purpose of each subsystem is diverse, the fact that each of them is autonomic provides a homogeneous abstraction of each subsystem that allows two assumptions:

> **Assumption 1:** Each autonomic subsystem has a set of configuration options that can tune the behavior of the subsystem to operate within a range of behaviors.

The autonomic configuration options are the method by which human administrators establish the organizational and business preferences/tradeoffs and constrain the behaviors of the autonomic manager for each subsystem. In the exemplar scenario, the maximum cost constraint is an autonomic configuration option that limits the ability of the shopping cart subsystem to add servers to improve the 'Average Page Response Time'. If an autonomic manager adjusts the 'Maximum Cost' constraint, the autonomic subsystem might have a different set of adaptation strategies available. For example, if the 'Maximum Cost' was increased, the autonomic subsystem might be able to add additional servers. The autonomic configuration options are also the method by which a meta-manager would adjust the behavior of the subsystems using a specialized type of adaptation tactic, referred to as a *meta-tactic*.

> **Assumption 2:** The states of the system that could result from adaptive behavior can be specified for any given state of the environment and managed system under any configuration parameters.

The configuration options, state of the managed system, and the state of the environment influence the adaptation strategy that is deployed, but the individual autonomic managers must attempt to select the 'best' adaptation strategy available. To do this, the adaptation manager evaluates the targeted state of the managed subsystem that is the desired result of the application of each adaptation strategy and determines its ability to meet the QoS objectives of the system. The adaptation strategy that produces a state that will best meet the QoS objectives of the subsystem is considered the 'best' adaptation strategy and is deployed. As a consequence of the predictability gained from the introduction of an autonomic manager, if the same set of conditions were to occur on two different occasions, the autonomic manager would select the same adaptation strategy in both circumstances.

Therefore, if one is able to enumerate the states of the managed system and the states of the environment, within the simplified state space defined for each, it is also possible to predetermine which adaptation strategy would be deployed by the autonomic manager and the possible new states of the managed system given the state of the managed system, the state of the environment, and the configuration of the autonomic manager. This specification is referred to as the adaptation policy and is represented as a function:

$$P(c, e, s) \rightarrow \eta \tag{4.3}$$

where:

- E is the finite set of states of the environment that can be elaborated from the autonomic system environment model, $e \in E$, is a *state*
- S is the finite set of states of the managed system that can be elaborated from the autonomic system model, $s \in S$, is a *state*
- C is the finite set of possible configurations and $c \in C$
- P is the adaptation policy of the adaptation manager which requires a state of the configuration, c, a state of the environment, e, and a state of the managed system, s
- η is the *transition vector* for the Z_{n+1} distribution of the probabilities of transitioning from the current state of the managed system, $[P_{s,j}]_{s,j \in S}$.

For the purposes of the automated approach to meta-management in this thesis it is assumed that the current state of the environment, e, and the current state of the managed system, s, has a very low degree of variance and as such only the represented state of both is considered. This causes the *transition matrix* to reduce to a *transition vector*, η_s, corresponding to a row of the *transition matrix*, $[P_{s,j}]_{s,j \in S}$. It is also assumed that each autonomic subsystem will behave as specified in the adaptation policy. For a discussion of this assumption, please see section 11.1.

The three key advantages of autonomic systems, the simplification of the state space, reduction of variance in the outcomes of adaptive actions, and the abstraction of the underlying managed system enable the creation of adaptation policies and work together to make an automated approach to meta-management possible and practical. However, there exists additional opportunities that further enable the computational scalability and the effectiveness of an automated approach to meta-management: (1) the choice of analysis method for meta-analysis and meta-synthesis and (2) the definition of a global system state, and (3) global knowledge.

Choice of Analysis Method for Meta-Strategy Synthesis

By providing the information about the autonomic behavior of each of the subsystems, the adaptation policies can be used to perform a variety of analyses to examine multiple potential scenarios to determine which configuration for each autonomic subsystem would improve the quality of service (QoS) for the collection of autonomic subsystems.

As already introduced, the adaptation policies represent a discrete time Markov chain, but the adaptation policies can also represent a closely related *Markov Decision Process* (MDP). In [11], a MDP is defined as a tuple, ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{T}$, $\mathcal{R}$) where $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, $\mathcal{T}$ is a transition function, $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$, and $\mathcal{R}$ is a reward function, $\mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R}$. Solving the MDP consists of finding a policy, $\tau : \mathcal{S} \rightarrow \mathcal{A}$, which determines the agent's actions to maximize the reward function.

An adaptation policy defines the state space, $\mathcal{S}$, of the MDP as $E \times S \times C$ and the transition function, $\mathcal{T}$, as η, from equation 4.3. However, the adaptation policy does not define the actions,

$\mathcal{A}$, of the autonomic manager. Instead, this approach to meta-management recognizes that the actions of the autonomic manager will result in a new state of the managed system which maximizes the reward function established for that autonomic manager; $\tau : (\mathcal{S} \rightarrow \mathcal{A}) \rightarrow \mathcal{S}_{max}$, where $\mathcal{S}_{max}$ maximizes $\mathcal{R}$.

Therefore, an adaptation policy defines the policy function of a MDP as $\tau : \mathcal{S} \rightarrow \mathcal{S}_{max}$ where $\mathcal{S} = E \times S \times C$. Focusing on the pre and post adaptation states, and not on the actions or reward function, provides an abstraction of the autonomic processes that (1) partially enables a practical specification of the adaptive behaviors of each subsystem, see chapter 5, and (2) allows for a set of available analysis methods that can be used to compose the adaptation policies into a single MDP that is used to synthesize a plan of changes to the individual subsystems to improve their performance against a global quality objective. Therefore, this automated approach to meta-management does not mandate a specific type of analysis to determine the best configuration for each subsystem.

However, the choice of analysis technique used to synthesize a plan of changes to the configurations of the autonomic subsystems, referred to as a meta-strategy, is a critical task to ensure the effectiveness of the meta-manager as the timeliness, assurance, and tractability properties of each technique should align, as best as possible, to the requirements of the operating context. For example, if a particular synthesis technique can provide a high degree of assurance through an exhaustive exploration of the state space (e.g., a stochastic multi-player game) for a collection of autonomic systems with large configuration spaces, then it might require too much time to synthesize an appropriate meta-strategy. This would make the technique inappropriate for that specific context. However, that same synthesis technique might be preferred in a context in which more time is available to perform the strategy synthesis and a higher degree of assurance is required (e.g., an electrical utility grid). Chapter 6 will provide more information on meta-analysis and meta-synthesis techniques and the selection of an appropriate technique is addressed in each of the case studies presented in chapters 7, 8, and 9.

Definition of Global System State

Similar to the *state* for the individual autonomic subsystems, the meta-manager also defines a *state* that includes features beyond what is available in the states of the individual autonomic subsystems and is known only at the global level. These global features can be an amalgamation of properties from the individual subsystem states. In the exemplar scenario, the complete round trip time for a user's interaction with the shopping cart system would require adding the response time for the shopping cart system, the middleware common services, and the data services. The global features can also be information that is not available to any one autonomic subsystem. In the exemplar scenario, the total maximum cost constraint for the entire systems could be an important piece of information that is not available to any autonomic subsystem. The global system state is represented as g and a global system *transition vector*, $\pi := [P_{g,j}]_{g,j \in \mathcal{G}}$, which is the probability of moving from the current state of the global system to any potential state of of the global system due to changes to the configuration of autonomic subsystems through the deployment of a meta-strategy.

A global system state allows for additional optimizations to improve the tractability of the analysis for meta-strategy synthesis. For example, the effectiveness of the meta-manager can be

improved by allowing the meta-manager to consider optimizations against a global objective instead of just improving instantaneous global aggregate utility. For example, in a security context, it might be in the best interest of the system to hold the attention of the attacker by sacrificing the currently compromised subsystem to allow time for the other systems to mitigate the threat to themselves [79, 92]. Instead of improving global aggregate utility by mitigating the threat immediately and increasing the raw utility of the collection, the meta-manager improves against a global objective, maintaining security, even if that results in a lower utility score in the near term.

Global Knowledge

In addition to the adaptation policies, the meta-manager also has additional information, referred to as *global knowledge*, about the subsystems and the environments that might be only partially known to the subsystems because it is a property of global system state or the global environment in which it operates. There are at least two types of global knowledge: (1) information on the interrelationships between individual subsystems, local environments, and global properties and (2) constraints on individual subsystems and global state.

In the exemplar scenario, there is a relationship between subsystems where the request load on the middle tier services is correlated with the user load on the shopping cart system. This is a consequence of the architectural structure of the collection of autonomic systems. Similar interrelationships can also exist between local environments and elements that are part of the global state. This architectural structure and the resulting interrelationships can be exploited by the meta-manager to better assess the current and future states of the collection of autonomic systems and how it will perform against global quality objectives.

Additionally, constraints might also exist on the individual subsystems and elements of the global state as a result of the architectural structure or operations of the collection of autonomic systems. In the exemplar scenario, there is a constraint that defines the maximum cost the complete collection of autonomic systems can utilize. While each of the individual subsystems is also likely to have a maximum cost constraint, the operations of the collection of autonomic systems might need to further relax or restrict this constraint to improve the performance against the QoS objective.

Global knowledge is represented as a set of functions, $\mathcal{K}$. The first type of function represents the relationships between elements of the autonomic systems represented by the function, $k(x) \rightarrow x'$, where x, x' are *states*. These functions influence the analysis of the meta-manager by determining the 'best' state of the environment or managed system to be used in strategy synthesis. Specifically, the *state* of the environment for a subsystem, e_i, the *state* of the managed system for a subsystem, s_i, and the global *state*, g, in equation 4.7 are defined as:

$$e_i = \delta(e_n, \mathcal{K}_E)_{e_n \in E_j} \tag{4.4}$$

and

$$s_i = \delta(s_n, \mathcal{K}_S)_{s_n \in S_j} \tag{4.5}$$

and

$$g = \delta(g_n, \mathcal{K}_G)_{g_n \in G} \tag{4.6}$$

The function δ accepts an environment state for subsystem j, managed system state for subsystem j, or global state at step n and the relevant subset of global knowledge functions, $\mathcal{K}_E$, where $\forall k \in \mathcal{K}_E, x, x' \in E_j$ for states of the environment or $\mathcal{K}_S$, where $\forall k \in K_S, x, x' \in S_j$ for states of the managed systems, or $\mathcal{K}_G$, where $\forall k \in K_G, x, x' \in G$ for global system states where $E_j \in \mathcal{E}$, the set of all sets of possible states of the environments and $S_j \in \mathcal{S}$, the set of all sets of possible states of the managed systems, and subsystem i can, but does not have to, be the same as subsystem j.

The second type of global knowledge are the constraints placed on the managed systems and global states and are represented by a function, $k(X) \rightarrow X'$, where $X \in \{S_i, G\}$. These functions influence the analysis of the meta-manager by determining the set of available states of a managed subsystem, S_i, or the global state, G, that are present in equations 4.5 and 4.6.

Definition of Meta-Manager

With the definitions of an adaptation policy, P, and global knowledge, $\mathcal{K}$, the meta-manager is defined as a tuple $\mathcal{M}(\mathcal{G}, \mathcal{U}, \mathcal{P}, \mathcal{C}, \mathcal{E}, \mathcal{S}, \mathcal{K})$ where:

- $\mathcal{G}$ is the finite set of all states of the global system that can be elaborated from the global system model, $g \in \mathcal{G}$, is a *state*,
- $\mathcal{U}$ is the global utility function where $\mathcal{U}(\pi) \rightarrow [0..1]$ where π is the global *transition vector*, $[P_{g,j}]_{g,j \in \mathcal{G}}$,
- $\mathcal{P}$ is the set of all adaptation policies, P,
- $\mathcal{C}$ is the set of all sets of possible configurations for the subsystems, C,
- $\mathcal{E}$ is the set of all sets of possible states of the environment for the subsystems, E,
- $\mathcal{S}$ is the set of all sets of possible states of the managed system, S,
- $\mathcal{K}$ is a set of functions, $k \in \mathcal{K}$, representing the global knowledge.

The adaptation policies, P, are critical to the ability of the meta-manager to improve the performance of a collection of autonomic systems against a global QoS objective. Specifically, the goal of the meta-manager is to determine which configuration, $c \in C_i$, for a specific subsystem, i, will maximize the global utility function, $\mathcal{U}$, using a state of the environment, $e_i \in E$, a state of the managed system, $s_i \in S$, and the adaptation policy, $P_i(c_i, e_i, s_i)$. Formally this can be stated as:

$$\forall P_i \in \mathcal{P}; \underset{c \in C_i}{\operatorname{argmax}} \mathcal{U}(\lambda(P_i(c, e_i, s_i), g)) \tag{4.7}$$

where C_i is the set of all possible configurations for subsystem i, λ is a function that returns the global *transition vector*, π, given the *transition vector*, η_i, for subsystem i and the global state of the collection of autonomic systems, $g \in G$.

4.1 Meta-MAPE-K Loop

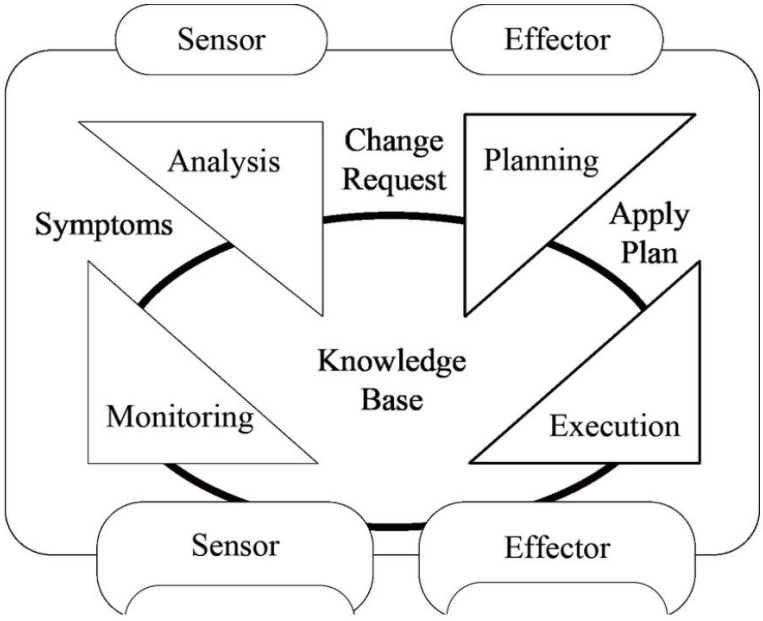


Figure 4.5: MAPE-K Diagram [57]

The automated approach to meta-management presented in this thesis must be implemented in a principled manner to ensure reusability across different types of collections of autonomic systems. This is accomplished by creating a version of the MAPE-K [57] control loop specialized for the purposes of meta-management. This section details the similarities and differences of implementations between the MAPE-K and Meta-MAPE-K control loops at both design time and run time for each of the five components: knowledge(K), monitoring(M), analysis(A), planning(P), and execution(E) in the context of the exemplar scenario presented in chapter 2.

Meta-Knowledge

The purpose of the knowledge component in the original MAPE-K loop [57] is to enable the data sharing and communications amongst the components of the MAPE-K feedback loop and serves a similar critical purpose in the the Meta-MAPE-K control loop. The Meta-Knowledge component of the Meta-MAPE-K control loop is responsible for, at least, four distinct pieces of knowledge: (1) the adaptation policies for each of the autonomic subsystems, (2) models for each of the autonomic subsystems, global system, and local environments, (3) the global utility function, and (4) the global knowledge.

Adaptation Policies for Autonomic Subsystems

Adaptation policies are the specification of the states of the autonomic subsystem that could be the result of adaptive behavior based upon the state of the environment, the state of the managed

system, and the configuration of the autonomic manager. For example, if the user load on the shopping cart system was to cause a degradation in the 'Average Page Response Time' due to changes in the local environment, the autonomic manager would evaluate the current state of the managed system, specifically the number of servers currently deployed, to determine if the system was running at the maximum configured cost, an element of the configuration of the autonomic manager. Based upon that evaluation, the autonomic manager would make the choice to do one of three things: (1) take no action, (2) deploy additional servers, or (3) reduce content fidelity. The expected behavior of the autonomic manager in any given situation is the adaptation policy.

At run time, the adaptation policies are what enable the meta-manager to reason about what changes to the configuration of each autonomic subsystem, if any, can be made to improve collective performance against the global QoS objectives. For example, by examining the adaptation policies of the shopping cart system, the meta-manager can examine what the behavior of the autonomic manager would be if the meta-manager adjusted the maximum configured cost the system was allowed to operate under. This might have the net result of the shopping cart system deploying additional servers, which improves the 'Average Page Response Time' and, consequently, the collective performance of the collection of autonomic systems.

Adaptation policies are created at design time and are part of the configuration of the meta-manager and updated as required. For example, the organization might update the utility function for the automated subsystem to reflect new organizational priorities leading to a new adaptation policy. An adaptation policy is typically created and maintained by human administrators and is addressed as part of this thesis through the use of a domain specific language, referred to as SEAM, see chapter 5.

Models for Autonomic Subsystems, Global *State*, and Local Environments

A meta-manager is dependent upon the current states of both the local environment and the managed system for each subsystem as well as the current global state of the collection of autonomic systems. As represented in equation 4.7, these run-time models of the systems are abstractions of the running systems and are defined by the observable properties (e.g., server count of the managed system or the user load on the shopping cart system) and the QoS objectives (e.g., page response time) relevant for improving the performance of the collection of autonomic systems.

The determination of which observable variables and QoS objectives are included in the run-time models for each subsystem and the global state is done by human administrators during the implementation of the meta-manager. The selection of these elements and the scope of their potential values is important as it partially determines the computational complexity the meta-manager must handle to determine the best adaptation options for the collection of autonomic systems. The selection of the elements also strongly influences the level of effort required by a human to specify the adaptation policies for each of the autonomic subsystems. See section 5.1 for additional information.

It is also possible for the *knowledge* component of the Meta-MAPE-K control loop to contain other run-time models that are not required, but can improve the ability of the meta-manager to improve the QoS for the collection of autonomic systems. For example, historical models for each of the autonomic subsystems and/or local environments can be used to predict the behavior

of those entities at later time steps leading to potentially improved meta-strategy synthesis and improvement against the global QoS objectives. For example, a historical model of the local environment for the shopping cart system might allow the meta-manager to predict the seasonality of the user load and synthesize a plan to proactively adjust the configurations of the subsystems to improve or maintain the QoS properties of the collections of autonomic systems.

Global Utility Function

The global utility function provides the mechanism for the meta-manager to determine how 'good' any given state of the system is by providing a 'score' between 0 and 1. This allows the meta-manager to examine potential alternative states that might result from changes to the configuration of the autonomic subsystems to determine if any of those changes will result in an improvement in the global utility score. If an improvement is possible, the adaptation strategy that leads to the improved score is deployed by the meta-manager. If no improvement is possible, then no changes are deployed by the meta-manager. The global utility function is a required piece of the *knowledge* component of the Meta-MAPE-K loop.

Commonly, global utility functions in adaptive systems are built upon expected utility theory [9]. However, there are several other types of utility that can be used to achieve different types of organizational objectives. For example, a rules based approach can be used to implement a set of constraints for different conditions in which the goal of the meta-manager is to ensure that a specific invariant is satisfied. Another option is to use rules to determine which utility function should be used based upon different conditions in the collection of autonomic systems. The AWS Shopping Cart case study presented in chapter 7 provides an example of this. The examination of the different types of utility theory and their potential combinations is not a topic of this thesis, but more information can be found in [50].

Global utility functions are established by human administrators at design time and reflect the priorities of the organization. For example, in the shopping cart system a human administrator could create a utility function that weighs the 'Average Page Response Time' higher than the cost of the system to operate. This would reflect an organizational preference to ensure a higher quality of service for the system ahead of ensuring the lowest possible cost to operate.

Global Knowledge

Global knowledge is the information that is primarily known at the global level of the collection of autonomic systems. Global knowledge includes information on the interrelationships between the individual autonomic subsystems, interrelationships between the individual local environments, and any constraints under which the collection of autonomic systems must operate. For example, the 'Average Page Response Time' of the shopping cart system is dependent upon the performance of both the middleware common services and the data services system. The global knowledge for the exemplar collection of autonomic systems would capture these interdependencies through an equation that relates the 'Average Page Response Time' of the shopping cart system to the key performance metrics of the middleware and data systems. The meta-manager could use this information to make more accurate predictions of the future state of the shopping

cart system which would improve the effectiveness of both the meta-analysis and meta-planning phases of the Meta-MAPE-K loop.

During the implementation of the meta-manager, the global knowledge is specified by human administrators based upon their knowledge of the context in which the meta-manager is being deployed.

Meta-Monitor

The meta-monitor phase of the Meta-MAPE-K loop is responsible for gathering and pre-processing relevant contextual information from entities in the execution environment and updates the run time models for each of the autonomic subsystems, the local environments, and the global system state. Therefore, at run time, it is similar in function and purpose as the monitoring phase of the original MAPE-K loop [68], but with one advantage. The primary difference is that all the managed resources in the execution environment of the Meta-MAPE-K loop are all autonomic subsystems. In some contexts, like the Google Control Plane case study presented in chapter 8, this allows for a degree of abstraction and reuse of the meta-monitor objects.

Meta-Analysis & Meta-Planning

The analysis phase of a standard MAPE-K loop is responsible for supporting the decision making on the necessity of adaptation and the planning phase is responsible for structuring a plan of actions to make the necessary changes. This sequential two-step process in making adaptation decisions is broadly applicable to *reactive adaptation* in which the autonomic control system reacts after QoS properties of the managed system have already degraded [71, 94].

However, another approach is *proactive adaptation* [71, 94] in which the autonomic control system adapts the managed system in anticipation of changes in conditions in the local environment and/or managed system. In this model of autonomic control the differences between the analysis and planning phases of the MAPE-K loop are not distinct nor sequential. As discussed in [82], it is often necessary to examine what the net benefit of deploying an adaptation strategy would be over time to determine if it is worth adapting at all.

A meta-manager could operate in either a *reactive* or *proactive* manner as needed to meet the requirements of the context to maintain a homeostatic system state and continuously improve the collective state of the collection of autonomic systems under management. Therefore, for the purposes of the Meta-MAPE-K loop, it is expected that the analysis and planning phases are performed jointly.

The choice of the meta-manager acting in a *reactive* or *proactive* manner is one decision point in the implementation of the analysis and planning phase of a Meta-MAPE-K loop. Another decision point is the choice of technique to synthesize an adaptation strategy. In an implementation of a standard MAPE-K loop, the choice of plan synthesis technique influences the quality of the adaptation strategies deployed, which consequently impacts the ability of the autonomic manager to meet the QoS objectives of the managed system. However, since a Meta-MAPE-K loop is specialized for the control of autonomic systems, as represented by their adaptation policies, a different set of criteria must be used in the evaluation of plan synthesis techniques.

The criteria for the evaluation of a strategy synthesis technique for meta-management should include the level of assurance required by the context, the computational scalability of the technique, and the timeliness of the analysis. These properties are important to ensure that the meta-manager can generate a plan that will meet the standard requirements of the context in which it is operating. For example, the administrators of a web based system like the one presented in chapter 2, might choose a technique that provides less assurance and less scalability as long as the result is generated quickly because the consequences of a sub-optimal plan are minimal. However, the administrator of an electrical grid might prefer a technique that provides a much higher degree of assurance and scalability and takes longer to generate the adaptation strategy as the consequences of a sub-optimal plan could be disastrous.

As will be detailed in chapter 5, it is possible to practically specify the complete state space for the managed system, the environment, and the configuration options of the autonomic subsystem. In a context in which that can be done, it is also possible to pre-generate the meta-strategy offline, significantly reducing the importance of the computational scalability of the analysis and synthesis technique. However, in contexts in which the specification of the state space is not possible or impractical, then the meta-analysis and meta-synthesis technique likely needs to be dynamically and repreatly run to determine the most appropriate meta-strategy. Additionally, in some edge case contexts, the offine generation of the meta-adaptation plan could yield a result that is too large to search and/or store due the complexity of the adaptation policies for the subsystems (e.g., the size of the state space and/or number of adaptation options) and the calculation of the meta-strategy is actually more timely. In contrast, the adaptation policies for the autonomic subsystems, for the purposes of this thesis, are considered static during the meta-analysis and meta-planning phases because those adaptation policies are most commonly slowly changing, requiring only infrequent updates. However, there are contexts in which this assertion may not hold. Please see chapter 11 for a more detailed discussion of this topic.

The individual plan synthesis techniques can also have considerations unique to the implementation of a Meta-MAPE-K loop. For example, if the administrator will be using a stochastic multi-player game (SMG) [110], how the adaptation policies for the individual autonomic subsystems are composed into the specification of the game can have a significant affect on the assurance, timeliness, and scalability of the approach. If each individual autonomic subsystem and local environment is its own player, the analysis will have a potentially computationally intractable state space. If the local environments and autonomic subsystems are consolidated into coalitions, the state space might become computationally tractable but not provide as high a degree of assurance because the model is less representative of the actual collection of autonomic systems. Further, when the individual autonomic systems share a high degree of similarity, it might be possible to limit the SMG to considering only one adaptation policy from one autonomic subsystem and assume that the changes that are appropriate for one autonomic subsystem are appropriate for all. Guidance on the methods, considerations, and scaling techniques is a topic of this thesis and more information can be found in chapter 6.

Meta-Execution

The execution phase of the Meta-MAPE-K loop implements the plan of action that was generated by the planner with the goal of changing the configuration of the relevant autonomic managers

so they may improve the performance of their managed subsystem against the global quality objectives. This functional purpose does not significantly vary in purpose from the original MAPE-K loop [57]. However, the meta-execution phase operates over a known type of managed component, specifically the individual autonomic subsystems.

Chapter 5

The SEAM Language

The automated approach to meta-management presented in chapter 4, defines a set of requirements for the creation of a language specialized to the needs of meta-management. Specifically, the language needs to provide methods of specifying:

- The state spaces and models for the managed system, the environment, the configuration of the autonomic manager for each autonomic subsystem,
- The state space and model for the global system state,
- The set of initial states and the states of the managed systems that are the result of adaptive actions,
- A probability distribution or explicit probabilities for the resulting states of adaptive actions,
- For which states a set of autonomic behaviors is applicable,
- The global knowledge, specifically, correlations and constraints,
- An adaptation policy,
- The global utility function using either expected utility function or a rules-based approach.

Many of these requirements have been previously elaborated on, however, an important item to note is that the formula defined as part of the global utility definition, see equation 4.7, is assumed to be used as part of an expected utility calculation [9]. This has the consequence that the formula is used to score the states that could be the result of an adaptive action and aggregate those scores weighted by the probability of the individual state occurring. This is necessary to support collections of autonomic systems in which the primary goal of the meta-manager is to maintain and improve homeostatic operations. However, a rules based utility [9], in which the utility is dependent upon a set of invariant conditions is also necessary to support collections of autonomic systems in which the primary goal of the meta-manager is to mitigate the effects of extreme operational conditions.

This chapter presents SEAM, the domain-specific language created to address these requirements and facilitate the practical implementation of a meta-manager to improve a collection of autonomic systems against the defined QoS objectives. As a top-level enumeration, the following is the complete list of the elements defined in SEAM and which base elements are utilized by which elements:

- Adaptation Policy
  - Predicate
    Used to identify the set of initial and resulting states
  - Probability Distribution
    Used to specify the probabilities of the resulting states
- Global Utility Function
  - Predicate
    Used to specify the applicable states of the utility function and create a rules based approach
  - Formula
    Used to specify the utility function
- Global Knowledge
  - Predicate
    Used to specify the applicable states of the global knowledge
  - Relationship
    Used to specify global knowledge function
- Subsystem
  - Current State and Current Configuration
    Used to define the current state models of the autonomic subsystem
  - Property
    Used to define the elements of the state space
  - State Space
    Used to specify the state space for the managed system, environment, and autonomic manager configuration
  - Environment
    Used to specify the current state, state space, and behavior of the environment
- MetaManager

SEAM is a declarative language based on the JavaScript Object Notation(JSON) [13] designed to handle the specific needs of meta-management. The JSON specification for SEAM is structured under a single root node, represented as $ (ASCII Decimal 36), yielding a single JSON namespace. This eases the identification of specific elements within the namespace and ease of use within many of the popular programming languages. Many programming languages, e.g., Java and Javascript, .NET (VB and C#), Python, C++, and PHP all have easily available libraries for the parsing, reading, and manipulation of JSON documents. Each of the elements within SEAM have a specific JSON structure and these elements are defined and an example of their implementation provided based upon the exemplar system presented in chapter 2. Further, SEAM also leverages the JSON Path [55] notion to reference individual properties and elements within the document. As there is no standardized syntax for embedding JSON Path statements within a JSON document, SEAM will use a number or pound sign, # (ASCII Decimal 35), to mark both the beginning and ending of the JSON Path statement

Additionally, to facilitate understanding of the specification of the individual elements, the definitions include placeholders for specific values that are enclosed within guillemets (« »). For example:

```
1 <<ExampleDescriptor>>
```

Listing 5.1: Specification Example

These statements, including the guillemets, are not part of SEAM syntax and are used to facilitate human understanding of the definitions, and are replaced by appropriate values at the time of implementation. To further facilitate understanding of the element specifications and examples, each begins at the root JSON node to demonstrate where the element falls within the JSON document and facilitate the use of JSON Path statements in the examples, comments to facilitate clarity are included in green, and the SEAM reserved keywords are highlighted in blue.

This chapter is organized as follows: section 5.1 details the definition of the elements required for the specification of the adaptation policy, section 5.2 details the definition of the global utility function, section 5.3 details the definition of the elements required for the specification of global knowledge, section 5.4 details the specification of a subsystem, and section 5.5 details the definition of elements required for the specification of the meta-manager.

5.1 Adaptation Policy

As discussed in chapter 4, the adaptive behavior of each autonomic subsystem is influenced by multiple factors including the state of the system under management, the state of the local environment, and the state of the configuration of the autonomic manager. The complexity of how each of these factors influence the adaptive behavior of a autonomic subsystem is what defines the probabilities of the transition matrix of the adaptive behavior of the autonomic subsystem and presents a substantial challenge to the practical specification of an adaptation policy. To address this challenge, SEAM defines two elements necessary to specify an adaptation policy: (1) a predicate that is used to identify the subsets of initial states and the subsets of the resulting states and (2) a probability distribution or explicit probabilities that define the probability of the each of the resulting states. However, to understand how these SEAM elements address the challenge of specifying the adaptation transition matrix, it is first necessary to characterize the structure of the information that influence the adaptation policy.

The base unit for the contextual information is a *state* which, for the purposes of SEAM, is a collection of properties each of which has an associated value that is part of a finite set of possible values. Therefore, a state of the environment can be formally defined as $e = \{x_0, ..., x_n\}$ where $x_i \in X_i$ where X_i is the finite set of possible values for x_i, representing the set of values a property of the environment can take. We can similarly define the state of the managed system as $s = \{y_0, ..., y_n\}$ where $y_i \in Y_i$ where Y_i is the set of possible values for y_i and the state of the configuration as $c = \{z_0, ..., z_n\}$ where $z_i \in Z_i$ where Z_i is the set of possible values for z_i. These *states* can be used to define three sets, (1) the finite set of possible *states* of the local environment, E, (2) the finite set of possible *states* of the managed system, S, and (3) the finite set of possible *states* of the configuration of the autonomic manager, C.

The autonomic manager of the subsystem determines if an adaptation is necessary by evaluating both the current state of the managed system and the current state of the local environment. Consequently, the set of possible states that the autonomic manager considers is a combination of the states of the environment local to the subsystem, E, and the states of the managed system, S. This results in a new set of possible states referred to as the *pre-adaptation set* defined as:

$$A = E \times S \tag{5.1}$$

If an adaptation is necessary, the autonomic manager synthesizes a plan to make changes to the managed system, referred to as an adaptation strategy, and executes that plan. If no adaptations are necessary then the autonomic manager takes no action. In either case, as elaborated in chapter 4, this process can be characterized as the autonomic manager selecting a state, s, of the managed system from the set of possible states of the managed system, S where $s \in S$. Additionally, the representation of the contextual information must also account for the set of possible states of the configuration, C, as the adaptive actions of the system are at least partially dependent upon the current state of that configuration, c where $c \in C$, as discussed in chapter 4. Therefore, the set of contextual information relevant to the specification of an adaptation policy is:

$$\Omega = C \times (E \times S) \times S \tag{5.2}$$

The combined set Ω represents the contextual information necessary to specify an adaptation policy, but the elaboration of an adaptation policy for an individual autonomic system requires a method of specifying which states of the configuration, C, relate to which states of the *pre-adaptation set*, $E \times S$, and the new state of the managed system, S. Referring back to the exemplar presented in chapter 2, the values of *Avg. Page Response Time* define the state space for the environment, E, the combination of values for *Server Count* and *Content Fidelity* define the state space of the managed system, S, and the combination of values of the *Maximum Cost* and *Capacity Buffer* define the state space of the configuration, C. The combination of the *Avg. Page Response Time*, *Server Count*, and *Content Fidelity* define the *pre-adaptation set*, $E \times S$, and the set of resulting states, S, changes depending on the value of the *Maximum Cost* which is part of the configuration of the autonomic manager.

In the specification of the adaptation policy, it is important to note that the adaptive actions the autonomic subsystem deploys as part of an adaptation strategy are not important. What is important is the combination of which states of the configuration, C, managed system, S, and the environment,E, resulted in what end state of the managed system, S. Therefore, one method of defining the relationship between these states is by specifying subsets of the combined set of textual information, Ω, in which each subset equates to one or more adaptation strategies.

A method of defining a subset is through the use of an indicator function [117]. An indicator function of a subset of a set is a function that maps the elements of a subset to one and all others to zero, defined as:

$$\mathbf{1}_X(x) := \begin{cases} 1 & \text{if } x \in X, \\ 0 & \text{if } x \notin X \end{cases} \tag{5.3}$$

To define the necessary subsets of C, it is necessary to define a set of indicator functions, $C_{\mathbf{1}} = \{c_{\mathbf{1}}(x)_0, ..., c_{\mathbf{1}}(x)_n\}$, where $x \in C$ with the properties that if $c_{\mathbf{1}}(x)_i = 1$ then $c_{\mathbf{1}}(x)_j = 0$ $\forall i, j \in C_{\mathbf{1}}$ where $i \neq j$ (uniqueness) and where $\forall x \in C$ there exists i such that $c_{\mathbf{1}}(x)_i = 1$ (completeness). A similar approach can be taken to defining subsets of the *pre-adaptation set* of states, $A = (E \times S)$. Specifically, $A_{\mathbf{1}} = \{a_{\mathbf{1}}(x)_0, ..., a_{\mathbf{1}}(x)_n\}$, where $x \in A$ with the property that $\forall x \in A$ there exists i such that $a_{\mathbf{1}}(x)_i = 1$ (completeness).

SEAM supports the specification of these functions through the use of predicates based upon boolean algebra [49] with negation (!), AND (&), OR (|), and XOR ($\wedge$) operators and comparison operators including greater then (>), less than (<), greater than equal to (>=), less than equal to (<=), equal to (=), and not equal to (!=).

```
1 <<JSON Path>> <<comparison operator>> <<value>> <<boolean operator>>
```

Listing 5.2: Predicate Specification

The following is an example of a predicate for the shopping cart exemplar presented in chapter 2.

```
1 #$.ShoppingCart.CurrentState.ServerCount# > 10
```

Listing 5.3: Predicate Example

Individual statements can be combined to form complex statements as necessary using boolean operators. For example:

```
1 #$.ShoppingCart.CurrentState.ServerCount# > 10 & #$.ShoppingCart.CurrentState.ContentFidelity# = "HIGH"
```

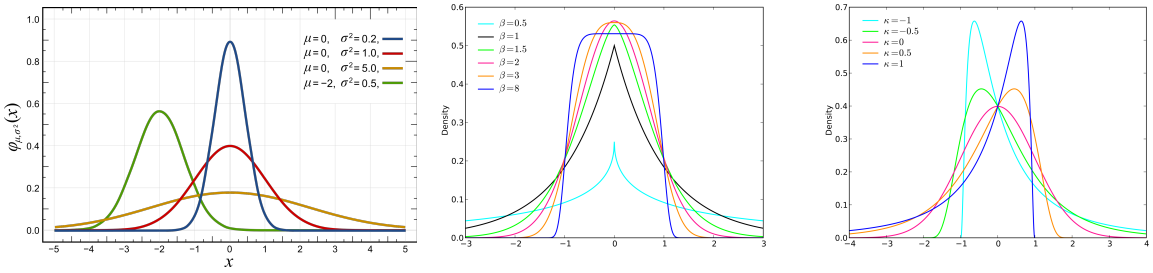
Listing 5.4: Multi-Statement Predicate Example

An indicator function, implemented as a predicate statement in SEAM, is appropriate when the subset of the states of the contextual information, Ω, can be precisely defined. However, because the results of adaptive actions have a degree of uncertainty or variance, a more generalized form of indicator function is required, specifically a membership function [127].

A membership function is defined as $m : U \rightarrow [0, 1]$ and instead of defining an element to be a member of a set with a binary 0 or 1, like an indicator function, a membership function gives a likelihood, between 0 and 1, that a particular element is included in a set. Using membership functions, $\eta_{\mathbf{1}}$, the subsets of the set of states of the managed system post-adaptation, H, can be defined as $H_{\mathbf{1}} = \{\eta_{\mathbf{1}}(x)_0, ..., \eta_{\mathbf{1}}(x)_n\}$, where $x \in S$ with the property that $\forall x \in S$ there exists i such that $\eta_{\mathbf{1}}(x)_i > 0$ (completeness) and S is the set of possible states of the managed system.

SEAM represents membership functions as a probability distribution embedded into the predicate statement. While there are many potential probability distributions that might be applicable to describing the outcomes of adaptive behavior (see [36] for a comprehensive list), for the purposes of this thesis it is expected that either a normal [62] distribution or the related generalized Gaussian distribution [84] would service the needs of a significant majority of applications in adaptive systems (see chapter 11 for more information).

A normal(N) or Gaussian probability distribution is a continuous probability distribution for a real-valued random variable with the probability density function:



((a)) Normal Distribution ((b)) Symmetric Gaussian ((c)) Asymmetric Gaussian

Figure 5.1: Examples of Probability Distributions

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \tag{5.4}$$

where μ is the mean or expectation of the distribution and σ is the standard deviation with σ^2 referred to as the variance. This probability distribution allows an administrator to specify a particular result of an adaptive action as the expected or most likely result and set the variance to indicate the amount of variation expected in that result. The relative simplicity of this distribution means that it can be specified using only the notation $N(\mu, \sigma^2)$.

The generalized Gaussian distribution (GGD) has two variations, the symmetric(SGGD) and asymmetric(AGGD)[85] probability distributions. The symmetric probability distribution function is:

$$f(x) = \frac{\beta}{2\alpha\Gamma(1/\beta)} e^{-(|x-\mu|/\alpha)^\beta} \tag{5.5}$$

where $\Gamma(a) = \int_0^\infty t^{a-1}e^{-t}\,dt$ and is referred to as the Gamma Function. This probability distribution allows for a wider variety of shapes in the symmetric curves than defines the probability distribution around the expected value. This probability distribution can be specified using the notation $SGGD(\mu, \alpha, \beta)$ where μ is the location or expectation, α is the scale or variance, and β is the shape of the distribution.

The asymmetric generalized Gaussian distribution is a combination of two different generalized Gaussian distribution functions, one on each side of the expected value. This allows for an administrator to specify the characterization of the uncertainty as a result of adaptive behavior in even more detail. The asymmetric probability distribution function is:

$$f(x) = \begin{cases} \frac{\beta}{(\alpha_1+\alpha_2)\Gamma(1/\beta)} e^{-[(-x+\mu)/\alpha_1]^\beta} & \text{if } x < \mu, \\ \frac{\beta}{(\alpha_1+\alpha_2)\Gamma(1/\beta)} e^{-[(x-\mu)/\alpha_2]^\beta} & \text{if } x \geq \mu \end{cases} \tag{5.6}$$

where μ is the expectation, α_1 is the variance to the left of the expectation, α_2 is the variance to the right of the expectation, β is the shape of the distribution, and $\Gamma(a)$ is the same

has previously defined. This probability distribution can be can be specified using the notation $AGGD(\mu, \alpha_1, \alpha_2, \beta)$. For the purposes of SEAM, it is assumed that each of the distributions are representing a single independent variable as the specification and proper use of multivariate probability distributions would be impractical for the purposes of specification, require more specialized training in statistics than is common with IT administrators, and have limited value in increasing the accuracy of the model.

The following is the SEAM specification for normal probability distribution, symmetric and asymmetric Gaussian distributions:

```
1 //Normal Distribution
2 <<JSON Path>> = N(<<mean>>,<<variance>>)
3 //Symmetric GGD
4 <<JSON Path>> = SGGD(<<mean>>,<<variance>>,<<shape>>)
5 //Asymmetric GGD
6 <<JSON Path>> = AGGD(<<mean>>,<<left>>,<<right>>,<<shape>>)
```

Listing 5.5: Distribution Specifications

Additionally, there are some situations in which the state of the attribute is subject to uncertainty, but cannot be easily described by a continuous probability distribution. For example, if a server instance is added to the shopping cart system as discussed in chapter 2, there is a 99% chance of that adaptive action occurring successfully, but a 1% chance of that action failing. There is no practical way of using any of the previously mentioned probability distributions to specify that situation. Therefore, SEAM provides the ability to define explicit probabilities on the potential values of an attribute. The specification for *explicit probabilities* is defined here:

```
1 //Explicit Probabilities
2 <<JSON Path>> = [<<probability>>|<<value>>,...,<<probability>>|<<value>>]
```

Listing 5.6: Explicit Probability Specification

The individual probabilities are specified as a decimal between 0 and 1 and the combined total must add to 1.

Each of the probability definitions offer practical options for the specification of uncertainty of adaptive behavior by a human administrator and is applied to the values of the individual properties that define a *state*. The following is an example of the use of the defined probability distributions using the exemplar shopping cart system presented in chapter 2:

```
1 //Normal Distribution
2 #$.ShoppingCart.CurrentState.AvgPageResponse# = N(2.8,1.0)
3 //Symmetric GGD
4 #$.ShoppingCart.CurrentState.AvgPageResponse# = SGGD(2.8,1.0,2)
5 //Asymmetric GGD
6 #$.ShoppingCart.CurrentState.AvgPageResponse# = AGGD(2.8,1.0,1.5,3)
7 //Explicit Probabilities
8 #$.ShoppingCart.CurrentState.AvgPageResponse# = [0.5|1,0.3|2,0.2|3]
```

Listing 5.7: Distribution Examples

In each of the first three cases the probability distribution is set around the mean or expected value of the $AvgPageResponse$ value of 2.8 seconds. In the last case a set of explicit probabilities are defined for the values 1, 2, and 3.

Predicates can also have a probability distribution defined for individual properties which is used to define the membership function for the resulting states in an adaptation policy. For example:

```
#$.ShoppingCart.CurrentState.AvgPageResponse# = N(2.8,1.0) & #$.ShoppingCart.CurrentState.ServerCount# =
    ↪ [0.99|10,0.01|9]
```

Listing 5.8: Predicate with Probability Distribution Example

With the use of predicates augmented with the option to define the probability distributions for individual properties, an *adaptation policy* can be specified as a collection of elements that defines the adaptation policy for a specific subsystem. The following is the specification of the *adaptation policy* element:

```
{ //Root Node
  "<<SubsystemName>>":
  {
    "AdaptationPolicies":
    [
        {
          "ConfigPredicate": "<<Config Predicate>>",
          "isDefault": "<<Is Default: True or False>>",
          "Behaviors": [
            {
              "StatePredicate":"<<State Predicate>>",
              "ResultState": "<<Result Predicate>>",
              "ConfigUpdate":"<<JSON Path>>"
            }
          ],
          "BehaviorModifiers": [
            {
              "StatePredicate":"<<State Predicate>>",
              "ResultState": "<<Result Predicate>>"
            }
          ]
        }
    ]
  }
}
```

Listing 5.9: Adpatation Policy Specification

The $AdaptationPolicies$ element is a collection of individual $AdaptationPolicy$ elements. The $AdaptationPolicy$ element consists of three elements. The first is the $ConfigPredicate$ predicate that defines which states of the configuration of the subsystem a particular $AdaptationPolicy$ element is applicable for. The second is the $isDefault$ element which is used to define which

of the $AdaptationPolicy$ elements is the default. The third is the $Behaviors$ element which is a collection of elements with three potential properties. The $StatePredicate$ is a ***predicate*** elements that defines the states of the managed system pre-adaptive action and the $ResultState$ is another *predicate* element that defines the expected states of the managed system post-adaptive action. The $ResultState$ is the predicate that would be likely to have a *probability distribution* defined as part of the $predicate$ element. The $ConfigUpdate$ element is only valid in the $AdaptationPolicies$ of the $MetaManger$ as they specify which elements of the $CurrentConfig$ of the $Subsystems$ can be updated by the $MetaManager$ to adjust the autonomic behavior of the subsystems. The $BehaviorModifiers$ is a collection of the same elements that define the $Behaviors$ element. However, the elements in the $BehaviorModifiers$ are used to change the behaviors of the $AdaptationPolicy$ element that is defined as the default. The $StatePredicate$ entry for the elements in the $BehaviorModifiers$ must exactly match an entry in the $Behaviors$ element of the default $AdaptationPolicy$. The $BehaviorModifiers$ allows an administrator to create an additional $AdaptationPolicy$ by only defining the changes as opposed to having to define again a complete $AdaptationPolicy$ element.

The following is an example of *AdaptationPolicy* elements as defined for the shopping cart subsystem as presented in chapter 2.

```
{ //Root Node
  "ShoppingCart":
  {
    "AdaptationPolicies":
    [
      {
        "ConfigPredicate":"#$.ShoppingCart.CurrentConfig.MaxiumumCost# = 2000 &
            ↪ #$.ShoppingCart.CurrentConfig.CapacityBuffer# = 10",
        "isDefault":"True",
        "Behaviors":
        [
          {
            "StatePredicate":"#$.ShoppingCart.CurrentState.AvgPageResponse# <= 2.5 &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# >= 10 &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# <=15",
            "ResultState":"#$.ShoppingCart.CurrentState.AvgPageResponse# = N(2.5,1) &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# = 16"
          },
          {
            "StatePredicate":"#$.ShoppingCart.CurrentState.AvgPageResponse# > 2.5 &
              ↪ #$.ShoppingCart.CurrentState.AvgPageResponse# <= 3.5 &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# >= 10 &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# <= 15",
            "ResultState":"#$.ShoppingCart.CurrentState.AvgPageResponse# = N(3.5,1) &
              ↪ #$.ShoppingCart.CurrentState.ServerCount# = 16"
          }
        ]
      },
      {
        "ConfigPredicate":"#$.ShoppingCart.CurrentConfig.MaxiumumCost# = 3000 &
            ↪ #$.ShoppingCart.CurrentConfig.CapacityBuffer# = 10",
```

```
23            "BehaviorModifiers":
24            [
25               {
26                  "StatePredicate":"#$.ShoppingCart.CurrentState.AvgPageResponse# <= 2.5 &
                      ↪ #$.ShoppingCart.CurrentState.ServerCount# >= 10 &
                      ↪ #$.ShoppingCart.CurrentState.ServerCount# <= 15",
27                  "ResultState":"#$.ShoppingCart.CurrentState.AvgPageResponse# = N(2.0, 1) &
                      ↪ #$.ShoppingCart.CurrentState.ServerCount# = 20"
28               }
29            ]
30         }
31      ]
32   }
33 }
```

Listing 5.10: Adpatation Policy Example

This example demonstrates the implementation of two *AdaptationPolicy* elements. The first, is the default *AdaptationPolicy* element which defines two behavior elements. The first behavior element defines the *state predicate* which specifies the *pre-adaptation set* of states where the *AvgPageResponse* is less than or equal to 2.5 seconds and the *ServerCount* is greater than or equal to 10 and less than or equal to 15. This specification matches the adaptation policy represented in the exemplar scenario presented in figure 2.2. It also defines the set of the resulting states of the managed system with the *ResultState* in which a normal probability distribution is defined over the values of the $AvgPageResponseTime$ with a mean of 2.5 seconds and a variance of 1 and where the *ServerCount* is equal to 16. The second *AdaptationPolicy* defines a behavior modification that updates the behaviors of the default *AdaptationPolicy*.

5.2 Global Utility Function

A SEAM *global utility* element is used to define the global utility function for the collection of autonomic systems and uses two SEAM elements: $Formula$ and $Predicate$.

A *formula* is used to create a mathematical expression to be evaluated. It uses the standard mathematical operators for addition (+), subtraction (-), multiplication (*), division (/), exponent ($\wedge$), and remainder($\%$) and parentheses to ensure the proper order of operations and JSON Paths to identify the necessary properties. The $Objective$ property can be set to either Min or Max and defines if the objective of the utility function should be maximized (i.e., a reward) or a minimized (i.e., a cost). The $Predicate$ defines the applicability of the utility function as defined previously in listing 5.2. If the $Predicate$ property is missing, then the global utility function is applicable in all states.

The following is an example of a formula for a global utility function for the exemplar system presented in chapter 2.

```
1 ((3 – #$.ShoppingCart.CurrentState.AvgPageResponse#) /3 )
```

Listing 5.11: Formula Example

For the purposes of a global utility function, the $Formula$ must result in a value between 0 and 1. The following is the specification for the *global utility* element:

```
{ //Root Node
  "MetaManager":
  {
    "GlobalUtility":
    [
      {
        "Predicate": "<<SEAMPredicate>>",
        "Formula": "<<SEAMFormula>>",
        "Objective": "<<Min|Max>>"
      }
    ]
  }
}
```

Listing 5.12: Global Utility Specification

The following is an example specification for a *global utility* element:

```
{ //Root Node
  "MetaManager":
  {
    "GlobalUtility":
    [
      {
        "Predicate":"#$.MetaManager.CurrentState.IsCompromised# = False",
        "Formula":"((3 – #$.ShoppingCart.CurrentState.AvgPageResponse#) /3)",
        "Objective": "Max"
      }
    ]
  }
}
```

Listing 5.13: Global Utility Example

In listing 5.13, the *Predicate* element defines the applicability criteria for the defined global utility function. In this case the defined global utility function is only applicable when *IsCompromised* is false. The *Formula* element defines the global utility function which scales the utility score by setting 3 seconds, or more, as the lowest utility score, 0, and any score less than 3 seconds increasing towards the best possible score of 1. The *Objective* element specifies that the utility function should be maximized as it is a reward as opposed to minimized in which it is commonly characterized as a cost.

5.3 Global Knowledge

As discussed in chapter 4, global knowledge is the additional information that is available to the meta-manager but might only be partially available to the individual subsystems. Examples of

this include interrelationships between subsystems in which the state of one subsystem can be influenced or impacted by the state of another (e.g., load on a web server can impact load on the database). Another example of global knowledge is a constraint that the collection of autonomic systems must respect (e.g., a maximum cost constraint or the minimum resources that must be in use).

The *GlobalKnowledge* element of SEAM has two properties. The first is an optional predicate that defines the applicability of the entry as defined in listing 5.2. In the event that a predicate is not defined, it is assumed that the entry is valid at all times. The second is a required *relationship* that defines the information that is being specified.

While other types of *relationship* entities are possible, SEAM defines two types of *relationship* entities: constraint and correlation. The constraint defines the relevant operating limits for the collection of autonomic systems. For example, referring to the exemplar presented in chapter 2, the maximum cost of the complete web system, not just the maximum cost of the individual autonomic subsystems. Constraints restrain the meta-synthesis technique by limiting which combinations of changes to the individual subsystems are valid. Correlations represent additional information about the structure of the collections of autonomic systems. For example, referring to the exemplar presented in chapter 2, the user load on the shopping cart subsystem directly influences the request load on the middleware common services which influences the query load on the data services system. These correlations can be used by the meta-synthesis technique to better predict the current and future states of the systems leveraging information that might not be available to the individual autonomic subsystems.

Constraint

A *constraint* establishes a boundary for a specific metric that is not to be violated. The constraint is expressed in the form of a predicate which allows an administrator to establish, potentially state dependent, rules that can constrain the potential options of the synthesis of meta-adaptation policies. Without this option available, the meta-manager could implement changes to the autonomic subsystems that put the collection of autonomic systems into an undesirable state.

The following is the specification for a constraint:

```
"Relation": {
   "Type": "Constraint",
   "Predicate":"<<predicate>>"
   }
```

Listing 5.14: Constraint Property Specification

The following is an example of a constraint based upon the shopping cart system presented in chapter 2 which says that the *ServerCount* for the *ShoppingCart* system should always be greater than or equal to 10:

```
"Relation": {
   "Type": "Constraint",
   "Predicate":"#$.ShoppingCart.CurrentState.ServerCount# >= 10"
```

```
4    }
```

Listing 5.15: Constraint Property Example

It is possible to constrain a specific property for a specific system by establishing a *constraint* relationship, but it is also possible to establish it by defining the scope of a critical element using a SEAM *property*. It is recommended to establish the physical boundaries of a critical element using a SEAM *property* and use a SEAM *constraint* to define organizational preferences that are potentially subject to change over time. This is due to the fact that the specified SEAM *property* elements also define the complete state space for the contextual information. If organizational preferences were established by altering the state space of the contextual information, it could result in the invalidation of defined predicates and, potentially, unexpected behavior from the meta-manager. However, if it is a complex constraint dependent upon multiple factors, then a SEAM *constraint* is the only option available.

Correlation

A SEAM *correlation* defines a relationship between two critical elements that share a dependency. The relationships between subsystems are expressed in the form of correlations between the properties of different subsystems. For example, an administrator could define a correlation between the user load on the shopping cart system and the query load on the database system. These correlations allow the meta-manager to have a more accurate evaluation of the state of each of the subsystems at both the current time step as well as future time steps. For example, if the meta-manager is expecting the environment to increase the user load on the shopping cart system in the next time step, the correlation between the user load and the query load on the database system allows for a more accurate state of the database subsystem which increases the effectiveness of the meta-manager by improving meta-strategy synthesis.

The following is the specification of the *correlation* property:

```
1 "Relation": {
2    "Type": "Correlation",
3    "Target": "<<JSONPath>>"
4    "Formula":"<<SEAMFormula>>",
5    "Timedelay":<<timedelay>>
6 }
```

Listing 5.16: Correlation Property Specification

The $Target$ defines the element that has a dependency on the elements defined in the $Formula$. The $Formula$, as specified in listing 5.11, defines the correlation between the critical elements in the collection of autonomic systems. The $Timedelay$ property of the *correlation* is how many time steps into the future the affect of the dependency between the elements can be observed. This can be used by some meta-synthesis techniques to understand how long it takes into the future to observe the effects of the correlation. The following is an example of the *correlation* relationship implemented for the collection of autonomic systems described in chapter 2:

```
"Relation": {
  "Type": "Correlation",
  "Target":"#$.MetaManager.CurrentState.TotalResponseTime#",
  "Formula":"1.2 * #$.ShoppingCart.CurrentState.AvgPageResponse# + 1.1 *
    ↪ #$.Middleware.CurrentState.ResponseTime#",
  "Timedelay":2
}
```

Listing 5.17: Correlation Property Example

The example demonstrates how to define the correlation between elements from different subsystems and the global system state. Specifically, the global $TotalResponseTime$ is defined as a correlation which is dependent upon the $AvgPageResponse$ from the Shopping Cart subsystem and the $ResponseTime$ from the Middleware Common Services subsystem. It further defines that the effects of this dependency should be observable 2 time steps into the future.

The *constraints* and *correlation* elements are used in the following specification of the *GlobalKnowledge* element:

```
{ //Root Node
  "MetaManager":
  {
    "GlobalKnowledge":
    [
      {
        "Predicate":"<<predicate>>",
        "Relation": {
          <<relation properties>>
        }
      }
    ]
  }
}
```

Listing 5.18: Global Knowledge Specification

The following is an example of a specification of a *GlobalKnowledge* element for the shopping cart system as presented in chapter 2:

```
{ //Root Node
  "MetaManager":
  {
    "GlobalKnowledge": [
      {
        "Predicate":" #$.MetaManager.CurrentState.IsCompromised = False#",
        "Relation": {
          "Type": "Correlation",
          "Target":"#$.MetaManager.CurrentState.TotalResponseTime#",
          "Formula":"1.2 * #$.ShoppingCart.CurrentState.AvgPageResponse# + 1.1 *
              ↪ #$.Middleware.CurrentState.ResponseTime#",
```

```
11              "Timedelay":"2"
12            }
13          },
14          {
15            "Relation": {
16              "Type": "Constraint",
17              "Predicate":"#$.ShoppingCart.CurrentState.ServerCount# >= 10"
18              }
19          }
20        ]
21    }
22 }
```

Listing 5.19: Global Knowledge Example

This example defines two entries in the *GlobalKnowledge* element. The first is a *correlation* element that defines a correlation between the $TotalResponseTime$ and the $AvgPageResponse$ from the shopping cart subsystem and $ResponseTime$ from the middleware subsystem. However, this *correlation* is only valid if the current value of $IsCompromised$ in the global current state is false. The second element is a *constraint* that defines that the $ServerCount$ for the shopping cart system should never be less than 10.

5.4 Subsystem

A SEAM *subsystem* element describes an individual autonomic subsystem and defines four elements and one property.

The first is the *CurrentState* element which represents the current values for the set of properties that describe the current state for the managed system. This is a requirement for the automated approach to meta-management. Similarly, the second is the *CurrentConfig* element which represents the current values for the set of properties that describe the current configuration for the autonomic manager and is also a requirement for the automated approach to meta-management. The third is the *StateSpace* element which contains two other elements, *Properties* and *Configuration*. Both *Properties* and *Configuration* define the set of properties, and their potential values, for the *CurrentState* and *CurrentConfig* elements. These elements, while not required for the approach to meta-management, allow for the definition of the state space for each subsystem. This can be exploited by the meta-manager to ensure each of the predicate statements are valid and the predicates address the complete state space. Additionally, the meta-manager would require this state space definition to support some meta-analysis and strategy synthesis techniques and in some other cases the meta-manager could use the state space definitions to optimize the meta-analysis and meta-synthesis technique in use. The fourth element is the *AdaptationPolices* which is a collection of *AdaptationPolicy* elements as previously described and at least one is required for the automated approach to meta-management.

Finally, the *Subsystem* defines one property, *InstanceCount*, that defines how many physical subsystems the specification represents and is optional with the default value of 1. This allows the administrator to define a single specification that can be reused as needed for physical systems with a high degree of similarity avoiding unnecessary duplication in the specification.

The following is the specification of the *subsystem* element:

```
{ //Root Node
  "<<SubsystemName>>":
  {
    "InstanceCount":<<Integer>>,
    "CurrentState": {
      <<SEAMCurrentState>>
    },
    "CurrentConfig": {
      <<SEAMCurrentConfig>>
    },
    "StateSpace": {
      <<SEAMStateSpace>>
    },
    "AdaptationPolicies": [
      <<SEAMAdaptationPolicy>>,
      ...
      <<SEAMAdaptationPolicy>>
    ]
  }
}
```

Listing 5.20: Subsystem Specification

A SEAM *subsystem* is a top level element that must appear directly under the root node of the JSON document. A $SubsystemName$ can be anything except for $MetaManager$ and $Environment$ which are reserved names. Each complete specification must have at least one subsystem element.

Current State & Current Configuration

The *current state* and *current configuration* elements of SEAM are used to elaborate the current values of the key properties and the configuration for a specific autonomic subsystem. These elements are defined as a JSON object composed of a set of name and value pairs representing the property name and the current value for that property. The specification for the *current state* element is:

```
{ //Root Node
  "<<SubsystemName>>": {
    "CurrentState": {
      "<<PropertyName>>":"<<PropertyValue>>", //String Value
      "<<PropertyName>>":<<PropertyValue>>, //Numeric Value
      ...
      "<<PropertyName>>":"<<PropertyValue>>"
    }
  }
}
```

Listing 5.21: Current State Specification

Similarly, the definition of the *current configuration* element is:

```
{ //Root Node
   "<<SubsystemName>>": {
      "CurrentConfig": {
         "<<PropertyName>>":"<<PropertyValue>>", //String Value
         "<<PropertyName>>":<<PropertyValue>>, //Numeric Value
         ...
         "<<PropertyName>>":"<<PropertyValue>>"
      }
   }
}
```

Listing 5.22: Current Configuration Specification

Referring back to the exemplar system presented in chapter 2, the following is an implementation of the *current state* element for the shopping cart subsystem:

```
{ //Root Node
   "ShoppingCart": {
      "CurrentState": {
         "AvgPageResponse":2.6, //Numeric Value
         "ServerCount":12, //Numeric Value
         "ContentFidelity":"High" //String Value
      }
   }
}
```

Listing 5.23: Current State Example

Additionally, the following is an example implementation of the *current configuration* for the shopping cart system from the exemplar presented in chapter 2:

```
{ //Root Node
   "ShoppingCart": {
      "CurrentConfig": {
         "MaximumCost":2000, //Numeric Value
         "CapacityBuffer":10, //Numeric Value
         "CostPerServer":100, //Numeric Value
      }
   }
}
```

Listing 5.24: Current Configuration Example

The values of both the *current state* and *current configuration* elements will be updated at runtime by the meta-monitoring phase of the Meta-MAPE-K as defined in section 4.1. These runtime values represent the current state of both the managed systems and the configuration of the autonomic manager and, consequently, are likely to be referenced by JSON paths in other parts of the SEAM specification. Therefore, the specification of both elements is necessary to define where in the SEAM JSON structure the current state of both elements will be.

Property

A SEAM *property* is used to define the scope of the critical properties of the state space for the *CurrentState* and *CurrentConfig* elements of the subsystems. SEAM defines two types of *property*: a numeric property and a string property.

A numeric property is the most common type of definition that would be used. However, as the implementation of the autonomic subsystem could be a 'black box' the string property definition gives the administrator the ability to validate categorical properties that describe the autonomic subsystem through the use of a regular expression [123]. This gives a flexible way of handling a variety of potential cases in defining the scope for string based attributes and configuration properties in a well-defined and widely supported manner. The specification for a numeric *property* is:

```
"<<PropertyName>>": {
        "Type":"Numeric",
        "Min":<<MinValue>>,
        "Max":<<MaxValue>>,
        "Step":<<StepValue>>,
        "Intervals":<<IntervalsValue>>,
        "TimeCount":<<True|False>>
    }
```

Listing 5.25: Numeric Property Specification

The Min field specifies the lower bound of the potential values for the *property* and the Max field specifies the upper bound of the potential values for the *property*. The $step$ field specifies how big each step is between the lower and upper bounds for the value of the *property*. The $Intervals$ attribute defines that each individual value of the property can be further subdivided into the defined number of intervals. The $TimeCount$ property is a boolean value that specifies that this value should act as a continuous 'clock' in the specification and should loop back to its Min value after the Max value has been met. The $TimeCount$ and $Intervals$ property can be used together to define a granular clock for the model in which larger units of time (e.g., hours) are subdivided into smaller blocks (e.g., 5 min. intervals).

An example of a numeric *property* related to the exemplar shopping cart system presented in chapter 2 is:

```
"ServerCount": {
    "Type":"Numeric",
    "Min":1,
    "Max":20,
    "Step":1
    }
```

Listing 5.26: Numeric Property Example

The second type of *property* is the string *property* which is validated using a regular expression which allows for a wide variety of definition and validation options. The specification for the string *property* is:

```
"<<PropertyName>>": {
      "Type":"String",
      "Regex": "<<RegEx>>"
   }
```

Listing 5.27: String Property Specification

The following is an example of the definition of a string *property* relating to the shopping cart system in the exemplar scenario presented in chapter 2.

```
"ContentFidelity": {
      "Type":"String",
      "Regex": "^(HIGH|LOW)$"
      }
```

Listing 5.28: Numeric Property Example

As shown in the example, the string *property* can appropriately assist in validating a list of valid values by specifying that list as part of a regular expression.

State Space

The *StateSpace* element has two elements defined. Both are collections of *property* elements that define the complete state space of the attributes defined for both the *CurrentState* and *CurrentConfig* elements. The $Properties$ element contains the *property* elements that describe the properties in the *CurrentState* element and $Configuration$ contains the *property* elements that describe the properties in the *CurrentConfig* element. The following is the specification of the *StateSpace* element:

```
"StateSpace":
   {
      "Properties":
         [
            <<SEAMProperty>>
            ...
            <<SEAMProperty>>
         ],
      "Configuration":
         [
            <<SEAMProperty>>
            ...
            <<SEAMProperty>>
         ]

   }
```

Listing 5.29: State Space Specification

The following is an example elaboration of the *StateSpace* element for the shopping cart system presented in the exemplar in chapter 2:

```
"StateSpace":
  {
    "Properties":
      [
        "ServerCount": {
          "Type":"Numeric",
          "Min":1,
          "Max":20,
          "Step":1
        },
        "AvgPageResponse": {
          "Type":"Numeric",
          "Min":0.0,
          "Max":5.0,
          "Step":0.1
        },
        "ContentFidelity": {
          "Type":"String",
          "Regex": "^(HIGH|LOW)$"
        }
      ],
    "Configuration":
      [
        "MaximumCost": {
          "Type":"Numeric",
          "Min":100,
          "Max":2500,
          "Step":100
        },
        "CapacityBuffer":{
          "Type":"Numeric",
          "Min":0,
          "Max":100,
          "Step":10
        }
      ]
  }
```

Listing 5.30: State Space Example

The example *StateSpace* elaboration defines the scope of the state space for the $ServerCount$, $AvgPageResponse$, and $ContentFidelity$ properties of the $CurrentState$ element for the shopping cart system. Additionally, the state spaces for the $MaximumCost$ and $CapacityBuffer$ properties of the $CurrentConfig$ element are also defined.

Environment

The SEAM *environment* element is a specialized subsystem with the reserved name of $environment$ that is a top level element which must appear directly under the root node of the JSON document. This element provides the administrator the option to define the behavior of an additional subsystem to represent the environment which can influence the current and future states of the individual subsystems and is important for the effectiveness of some meta-analysis and meta-synthesis techniques.

```
1  { //Root Node
2    "Environment":
3    {
4      "CurrentState": {
5        <<SEAMCurrentState>>
6      },
7      "CurrentConfig": {
8        <<SEAMCurrentConfig>>
9      },
10     "StateSpace": {
11       <<SEAMStateSpace>>
12     },
13     "AdaptationPolicies": [
14       <<SEAMAdaptationPolicy>>,
15       ...
16       <<SEAMAdaptationPolicy>>
17     ]
18   }
19 }
```

Listing 5.31: Environment Specification

5.5 MetaManager

A SEAM *MetaManager* element describes the global state of the collection of autonomic systems and includes the defintions of the autonomic behaviors of the meta-manager, global knowledge, and global utility. Additionally, it also provides the optional *CurrentState* and *CurrentConfig* elements and a corresponding *StateSpace* element. While these elements are optional, this provides the administrator the option of defining a global state space and configuration for the collection of autonomic systems under management that could include information outside of what is available from each of the subsystems (e.g., the result of an outside security scan) which can be exploited by the meta-manager to improve the effectiveness of the meta-analysis and meta-synthesis techniques.

The following is the specification of the *MetaManager* element:

```
1  { //Root Node
2    "MetaManager":
```

```
3   {
4     "GlobalKnowledge": {
5       <<SEAMGlobalKnowledge>>
6     },
7     "CurrentState": {
8       <<SEAMCurrentConfig>>
9     },
10    "CurrentConfig": {
11      <<SEAMCurrentConfig>>
12    },
13    "StateSpace": {
14      <<SEAMStateSpace>>
15    },
16    "GlobalUtility": {
17      <<SEAMGlobalUtility>>
18    },
19    "AdaptationPolicies": [
20      <<SEAMAdaptationPolicy>>,
21      ...
22      <<SEAMAdaptationPolicy>>
23    ]
24  }
25 }
```

Listing 5.32: MetaManager Specification

A SEAM *MetaManager* is a top level element that must be named $MetaManager$ and each specification must have exactly one.

5.6 Runtime Implementation

To facilitate the practical implementation of SEAM as part of a meta-manager, a reusable parser and engine were created with the following elements:

- **JSON Parser**
  The component that parses the SEAM specification, ensures that it is semantically correct, and adheres to the defined specification for each SEAM element and produces the raw JSON structure of the SEAM specification.
- **JSON Path Resolver**
  The component that recursively examines the raw JSON structure for JSON Paths, determines the current values referenced by those paths, and places the references values into the place of the JSON path and outputs the resolved JSON structure.
- **Native Language Converter**
  The component that convert various elements, principally predicate statements, into statements that can be evaluated by the native programming language in use (e.g., C# or Java) and outputs the native language object structure.
- **Synthesis Tool Compiler**
  The component that uses the resolved object structure to compile the script specific to a

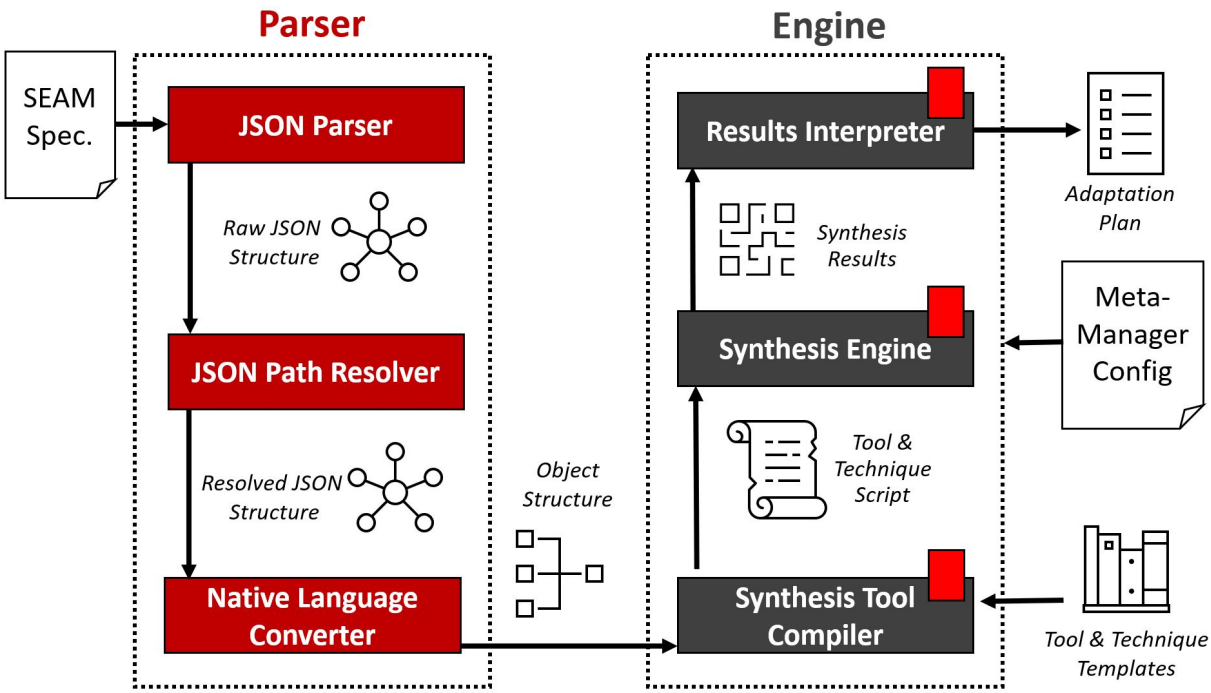


Figure 5.2: SEAM Runtime Component Architecture

particular tool and synthesis technique. This process is facilitated by the ability to define a template that can be leveraged to handle common tool or synthesis technique specific configuration. Additionally, the compiler is created to handle configurable plug-ins for each tool and synthesis technique to allow the compiler to be easily extended to additional tool and technique options.

- **Synthesis Engine**
  The component that uses the tool and technique script and interfaces with the specific tool to run the script and produce the synthesis results. This component supports a plug-in for the individual tool and synthesis technique to allow it to be easily extensible for additional tool and technique options.
- **Results Interpreter**
  The component that interprets the synthesis results and determines what the recommended changes to the configurations of the autonomic subsystems are and converts those into an adaptation plan.

The SEAM parser and engine was incorporated into the Rainbow [24] adaptation framework. As discussed in section 4.1, the analysis and planning phases of the standard MAPE-K loop have been combined in defining the Meta-MAPE-K loop to accommodate both *proactive* and *reactive* adaptation. Therefore, referring to the component architecture presented in figure 5.3, the SEAM parser and engine take the functional place of the model analyzer and the adaptation manager in the Rainbow adaptation framework. The components implement the same interfaces and communicate with the Meta-Models Manager to leverage the SEAM specification and the current state models of the managed system, environment, and autonomic configuration of each

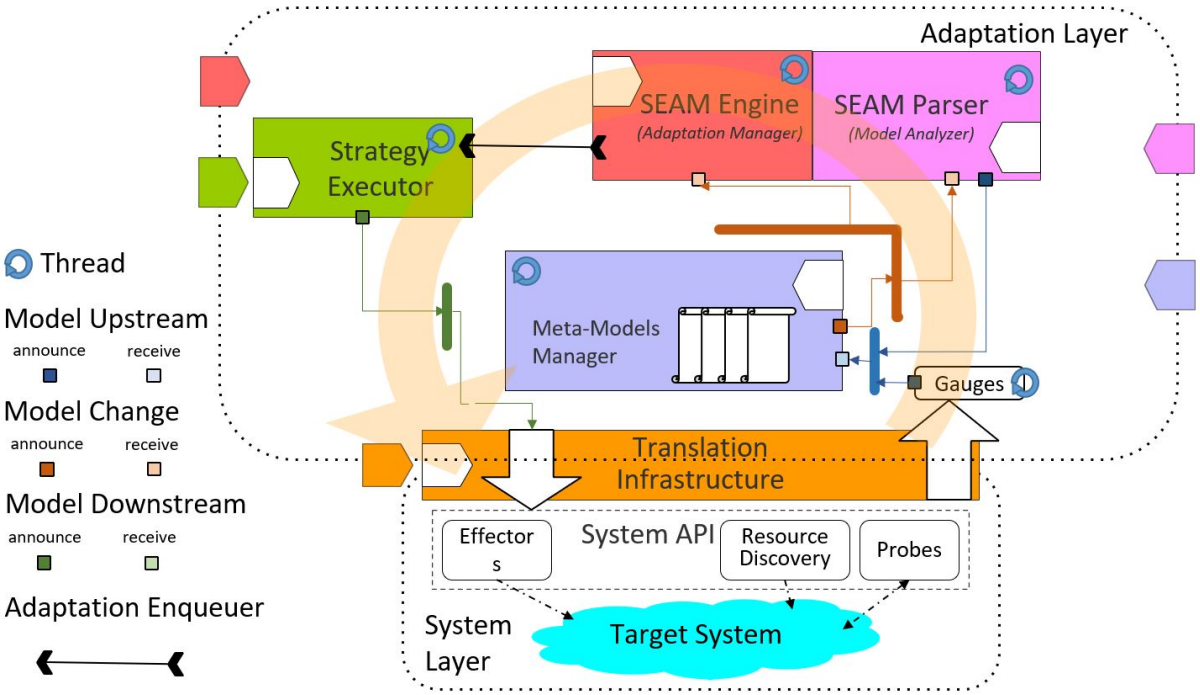


Figure 5.3: SEAM & Rainbow Component Architecture

subsystem. Further, additional probes were created to interface with the autonomic subsystems for each of the case studies and gather the required information necessary to update the elements in the meta-model manager.

This chapter presents SEAM, a domain specific language created to address the requirements for the automated approach to meta-management presented in chapter 4 including the specification of the state spaces and models for the managed systems, environment, autonomic configuration, and the global system state, the adaptation policy that is dependent upon the autonomic configuration, the global knowledge, and the global utility function. The specifications of these elements are implemented and integrated into the Rainbow adaptation framework [24] where they provide the information necessary for a meta-synthesis technique to generate an meta-strategy. The selection of a meta-synthesis technique appropriate to the context is a non-trivial design decision and is the subject of the subsequent chapter.

Chapter 6

Taxonomy of Synthesis Techniques

A key feature of the automated approach to meta-management presented in chapter 4 is that it does not mandate a specific meta-synthesis technique and/or toolset. This allows the implementing administrator to select which technique and toolset is appropriate for the unique demands of the context in which the meta-manager will be operating. However, the automated approach to meta-management does characterize the actions of the individual autonomic managers using a discrete notion of time in which the local environment and the autonomic manager take turns making actions. This is in contrast to a continuous notion of time in which the environment and autonomic manager could take actions simultaneously. Additionally, while there are other ways of meeting the requirements of the automated approach to meta-management (e.g., a simple heuristic approach), this approach builds upon the state-of-the-practice in autonomic systems, see 3.1, and would be expected to dynamically generate a meta-strategy using model checking techniques.

Consequently, this taxonomy will include analysis techniques that deal with a discrete notion of time such as Discrete Time Markov Chains (DTMC), Markov Decision Processes (MDP), Partially Observable Markov Decision Processes (POMDP), and Turn-based and Concurrent Stochastic Multi-Player Games (SMGs). It will not include analysis and synthesis techniques such as Probabilistic Time Autonoma (PTA), Partially Observable Probabilistic Timed Autonoma (POPTA), and Turned-based Probabilistic Timed Games because each has a continuous notion of time. However, this taxonomy will include information on Continuous Time Markov Chains (CTMC) as the extension of SEAM to include the necessary items is discussed in section 11.2.

Therefore, this chapter presents the taxonomy of meta-analysis and synthesis techniques to provide guidance on their suitability for various contexts of collections of autonomic systems. Each section will include the following information about each of the examined techniques:

- **Description** - A brief description of the model checking technique and will include references to comprehensive source material.
- **Timeliness** - A discussion regarding the key properties that relate to the speed in which the technique can produce a result.
- **Assurance** - A discussion regarding the level of confidence and administrator can have in the result the technique produces.
- **Computational Scalability** - A discussion of how well the technique can scale to models

of larger sizes.

- **Guidance on Applicability** - A discussion of the suitability of the technique to operate in various contexts of collections of autonomic systems.

However, there are some related items that are not examined as part of this taxonomy as they relate to lower level issues and concerns that are beyond the scope of this taxonomy. For example, the properties of the individual computer hardware the technique is run on will impact its timeliness and assurance, but that relationship is not of primary concern in determining the suitability of a technique to a particular context in which a collection of autonomic systems is operating. Similarly, it is also beyond the scope of this taxonomy to provide guidance on the efficiency of the implementation of any particular analysis algorithm. Additionally, the relationship between the fidelity of the model and the level of assurance provided by the technique is a fundamental trade-off in the use of any control system and therefore will not be explored. Therefore, the discussions of the assurance provided by any technique are assumed to be subject to the limitations relating to the fidelity of the model provided.

6.1 Discrete and Continuous Time Markov Chains

Description A discrete time Markov chain (DTMC) [72] is a characterization of a process with a set of states and a transition matrix that defines the probability of moving from one state to another state. Using the exemplar scenario from chapter 2, one state of the shopping cart subsystem would be having 10 servers and there would be a probability of it moving to a state with 11 servers and another probability for moving to 9 servers. A path is a sequence of states that the system transitions between and each transition in the path represents a single discrete-time step.

A continuous time Markov chain (CTMC) [72] is similar to a DTMC, except its transition matrix defines the rate at which one state transitions to another which can be used to determine the probability of transitioning from one state to another in a specific number of time steps. Consequently, the transitions between the states do not represent a discrete time-step.

The probability transitions of a DTMC and the transition rates of a CTMC represent the *probabilistic choice* of each of the models which causes different paths to be generated depending on the starting state and the transitions that are made. Therefore, to analyze a DTMC or CTMC, it is common to generate a set of paths, potentially with varying conditions like changing the starting state of the model, that are representative of the set of behaviors of the system.

Using different logic specifications like probabilistic computational tree logic (PTCL)[56], the set of paths can be analyzed to answer questions like; what is the probability of the shopping cart system moving from 10 servers to 11 servers within 3 time steps? More relevant to meta-management is the question; what is the maximum reward the system can accumulate over 5 time steps? For the purposes of this approach to meta-management, the reward is determined by scoring the value of each state in the path using a utility function [9, 10].

Timeliness The use of DTMCs and CTMCs for meta-analysis and meta-strategy synthesis are advantageous because of the ability to tune different parameters of the analysis to suit the

context. Specifically, the number of steps each path has in it, representing the time horizon for the analysis, and the number of paths to be generated and subject to analysis. The lower each of these values are the less time will be required to complete the analysis and vice-versa. However, the lower the number for either of these parameters, the less assurance the technique will be able to provide.

Assurance The analysis of DTMCs and CTMCs uses, primarily, a technique of generating a set of paths that are representative of the potential behavior of the collection of autonomic subsystems. Therefore, this technique does not guarantee that every possible path is examined which results in uncertainty in the answer to questions like: what is the highest reward the system can accumulate? The analysis can only answer this question based upon the generated representative paths of the behavior of the system, not all possible paths. Therefore, it is possible that a higher reward is actually possible, but that path was not included in the set of generated paths. This means that the assurance of the system is based upon the *most likely* behavior of the collection of autonomic systems. However, the longer the paths (i.e., time horizon) and the higher the number of paths generated the higher the degree of assurance can be established, but will take longer to produce a result.

Computational Scalability Another advantage of the analysis of DTMCs and CTMCs is that because of the use of generating samples of the behavior of the collection of autonomic systems they can be used to analyze models of significant size.

Guidance on Applicability DTMCs and CTMCs have been used in a variety of contexts in self-adaptive systems, see [15, 16, 83], and for the purposes of the meta-management of a collection of autonomic systems was used in the case studies presented in chapters 7 and 9 of this thesis. It is most appropriate in situations where the level of assurance may not be the critical concern of the administrator implementing a meta-manager. For example, the consequences of lower assurance in collections of autonomic systems serving as a web site, back end IT infrastructure, home automation system, asset monitoring systems, and others is likely to be minimal. However, the consequences of lower assurance in other collections of autonomic systems serving as an autonomous vehicle, naval warship, or similar might be severe requiring a higher degree of assurance than the analysis of DTMCs and CTMCs can provide.

6.2 Markov Decision Processes

Description A Markov Decision Process (MDP) [35] is a characterization of a process with a set of states, a set of actions, and a transition matrix which defines the probability of moving from one state to another state depending on the action selected. Using the exemplar scenario from chapter 2, if the shopping cart system is in a state with 10 servers and selects the action to add servers, there is a different set of probabilities for what the resulting state would be than if the action to decrease fidelity was selected. The selection of the resulting state is non-deterministic and represents the *probabilistic choice* of the MDP. The MDP also defines a reward or cost

function that determines the score of the transition based upon the state of the system and the action taken.

The analysis of a MDP is based upon *formal verification* or the systematic approach that applies mathematical reasoning to obtain guarantees about the correctness of the system and *model checking* is one approach [35]. *Model checking* is based on the construction and analysis of the system model in which states represent the possible configurations of the system and transitions between states capture the way the system can evolve over time [35]. This approach can answer questions to determine if a model meets a desired property like; will the system be in a state with 11 servers deployed? This type of question is of limited use in meta-management analysis and synthesis, but a generalized form of model checking, referred to a *probabilistic model checking* is of more use. By analyzing the complete set of states and transitions the analysis of an MDP can answer questions like: what is the probability of the average page response time going under 2.5 seconds in 3 time steps? These questions are expressed in a logic referred to as the probabilistic computational tree logic (PTCL) [56].

Timeliness The timeliness of the analysis of an MDP is dependent upon two factors: (1) the efficiency of the algorithms in the toolset or framework being used and (2) the complexity of the model being analyzed. As previously mentioned, for the purposes of this taxonomy and the automated approach to meta-management, it is assumed that there are no practical improvements to be made to the implementation of the algorithm that would impact its timeliness. Therefore, the complexity of the model has a critical impact on the timeliness of the model. The complexity of the model is characterized by the number of states, the number of transitions, and the degree of variance in the statistical options. The higher any of these measures are the larger the model of the system that will need to be generated and analyzed and the longer it will take to complete, can be in the range of minutes to hours.

Assurance The *probabilistic model checking* of an MDP examines the complete model of the system, not a representative set of paths. Therefore, it provides a higher degree of assurance about the result than sampling techniques.

Computational Scalability It is possible to create a model that is too large to be held within the defined memory space of the toolset or framework. While this is often a limitation of the toolset or framework in use, there is a point at which the model will become computationally intractable to analyze regardless of any other conditions.

Guidance on Applicability The use of an MDP directly in self-adaptive systems is common, but in a more specialized form, please see Stochastic Multi-Player Games later in this chapter, but for the purpose of the meta-management of a collection of autonomic systems it would be most appropriate in situations where the level of assurance is of primary concern and the timeliness of the analysis is secondary.

6.3 Partially Observable Markov Decision Processes

Description A Partially Observable Markov Decision Process (POMDP) [88] is an extension of a Markov Decision Process (MDP) with the difference being that MDPs operate over states of the system, POMDPs operate over observations of the system. Using the exemplar scenario presented in chapter 4, if the shopping cart system is currently using 10 servers, in an MDP, that is considered the state of the system. However, in a POMDP, the measurement that the shopping cart is using 10 servers is considered an observation, allowing for the possibility that the observation of the system and the actual state of the system might be different. This means that the actual state of the system cannot be directly determined. For the purposes of this taxonomy and as relevant to meta-management, the remaining properties of a POMDP function similarly to a MDP including the reward structures and the specification of the expressions using probabilistic computational tree logic (PTCL) [56].

However, in the analysis of an MDP you can use efficient computational techniques to iterate to build a sequence of strategies until an optimal one is found and value iteration to calculate increasing precise approximations of a value [88]. Since a POMDP is undecidable [76], these techniques are not available. Instead, the analysis method approximates the optimal value by calculating the upper and lower bounds of the value to give an approximation.

Timeliness The timeliness of a POMDP is similar to the timeliness of a standard MDP with the exception being that a POMDP will have increased sensitivity to the size of the model. A POMDP introduces uncertainty on the current state of the system in addition to the non-determinism and probabilistic behavior in standard MDPs. This can cause the number of elements that are part of the analysis to increase causing the technique to take longer to complete than a standard MDP.

Assurance The introduction of uncertainty in the current state of the system can account for other situations relevant to the meta-management of collections of autonomic systems and, consequently improve the assurance of the result. However, the final result can only be approximated within an upper and lower bound.

Computational Scalability Similarly to a MDP, there is a point at which the model will become computationally intractable to analyze regardless of any other conditions.

Guidance on Applicability There are examples of POMDPs in use within self-adaptive systems, see [80, 124]. However, for the purposes of meta-management the primary consideration is the advantage that accounting for uncertainty in the state of the system provides over the decreased levels of timeliness and an approximated result. As such the technique might be best appropriate in IoT systems like the meta-management of a collection of oil wells or robotic systems in which the collection of the current state information can be unreliable and the longer time to a results and approximated results might be acceptable trade-offs.

6.4 Concurrent & Turn-based Stochastic Multi-Player Game

Description A Turn-Based Stochastic Multi-Player game is a specific type of Markov Decision Process in which each player has a specific set of actions available to them and sequentially rotates which player is making the choice of which action to take in that discrete time step, see [21, 115]. In the event that a player in the game is established to make choices that are opposed to the goals of the other players, this is referred to as an adversarial game. Additionally, SMGs also allow for the definitions of reward structures based upon the scoring of the state of a player. Referring to the exemplar scenario presented in chapter 2, the shopping cart system is one player with a set of actions available, the adaptive tactics, and is trying to reduce the *average page response time*, the established reward structure. However, another player can be established to represent the environment in which the shopping cart is operating that makes choices to increase the *average page response time* of the shopping cart system. The interactions between these two players would constitute an adversarial stochastic multi-player game. A Concurrent SMG game shares all of the same characteristics as a Turn-Based SMG except that instead of the players making their choices sequentially, they can make them simultaneously, see [74] for more information.

Similarly to an MDP, the analysis of a SMG, turn based or concurrent, is a *formal verification* accomplished by *model checking* in which a model is built and analyzed that represents the possible states of the players and the actions they take capture the way the game evolves over time. Using the probabilistic alternating-time temporal logic with rewards [21] the *model checking* can determine what is the maximum or minimum reward that can be achieved if each player makes the best possible choice to achieve their objective at each opportunity. An adaptation strategy can be determined by examining the set of choice the player made to maximize that reward.

Timeliness The timeliness of a SMG is subject to the same limitations of a standard MDP which is highlighted by its sensitivity to the size of the model.

Assurance As the analysis of a SMG is based upon *probabilistic model checking*, the analysis considers the complete model of the system, not a representative set of paths. Therefore, it provides a higher degree of assurance about the result than sampling techniques.

Computational Scalability The computational scalability of a SMG is subject to the same limitations as an MDP in that it is possible to create a model that is too large to be held within the defined memory space of the toolset or framework and at some point would become computationally intractable. However, there might be some context dependent optimizations and assumptions that can be used to assist with this limitation. The Google Control Plane case study, presented in chapter 8, makes an assumption that because all of the autonomic subsystems are practically identical it was not necessary to model each one as an individual player. Instead only one system was modelled and any changes were applied to all uniformly.

Guidance on Applicability There are numerous examples of SMGs being used in the context of self-adaptive systems, see [15, 16, 17, 18]. For the purposes of meta-management, the Google

Control Plane Case Study presented in chapter 8 uses a SMG to perform meta-analysis and meta-strategy synthesis. However, as can be seen in that case study, the potential size of the model presents a practical challenge to the use of this method. Additionally, in the context of self-adaptive systems, most SMGs are constructed to be adversarial with a system playing against the environment. Consequently, their results are limited to either the best or worst case scenario as appropriate for the context. This is in contrast to a discrete time Markov chain (DTMC) which provides the 'most likely' scenario. Therefore, SMGs are most appropriate in contexts in which a conservative approach to meta-management is preferred and the timeliness requirements are not as demanding. It is also appropriate in contexts in which external factors, like the actions of the environment, play a significant role in the meta-management. This might include contexts like a building climate control system which has to manage multiple air conditioning, heating, humidity, and other systems while accounting for the actions of the weather and the timeliness requirements will tolerate the analysis of the model taking minutes to hours to complete.

This chapter presents a taxonomy of meta-synthesis techniques and provides guidance on their applicability to the various contexts in which collections of autonomic systems operate. This is a critical design time decision for the implementing administrator as the selection of an inappropriate technique for the context can compromise the scalability and effectiveness of the approach. The use of several of these techniques is a subject of the case studies presented in subsequent chapters to evaluate the automated approach to managing collections of autonomic systems.

Chapter 7

Case Study: Amazon Web Services Shopping Cart

Similar to the exemplar scenario presented in chapter 2, this case study examines a common use case for Amazon Web Services(AWS), the creation and operations of a web based system, specifically a shopping cart. The complete information on the validity and selection of this case study can be found in section 1.2. However, this case study was selected to evaluate the ability of the meta-manager to improve the homeostatic operations of the collection of autonomic systems on popular architectural pattern, code base, and operations platform that is in wide industrial use.

7.1 Background & Context

One of the many reasons AWS, and other similar public cloud services like Azure and Google Cloud, are popular platforms for web based systems is the ability to cost effectively create and operate a highly available and scalable web-based system with minimal effort dedicated to infrastucture and other lower level concerns. There are many different features of AWS that contribute to this objective such as multiple operating regions and availability zones within each region, redundant hardware and network resources, and security systems. At the application level AWS also offers multiple methods of providing autonomic management capabilities to address, primarily, performance and scalability QoS concerns.

The primary method is through the use of AWS Elastic BeanStalk (AWS ELB) [103]. The primary feature of AWS ELB is the ability to auto scale an application, both up and down, to maintain configured performance levels. The primary mechanism for this is referred to as an auto scaling group. An auto scaling group has several different options configurations available some of which are:

- Instances: The minimum and maximum number of virtual machines that can be deployed to maintain configured performance levels.
- Fleet Composition: The type of instances the auto scaling group should use, on-demand or spot
- On-Demand Base: The minimum number of on-demand instances the auto scaling group

provisions before considering spot instances.

- On-Demand above base: The percentage of on-demand instances above what is required to maintain configured performance levels
- Capacity Rebalancing: Whether or not the capacity should be automatically rebalanced in response to a spot recall notification.

An important factor in establishing auto scaling groups is the use of on-demand versus spot instances. An on-demand instance is a virtual machine that is specifically requested by an account and, once provisioned, is under the exclusive control of the account. In the absence of any other specific agreement, an on-demand instance is billed for every second it is in the 'running' state at a flat published rate. However, a spot instance is excess virtual machine capacity that AWS has available that can be bid on to provide additional capacity to an application. Spot instances can be up to 90% cheaper than an equivalent on-demand instance, but are not guaranteed to be available. However, if AWS projects it will need the additional capacity that is currently being utilized by spot instances, AWS can send a 'recall' notification which gives the application 120 seconds to appropriately adjust to the forced reclamation of those resources.

The choice of which instance type, on-demand instances versus spot instances, is most appropriate for a particular application is non-trivial and subject to change depending on conditions in the organization and the environment. For example, it is common for a web system to have an inter-day seasonality to its traffic demand, a scale up in the morning in the local geography and a scale down in the evening. It might be cost effective to handle this scaling with spot instances. However, if the capacity is 'recalled' then the application might not have the capacity available to meet the demand and fail to meet its quality objectives. Conversely, a solution that uses on-demand instances might become economically challenged in response to sudden and transient spikes in traffic demand.

An application can also leverage multiple cloud native AWS services. Cloud native services are managed services which means that it will service the needs of multiple AWS customers on the same underlying infrastructure. As a consequence, while these services can be an autonomic subsystem in an application that guarantees multiple QoS objectives related to availability and performance, the configuration of the autonomic management capabilities is not available to any individual AWS customer. Therefore, these services are considered a 'black box' autonomic subsystem that cannot be subject to this approach to meta-management. However, some AWS cloud native services offer additional add-on services to give application greater flexibility in the autonomic capabilities of the service.

As a relevant example, Amazon DynamoDB is a fully managed large scale columnar data system that implements the underlying data storage and retrieval system described in [27]. However, an additional service complimentary to DynamoDB is available with configurable autonomic capabilities. Amazon DAX [100] is a caching service that sits in between the application and the primary DynamoDB instance to store selected data in memory to support fast retrieval use cases. The DAX cluster is customer specific and some the autonomic capabilities of the cluster are configurable by the customer. However, while the basic autonomic options are similar (e.g., add compute instances) there is a different set of architectural concerns in using these mechanisms.

The two configurable properties with potentially significant impact on the ability of the data services tier to meet the QoS objectives and the overall solution cost are the instance type and the

number of instances deployed. For the representative purposes of this exemplar, there are two relevant instance types, fixed and burstable. A fixed instance type allocates a specific percentage of a virtual CPU (vCPU). The instance can use up to that allocated capacity, but that capacity is always available to the application it is allocated for. A burstable instance type establishes a baseline at a baseline amount. If the application uses less than the baseline then vCPU credits accrue up to a maximum. However, when the application uses above the baseline the credits are drawn down. If the burstable instance is configured for standard mode then when your application is above baseline and no accrued credits are available the system gradually reduces the in-use compute capacity back to the baseline. If the burstable instance is configured for unlimited mode the application will be provided all necessary compute capacity which will be billed at the end of the 24 hour period at a rate higher than the equivalent fixed capacity.

These options differ from the on-demand verses spot instances available in Elastic BeanStalk because the Amazon DAX application must maintain its state, specifically the cached data, to be effective. The Elastic Beanstalk autonomic manager has the underlying assumption that the individual instances can be added and removed to impact capacity without compromising the functionality of the application itself. However, that assumption is not appropriate for a data-centic application like Amazon DAX. Therefore, the scaling mechanism is more granular by dynamically allocating the compute units available to the instances themselves. The stateful nature of Amazon DAX also influences the choice of the number of instances to deploy.

In a each DAX cluster there is a single primary node which fulfills requests for data, handles write operations to DynamoDB, and coordinates the actions of the other 'read replica' nodes which are responsible for answering queries and evicting data from the cache. Adding additional nodes will increase throughput and, consequently, improve the ability of the system to meet its QoS objectives, but will take time to come online and receive a copy of the cache and will increase costs. Handling the effects of an adaptation tactic taking time to be effective, referred to as latency aware adaptation, is beyond the scope of this thesis, see [82], therefore only the effects of the increased cost and the performance benefit are considered.

Additionally, the database system will have a different seasonality to its traffic load than the web servers. For example, it is common for enterprises to run large data processing jobs during night time hours to ensure their load does not interfere with the seasonality of the web traffic. This leads to the database having a load profile that aligns with the load profile of the web servers during prime hours, but significantly diverges under the expanded and consistent load for the data processing jobs. This clear separation in the types of work mean that it is easily predicatable to choose between the different instance types for the database tier, burstable for the prime web hours and fixed for the overnight data processing jobs. Therefore, that choice is not considered as part of this case study. However, that load pattern does provide a second critical choice, the allocation of available resources. Since the demand for the services from the web tier of the application are significantly reduced but the demand on the data tier is quite high, it might be advantageous to the global system to reallocation resources (e.g., monetary budget) from the web tier to the database tier during the off-peak hours and then reset the resource allocation for the peak hours. This reallocation of resources, leading to additional database instances, allows the database tier to more effectively meet its QoS objectives.

7.2 Experiment

To evaluate the applicability and effectiveness of using a meta-manager in the context of improving the continuous operations of a web based system on Amazon Web Services (AWS). An exemplar web based system was established using an open source shopping cart system running on multiple native AWS services commonly available on the AWS public cloud. Additionally, a test bench was established that simulates a representative daily traffic load to the front end web based user interface. A meta-manager was then established to monitor the conditions of the individual subsystems to discover potential optimizations in the configurations of their autonomic managers with the goal of reducing cost to maintain the quality-of-service objectives. This section is organized as follows: (1) describes the architecture of the exemplar shopping cart system, (2) describes the SEAM specification of the meta-manager, (3) the resulting PRISM model that is used to perform the meta-analysis and meta-strategy synthesis and (4) describes the load simulation method used to exercise the exemplar system.

Architecture

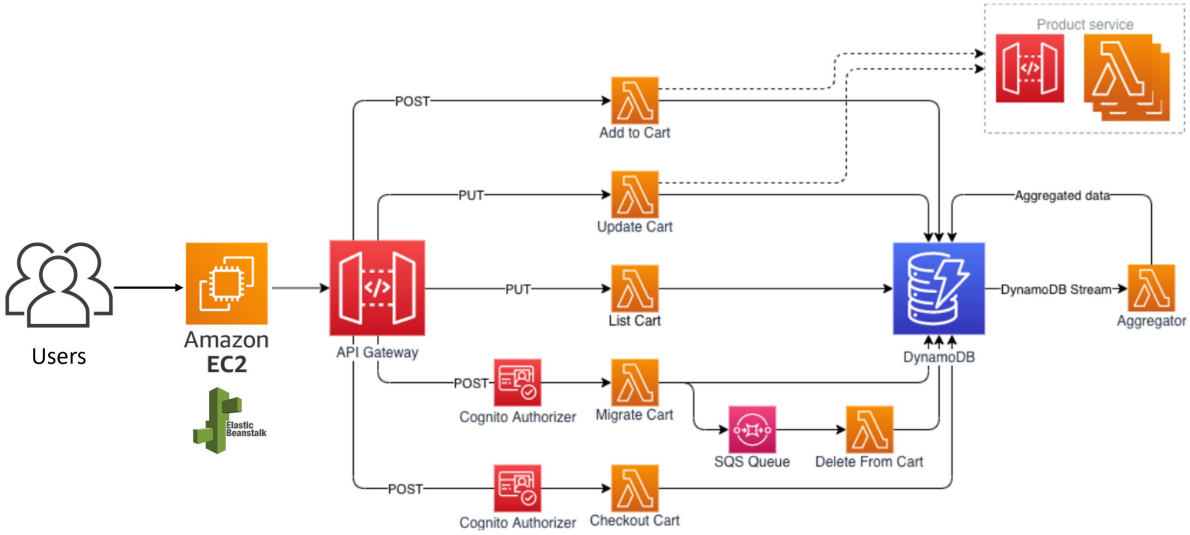


Figure 7.1: Shopping Cart Exemplar System - Architecture

The exemplar shopping cart system uses the open source AWS Serverless Shopping Cart System [108] provided by AWS as part of its AWS Samples package and hosted on Github [107] and augmented with a front end user interface hosted on Amazon EC2 instances managed by Elastic BeanStalk. Each of these elements and services will be described in detail and organized as (1) Front End UI, (2) Middle Tier Services, and (3) Data Services.

Front End User Interface

The front end user interface consists of three components: (1) the web application, (2) the Amazon EC2 instances, and (3) the Amazon Elastic BeanStalk management service. The front end user interface consists of an ASP .NET application, v. 4.7.2 which randomly accesses the services provided by the middle tier services and is the target of the load testing service to generate an appropriate user load for experimental purposes.

Amazon EC2 The Amazon Elastic Compute Cloud (EC2) [102] provides the ability to deploy and manage virtual machines as needed to support the needs of an application. For the purposes of this exemplar the front end user interface is hosted on Microsoft Windows 2022 Server Base using the t2.micro instance type. Details of the instance types can be found at [104].

Amazon Elastic BeanStalk The Amazon Elastic BeanStalk service (EBS) [103] manages the auto scaling capability of Amazon EC2 virtual machine instances and is acting as the autonomic manager for the Front End UI subsystem in the collection of autonomic systems that compose the shopping cart system. To manage the auto scaling the service was configured with a single auto scaling group with the following relevant properties with the described initial configuration:

- Instances: Minimum:1 and Maximum:10
- Fleet Composition: On-demand
- On-Demand Base: 1
- On-Demand above base: 0%
- Capacity Rebalancing: Enabled

The EBS service leverages the capabilities of another AWS service, CloudWatch [98], to examine the state of the cluster of virtual machine instances and determine if scaling is necessary. In this case, EBS and CloudWatch, were configured to scale up if the number of requests per instance exceeded 500 requests per instance.

Middle Tier Services

The middle tier services provide the functionality necessary to perform the activities required of a shopping cart. For example, a product catalog with the standard CRUD (Create, Retrieve, Update, and Delete) operations, and similarly, CRUD operations for the shopping cart itself. To enable this functionality the shopping cart system is composed of four components: (1) the API gateway, (2) the individual AWS Lambda functions, (3) the identity and access management (IAM) with Amazon Cognito, and (4) a queuing service with Amazon SQS. However, all of the AWS services used in the middle tier are managed cloud services meaning that it will service the needs of multiple AWS customers on the same underlying infrastructure. As a consequence, while the managed service is an autonomic subsystem that guarantees multiple QoS objectives related to availability and performance the configuration of the autonomic management capabilities is not available to any individual AWS customer. Therefore, for the purposes of this exemplar, the middle tier services are considered a 'black box' autonomic subsystem that cannot be subjected to this approach to meta-management.

Amazon API Gateway The Amazon API Gateway [97] provides the REST API endpoint management necessary to service web API interactions at scale. It will manage the intake of the message, the proper routing of that message, and any response as needed. It serves as the primary point of contact for the front end UI system.

Amazon Lambda Amazon Lambda [105] is a serverless, event driven compute service that allows the user to run code for many different types of application or backend service without having to provision or manage servers. In this exemplar it is used to provide the code that implements the business logic for the following shopping cart functions:

- Add To Cart
- Update Cart
- List Cart
- Migrate Cart
- Delete From Cart
- Checkout Cart

It also provides the code for a similar set of functions for the 'Product Service'.

Amazon Cognito Amazon Cognito [99] is an identity and access management (IAM) system. In this exemplar it is used to ensure the user is authenticated before allowing operations to alter the cart or confirm the purchase. However, for the purposes of this experiment it is configured with a single authenticated user with sufficient permissions to perform all necessary actions.

Amazon SQS Queue Amazon Simple Queue Service (SQS) [106] provides the ability to send, store, and receive messages between different software components. In this exemplar, it is used to queue interactions for batch processing as a best practice for transactions that remove data items from the data services provider in this case AWS DynamoDB.

Data Services

The data services tier of the exemplar shopping cart system is implemented on Amazon DynamoDB [101]. Amazon DynamoDB is a fulled managed large scale columnar data system that implements the underlying data storage and retrieval system described in [27]. While Amazon Dynamo is a fully managed service with autonomic capabilities that cannot be configured by customer accounts, an additional service complimentary to DynamoDB is available with configurable autonomic capabilities. Amazon DAX [100] is a caching service that sits in between the application and the primary DynamoDB instance to store selected data in memory to support fast retrieval use cases. The DAX cluster is customer specific and some the autonomic capabilities of the cluster are configurable by the customer. The two properties most relevant to this exemplar and their initial values are:

- Instance Count: 1
- Node Type Family: T-Type (Burstable), T2.Small (Standard)

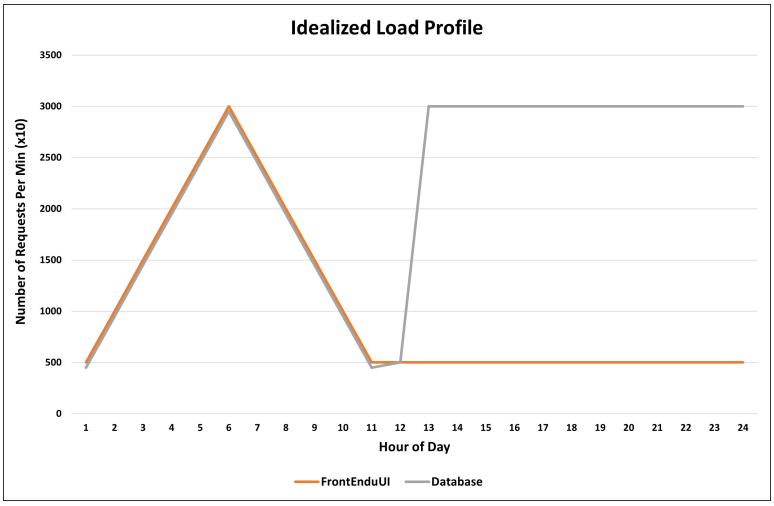


Figure 7.2: Shopping Cart Exemplar System - Idealized Load Profile

For a DAX cluster, the instance type cannot be changed once the cluster is provisioned and running. To change the instance type, the cluster must be replaced.

Load Simulation

The load for the shopping cart system was artificially generated using the load testing capabilities of Visual Studio 2019. The load test generates requests that target the front end user interface of the AWS shopping cart system which, in turn, generates load for the middle tier and database services. The amount of requests generated varies with time by using random load generation mechanism that follows a load profile as depicted in figure 7.2.

This load profile mimics a common load pattern for a shopping cart system by simulating a standard business day running between 6am on one day, hour 1 in figure 7.2, to 6am the next business day, hour 24 in figure 7.2. For the first 6 hours of the day, between 6am and 12pm, the load steadily increases and peaks. From 12pm to 6pm the load steadily drop down to a minimum established level. This pattern is followed for both the front end UI and the database system. However, while the front end UI maintains a varying but consistent load from 6pm to 6am and is continuously serviced by the database system, the load on the database system peaks due to the scheduled nightly data processing jobs that are common in enterprises.

SEAM Specification

The complete SEAM specification for this case study can be found in appendix A, but this section will provides details on several key areas including the definitions of global utility, global knowledge, the environment, the shopping cart system, and the database system.

Global Utility The global utility definition for the shopping cart case study can be found in listing 7.1. It defines four different utility functions, using predicate statements, depending on

the state of the $FrontEndUI$ and $Database$ systems. Specifically, whether the ratio between the $CurrentLoad$ and $CurrentCapacity$ for each system is current above or below the ideal target of 0.66 or 66% of total capacity leading to four different possibilities. Each of the utility functions is a weighted function, commonly used in expected utility [9], that provides equal weight (0.25 or 25%) to each of four factors: the difference between ideal and current value of the ratio of current load to current capacity for the $ShopCart$ and $Database$ systems and the cost for both of those systems as a difference from the $MaxCost$.

```
{
  "MetaManager": {
    "GlobalUtility": [
      {
        "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) > 0.66 &
            ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
            ↪ <= 1 & (#$.FrontEndUI.Database.CurrentLoad# /
            ↪ #$.FrontEndUI.Database.CurrentCapacity#) > 0.66 &
            ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) <= 1",
        "Formula":"0.25 * (1 - (#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) - 0.66) + 0.25 *
            ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
            ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 -
            ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#) -
            ↪ 0.66) + 0.25 * ((#$.Database.CurrentConfig.MaxCost# -
            ↪ #$.Database.CurrentState.CurrentCost#)/ #$.Database.CurrentConfig.MaxCost#)"
      },
      {
        "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) <= 0.66 &
            ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
            ↪ >= 0 & (#$.FrontEndUI.Database.CurrentLoad# /
            ↪ #$.FrontEndUI.Database.CurrentCapacity#) <= 0.66 &
            ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) >= 0",
        "Formula":"0.25 * (1 - (0.66 - #$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#)) + 0.25 *
            ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
            ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 - (0.66 -
            ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#)) +
            ↪ 0.25 * ((#$.Database.CurrentConfig.MaxCost# - #$.Database.CurrentState.CurrentCost#)/
            ↪ #$.Database.CurrentConfig.MaxCost#)"
      },
      {
        "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) <= 0.66 &
            ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
            ↪ >= 0 & (#$.FrontEndUI.Database.CurrentLoad# /
            ↪ #$.FrontEndUI.Database.CurrentCapacity#) > 0.66 &
            ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) <= 1",
        "Formula":"0.25 * (1 - (0.66 - (#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) - 0.66)) + 0.25 *
            ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
```

```
                ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 ∗ (1 −
                ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#) −
                ↪ 0.66) + 0.25 ∗ ((#$.Database.CurrentConfig.MaxCost# −
                ↪ #$.Database.CurrentState.CurrentCost#)/ #$.Database.CurrentConfig.MaxCost#)"
15        },
16        {
17          "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
                ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) > 0.66 &
                ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                ↪ <= 1 & (#$.FrontEndUI.Database.CurrentLoad# /
                ↪ #$.FrontEndUI.Database.CurrentCapacity#) <= 0.66 &
                ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) >= 0",
18          "Formula":"0.25 ∗ (1 − (#$.FrontEndUI.CurrentState.CurrentLoad# /
                ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) − 0.66) + 0.25 ∗
                ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# − #$.FrontEndUI.CurrentState.CurrentCost#)/
                ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 ∗ (1 − (0.66 −
                ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#)) +
                ↪ 0.25 ∗ ((#$.Database.CurrentConfig.MaxCost# − #$.Database.CurrentState.CurrentCost#)/
                ↪ #$.Database.CurrentConfig.MaxCost#)"
19        }
20      ]
21    }
22  }
```

Listing 7.1: Shopping Cart Case Study - SEAM Global Utility Specification

Global Knowledge The global knowledge for this case study, in listing 7.2, includes a constraint that defines the relationship between the $MaxCost$ properties for the $ShopCart$ and $Database$ subsystems. Specifically, the total of the systems $MaxCost$ cannot exceed 500. This does not mean that the individual systems will actually consume up to their $MaxCost$, just that the combination cannot exceed that amount. This allows the meta-manager the flexibility to determine the preferred setting for the $MaxCost$ for each system.

```
1 "GlobalKnowledge": [
2   {
3     "Relation":
4     {
5       "Type":"Constraint",
6       "Predicate":"#$.Database.CurrentConfig.MaxCost# = 500 − #$.FrontEndUI.CurrentConfig.MaxCost#"
7     }
8   }
9 ]
```

Listing 7.2: Shopping Cart Case Study - SEAM Global Knowledge Specification

Environment The complete definition for the environment can be seen in appendix A, but listing 7.3 presents the adaptation policy for the $Environment$. The first adaptation policy applies for the first 6 hours of the day (e.g., 6am to 12pm) in which the environment impacts the

$CurrentLoad$ on $CurrentState$ the $ShopCart$ subsystem which also sets the $CurrentLoad$ on the $Database$ subsystem. The $CurrentLoad$ for the $ShopCart$ system is set using an asymmetric Gaussian Distribution which is 'right skewed'. This models the increase load on the web system during this period of the day. Conversely, the second adaptation policy represents the second six hour period of the day and defines a 'left skew' asymmetric Gaussian distribution for the $CurrentLoad$ of both systems as the load on the web system drops through the end of the peak hours. The third and fourth policies define the $CurrentLoad$ for the third and fourth six hour periods in which there is a fixed load on the database system in addition to the reduced but still active load due to the $CurrentLoad$ on the $FrontEndUI$ which is specified using a normal Gaussian distribution.

```
"Environment": {
  "AdaptationPolicies": [
    {
      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# <= 6",
      "Behaviors": [
        {
          "StatePredicate":"",
          "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
              ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 3, 1, −0.5) &
              ↪ #$.Database.CurrentState.CurrentLoad# = #$.FrontEndUI.CurrentState.CurrentLoad#"
        }
      ]
    },
    {
      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 6 & #$.Global.CurrentState.HourOfDay# <=
              ↪ 12",
      "Behaviors": [
        {
          "StatePredicate":"",
          "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
              ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 1, 3, 0.5) &
              ↪ #$.Database.CurrentState.CurrentLoad# = #$.FrontEndUI.CurrentState.CurrentLoad#"
        }
      ]
    },
    {
      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 12 & #$.Global.CurrentState.HourOfDay# <=
              ↪ 18",
      "Behaviors": [
        {
          "StatePredicate":"",
          "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
              ↪ N(#$.FrontEndUI.CurrentState.CurrentLoad#, 0.2) &
              ↪ #$.Database.CurrentState.CurrentLoad# = 150 + #$.FrontEndUI.CurrentState.CurrentLoad#"
        }
      ]
    },
    {
      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 18 & #$.Global.CurrentState.HourOfDay# <=
```

```
                ↪ 24",
32          "Behaviors": [
33            {
34              "StatePredicate":"",
35              "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
                    ↪ N(#$.FrontEndUI.CurrentState.CurrentLoad#, 0.2) &
                    ↪ #$.Database.CurrentState.CurrentLoad# = 150 + #$.FrontEndUI.CurrentState.CurrentLoad#"
36            }
37          ]
38        }
39    ]
40  }
```

Listing 7.3: Shopping Cart Case Study - SEAM Environment Specification

Front End UI The complete specification of the $FrontEndUI$ can be found in appendix A, however, the definition of the two adaptation policies are presented in listing 7.4. The first adaptation policy defines the autonomic behaviors of the subsystem when the 'on-demand' instance type is being used, ($\$.FrontEndUI.Configuration.InstanceType = 0$). The first entry specifies that the adaptation manager will add capacity, $CurrentCapacity$, and cost, $CurrentCost$, when the ratio between $CurrentLoad$ and $CurrentCapacity$ is above 0.75 or 75%. The second specifies the behavior of the subsystem when the ratio is less then 0.5 or 50% when it removes capacity and cost. The third specifies that nothing happens when the ratio is between 0.5 or 50% and 0.75 or 75%. Finally, when an adaptation is made, either adding or removing capacity, there is a 'cooldown' period in which no additional adaptation can be made. This is a limitation of the underlying AWS ELB infrastructure. The fourth behavior defines that forced lack of adaptive behavior. The second adaption policy is for the case when 'spot' instances are used ($\$.FrontEndUI.Configuration.InstanceType = 1$) and is similar to the first with two key differences The first is the cost of adding a 'spot' instance is 90% less than the 'on-demand' instance and, second, the availability of a 'spot' instance is not guaranteed so there is an explicit probability definition in which the capacity provided by a 'spot' instance is available 80% of the time.

```
1   "FrontEndUI": {
2     "AdaptationPolicies": [
3       {
4         "ConfigPredicate":"#$.FrontEndUI.Configuration.InstanceType# = 0",
5         "Behaviors": [
6           {
7             "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                  ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                  ↪ >= 0.75",
8             "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                  ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# + 10 &
                  ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# + 10
                  ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
9           },
10          {
```

```
11              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                    ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                    ↪ <= 0.50",
12              "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# − 10 &
                    ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# − 10
                    ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
13            },
14            {
15              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                    ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                    ↪ > 0.50 & (#$.FrontEndUI.CurrentState.CurrentCost# /
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) < 0.75",
16              "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#"
17            },
18            {
19              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# > 0",
20              "ResultState":"#$.FrontEndUI.CurrentState.AdaptDelay# =
                    ↪ #$.FrontEndUI.CurrentState.AdaptDelay# − 1"
21            }
22          ]
23        },
24        {
25          "ConfigPredicate":"#$.FrontEndUI.Configuration.InstanceType# = 1",
26          "Behaviors": [
27            {
28              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                    ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                    ↪ >= 0.75",
29              "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                    ↪ [0.2|0,0.8|#$.FrontEndUI.CurrentState.CurrentCapacity# + 10] &
                    ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# + 1
                    ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
30            },
31            {
32              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                    ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                    ↪ <= 0.50",
33              "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# − 10 &
                    ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# − 1
                    ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
34            },
35            {
36              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                    ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                    ↪ > 0.50 & (#$.FrontEndUI.CurrentState.CurrentCost# /
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) < 0.75",
37              "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#"
38            },
```

```
39            {
40               "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# > 0",
41               "ResultState":"#$.FrontEndUI.CurrentState.AdaptDelay# =
                     ↪ #$.FrontEndUI.CurrentState.AdaptDelay# − 1"
42            }
43         ]
44      }
45   ]
46 }
```

Listing 7.4: Shopping Cart Case Study - SEAM Front End UI Specification

Database The complete specification for the $Database$ system is available in appendix A, but listing 7.5 presents the three adaptation policies for $Database$ subsystem. The first adaptation policy applies for all the hours of the day excluding 12 and 24 and defines the scaling policies for the $Database$ subsystem. The first behavior adds capacity and cost when the ratio between the $CurrentLoad$ and $CurrentCapacity$ is above 0.75 or 75%. The second behavior removes capacity and cost when the ratio between $CurrentLoad$ and $CurrentCapacity$ is below 0.5 or 50%. The third behavior specifies no adaptive action when the ratio is between 0.5, 50%, and 0.75 or 75%. The second adaptation policy specifies the scheduled adaptation of scaling up the capacity of the database in advance of the expected load due to due the nightly processing of data jobs. The third adaptation policy specifies the scheduled adaptation of removing the capacity required for the processing of data jobs.

```
1  "Database": {
2     "AdaptationPolicies": [
3           {
4              "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# != 12 &
                     ↪ #$.MetaManager.CurrentState.HourOfDay# != 1",
5              "Behaviors": [
6                 {
7                    "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                        ↪ #$.Database.CurrentState.CurrentCapacity#) >= 0.75 &
                        ↪ (#$.Database.CurrentState.CurrentCost# < #$.Database.Configuration.MaxCost#) + 10",
8                    "ResultState":"#$.Database.CurrentState.CurrentCost# = #$.Database.CurrentState.CurrentCost# +
                        ↪ 10 & #$.Database.CurrentState.CurrentCapacity# =
                        ↪ #$.Database.CurrentState.CurrentCapacity# + 10"
9                 },
10                {
11                   "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                        ↪ #$.Database.CurrentState.CurrentCapacity#) <= 0.50 &
                        ↪ (#$.Database.CurrentState.CurrentCost# < #$.Database.Configuration.MaxCost#) + 10",
12                   "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                        ↪ #$.Database.CurrentState.CurrentCapacity# − 10 & #$.Database.CurrentState.CurrentCost#
                        ↪ = #$.Database.CurrentState.CurrentCost# − 10"
13                },
14                {
15                   "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                        ↪ #$.Database.CurrentState.CurrentCapacity#) > 0.50 &
```

```
                    ↪ (#$.Database.CurrentState.CurrentCost# / #$.Database.CurrentState.CurrentCapacity#) <
                    ↪ 0.75",
16                "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.Database.CurrentState.CurrentCapacity#"
17              }
18            ]
19          },
20          {
21            "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# = 12",
22            "Behaviors": [
23              {
24                "StatePredicate":"",
25                "ResultState":"#$.Database.CurrentState.CurrentCapacity# = 200 &
                    ↪ #$.Database.CurrentState.CurrentCost# = 200"
26              }
27            ]
28          },
29          {
30            "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# = 1",
31            "Behaviors": [
32              {
33                "StatePredicate":"",
34                "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# & #$.Database.CurrentState.CurrentCost# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCost#"
35              }
36            ]
37          }
38        ]
39  }
```

Listing 7.5: Shopping Cart Case Study - SEAM Database Specification

PRISM Model

The model generated by the meta-manager for strategy synthesis is defined as a discrete time Markov chain (DTMC) that is implemented in PRISM [73] v.4.8. The meta-manager uses this model to run different Monte Carlo simulations, referred to as simulations in PRISM, each of which has a different set of parameters configured. To facilitate the strategy synthesis, the PRISM model must be updated by the meta-manager ahead of each run of simulations to provide the current state of the autonomic subsystems. A complete specification of the PRISM model with initial values can be found in appendix B, however, elements of the specifications of the global utility and properties, the environment, the front end UI, and the database, and the meta-manager will be presented here.

Global Utility and Properties The global utility definition of the PRISM model is presented in listing 7.6 and contains the definition of four similar utility functions that align to the SEAM specification of the utility functions presented in listing 7.1. The utility functions are weighted

functions, each factor with an equal 0.25 or 25% weighting with two factors for each subsystem, $FrontEndUI$ and $Database$. The first factor is the percentage difference from the ideal value, 0.66 or 66%, for the ratio of $CurrentLoad$ to $CurrentCapacity$. The further away from 0.66 the ratio of $CurrentLoad$ to $CurrentCost$ is the lower the utility score. The second factor for each subsystem is the percentage difference away from the $MaxCost$. The further away the actual value is from $MaxCost$ the higher the utility score.

```
1
2  rewards "GlobalUtility"
3    ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) <= 1) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) > 0.66) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 1):
4          (0.25 * ( 1 - ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) -0.66))) + (0.25 *
                    ↪ ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * ( 1 - ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) -0.66))) + (0.25 * ((MetaManager_Database_Config_MaxCost -
                    ↪ Database_CurrentCost) / MetaManager_Database_Config_MaxCost));
5
6
7    ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) >= 0) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) <= 0.66) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) >= 0):
8      (0.25 * (1 - (0.66 - (FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost))+ (0.25 * (1 - (0.66 -
                    ↪ (Database_CurrentLoad / Database_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_Database_Config_MaxCost - Database_CurrentCost) /
                    ↪ MetaManager_Database_Config_MaxCost));
9
10   ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) >= 0) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) > 0.66) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 1):
11         (0.25 * (1 - (0.66 - (FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * ( 1 - ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) -0.66))) + (0.25 * ((MetaManager_Database_Config_MaxCost -
                    ↪ Database_CurrentCost) / MetaManager_Database_Config_MaxCost));
12
13   ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) <= 1) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) <= 0.66) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) >= 0):
14         (0.25 * ( 1 - ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) -0.66))) + (0.25 *
                    ↪ ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * (1 - (0.66 -
                    ↪ (Database_CurrentLoad / Database_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_Database_Config_MaxCost - Database_CurrentCost) /
                    ↪ MetaManager_Database_Config_MaxCost));
```

```
15 endrewards
```

Listing 7.6: Shopping Cart Case Study - PRISM Global Utilities Specification

For each simulation PRISM must have an objective to which each individual run can converge towards. This objective is expressed in PRISM's property specification language which includes several well-known probabilistic temporal logics including PCTL [6, 7] which is used for specifying the objective in DTMCs and listing 7.7 presents the property used for meta-management. Specifically, it instructs PRISM to to maximize the $GlobalUtility$ of the model along all paths in which $MODEL\_Sink$ is true. The meta-manager constructs the model in such a manner that $MODEL\_Sink$ is set to true to end all paths of the model.

```
1 R{"GlobalUtility"}max=? [ F MODEL_Sink ]
```

Listing 7.7: Shopping Cart Case Study - PRISM Global Utilities Specification

Environment The PRISM model generated for the environment contains four behaviors corresponding to the four behaviors defined in the SEAM specification in listing 7.3 and the full PRISM specification for the environment can be found in appendix B, but two specific behaviors will be presented here. The first represents the behavior of the environment between hours 1 and 6 of the day and is the result of the specification of a 'right skew' asymmetric generalized Gaussian distribution (AGGD) and is presented in listing 7.8.

```
1 [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
        ↪ (MetaManager_HourOfDay <= 6) ->
2   0.10 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' =
        ↪ Database_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) & (MODEL_TurnCount' =
        ↪ MODEL_TurnCount + 1)
3   + 0.55 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
        ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) & (MODEL_Turn'
        ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
4   + 0.08 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
        ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) & (MODEL_Turn'
        ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
5   + 0.25 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
        ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) & (MODEL_Turn'
        ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
6   + 0.02 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
        ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) & (MODEL_Turn'
        ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
```

Listing 7.8: Shopping Cart Case Study - PRISM Environment Specification - Behavior 1

The probabilities that define the likelihood of each given action are generated by the meta-manager from the definition of the AGGD in the behavior specified in the SEAM definition. As there are potentially an infinite number of potential values, the meta-manager has a configuration option on how many options to calculate on either side of the mean. In this case the

meta-manager was configured to determine the probabilities for two points on either side of the mean at 2 and 5 units away from the defined mean. The net effect of this behavior is that the $CurrentLoad$ of the $FrontEndUI$ and $Database$ trend to being steadily increased, with potential upward and downward spikes, for hours 1 to 6 of the defined day.

Similarly, the third behavior of the $Environment$, presented in listing 7.9, defines the behavior of the environment in hours 13 to 18. This behavior is the result of the normal Gaussian distribution specified in the SEAM definition of the environment presented in listing 7.3. The net effect of this behavior is that the $CurrentLoad$ on both the $FronEndUI$ and $Database$ fluctuate around the mean value. This behavior also introduces the consistent load due to the $Database$ subsystem due to the nightly run of data processing jobs.

```
1 [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
        ↪ (MetaManager_HourOfDay > 12) & (MetaManager_HourOfDay <= 18) –>
2   0.70 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' = 150 +
        ↪ FrontEndUI_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) &
        ↪ (MODEL_TurnCount' = MODEL_TurnCount + 1)
3   + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
        ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
        ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
4   + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
        ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
        ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
5   + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
        ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
        ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
6   + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
        ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
        ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
```

Listing 7.9: Shopping Cart Case Study - PRISM Environment Specification - Behavior 3

Front End UI The PRISM specification for the $FrontEndUI$ contains eight defined behaviors which correspond to the eight behaviors defined in the SEAM specification presented in listing 7.4 and the full specification can be found in appendix B, but the same two behaviors for different settings of the autonomic subsystem configuration will be presented here. The first set of behaviors, presented in listing 7.10, defines the adaptive behavior of the $FrontEndUI$ when the $CurrentLoad$ is greater than 0.75, or 75%, of the $CurrentCapacity$, which triggers a scale up, and below 0.50, or 50%, which triggers a scale down. In both cases, the $InstanceType = 0$ which represents an 'on-demand' AWS instance and adds or removes ten units to both $CurrentCost$ and $CurrentCapacity$.

```
1 //FrontEndUI – Behavior 1
2 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
        ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) >= 0.75) &
        ↪ (FrontEndUI_Config_InstanceType = 0) –>
```

```
3 1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Upper_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Upper_10) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
4
5 //FrontEndUI – Behavior 2
6 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.50) &
                    ↪ (FrontEndUI_Config_InstanceType = 0) ->
7 1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Lower_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Lower_10) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
```

Listing 7.10: Shopping Cart Case Study - PRISM FrontEndUI Specification - Behaviors 1 & 2

The second set of behaviors, presented in listing 7.11, also defines the adaptive behavior of the $FrontEndUI$ under the same conditions for the ratio between $CurrentLoad$ and $CurrentCapacity$. However, the $InstanceType = 1$ represents the AWS 'spot' instances which adds one unit to cost and ten units to capacity, representing a 90% dicount on 'spot' instances. However, the behavior also defines the probability of the actually being able to acquire a 'spot' instance at 0.8 or 80% with a 0.20 or 20% chance of being unable to acquire one. This is a result of the explicit probability definition in the SEAM specification for this behavior presented in listing 7.4.

```
1 //FrontEndUI – Behavior 5
2 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) >= 0.75) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
3 0.80: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Upper_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Upper_1) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3)
4 + 0.20: (FrontEndUI_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (FrontEndUI_CurrentCost' =
                    ↪ FrontEndUI_CurrentCost) & (MODEL_Turn' = Database_Turn) &
                    ↪ (FrontEndUI_AdaptDelay' = 3);
5
6 //FrontEndUI – Behavior 6
7 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.50) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
8 1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Lower_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Lower_1) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
```

Listing 7.11: Shopping Cart Case Study - PRISM FrontEndUI Specification - Behaviors 5 & 6

Database The PRISM definition for the $Database$ contains five defined behaviors which correspond to the behaviors defined in the SEAM specification presented in listing 7.5 and the full specification can be found in appendix B, but two set of behaviors are presented here. Similar to the $FrontEndUI$, the first set of behaviors presented in listing 7.12, define the behavior of the database for when the hour of the day is not 1 or 12 and add or remove capacity

and cost to $CurrentCapacity$ and $CurrentCost$ depending on the ratio of $CurrentLoad$ to $CurrentCapacity$ being greater than 0.75 or 75% or below 0.50 or 50%.

```
1 //Database – Behavior 1
2 [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay != 12)  &
                    ↪ (MetaManager_HourOfDay != 1) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ >= 0.75) & (Database_CurrentCost < Database_Formula_CurrentCost_Upper_10)–>
3 1: (Database_CurrentCapacity' = Database_Formula_CurrentCapacity_Upper_10) & (Database_CurrentCost' =
                    ↪ Database_Formula_CurrentCost_Upper_10) & (MODEL_Turn' = MetaManager_Turn);
4
5 //Database – Behavior 2
6 [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay != 12) &
                    ↪ (MetaManager_HourOfDay != 1) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 0.50) & (Database_CurrentCost < Database_Formula_CurrentCost_Upper_10) –>
7 1: (Database_CurrentCapacity' = Database_Formula_CurrentCapacity_Lower_10) & (Database_CurrentCost' =
                    ↪ Database_Formula_CurrentCost_Lower_10) & (MODEL_Turn' = MetaManager_Turn);
```

Listing 7.12: Shopping Cart Case Study - PRISM Database Specification - Behaviors 1 & 2

The second set of behaviors, presented in listing 7.13, define the autonomic behavior of the $Database$ when the hour of the day is either 1 or 12. In hour 12, the $Database$ preemptively adds 200 units of capacity in advance of the nightly database processing jobs. In hour 1, the $Database$ subsystem resets the capacity of the $Database$ system to be back in-line with the load on the $FrontEndUI$.

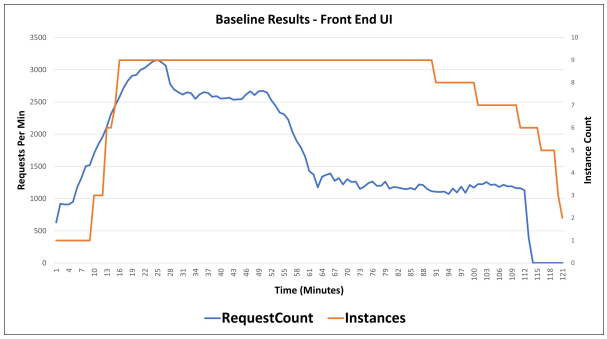
```
1 //Database – Behavior 4
2 [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay = 12) –>
                    ↪ (Database_CurrentCapacity' = 200) & (Database_CurrentCost' = 200) & (MODEL_Turn' =
                    ↪ MetaManager_Turn);
3
4 //Database – Behavior 5
5 [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay = 1) –>
                    ↪ (Database_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (Database_CurrentCost' =
                    ↪ FrontEndUI_CurrentCost) & (MODEL_Turn' = MetaManager_Turn);
```

Listing 7.13: Shopping Cart Case Study - PRISM Database Specification - Behaviors 4 & 5

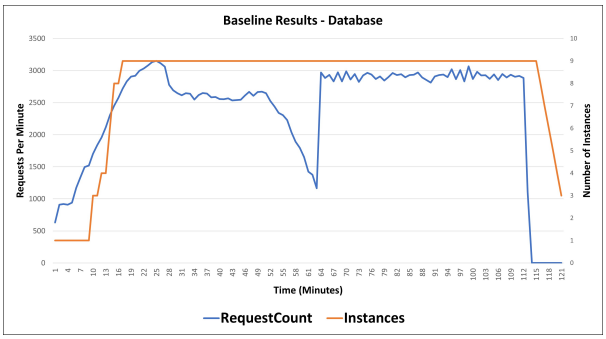
Meta-Manager The full PRSIM definition of the $MetaManager$ module can be found in appendix B, but for a DTMC in PRISM the module principally provides model administration and control functions. However, as presented in listing 7.14 the $MetaManager$ module does define the global knowledge presented in the SEAM specification in listing 7.2.

```
1 const int MetaManager_Database_Config_MaxCost = 500 – MetaManager_FrontEndUI_Config_MaxCost;
```

Listing 7.14: Shopping Cart Case Study - PRISM Meta-Manager Specification - Global Knowledge



((a)) Front End UI



((b)) Database

Figure 7.3: AWS Shopping Cart - Baseline Results

7.3 Results

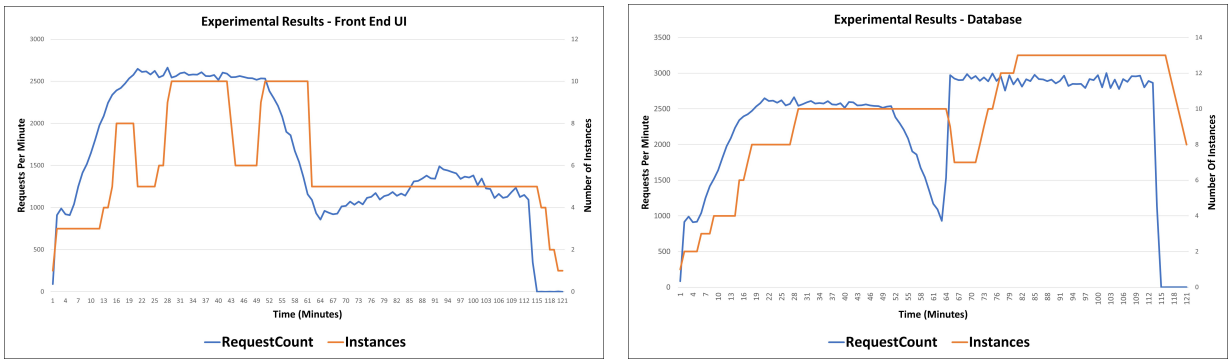
Baseline

The baseline results, presented in figure 7.3, were established by applying load to the configured *FrontEndUI* and *Database* ElasticBeanStalk (EBS) applications over a period of 96 minutes in which each 1 minute of the simulation represents 15 minutes in the daily cycle of the AWS Shopping Cart scenario presented earlier. Therefore, the simulation is composed of 24 cycles each representing an hour of the scenario and each cycle is divided into 4 intervals, each representing 15 minutes of the simulated hour. For the first 6 cycles the load is steadily increases on the front end UI which makes a query to the database system and returns the results of the request resulting in a peak of 4375 requests per minute. Over the next 6 cycles the lead steadily decreases. At cycle 13, the load between the two systems diverges as a specific load is applied to the database system, representing the nightly data processing jobs, which remains consistent through hour 24. Both of the systems add additional service capacity in correlation with the load on the system up to their default configured maximum of 10 instances and both the *FrontEndUI* and *Database* systems use on-demand instances exclusively.

As mentioned previously, this case study was selected to demonstrate the effectiveness of the meta-manager in improving the homeostatic operations of the AWS Shopping Cart system. These baseline results establish the utility generated by the collection of autonomic systems without the use of the meta-manager. The experimental results, presented next, establish the utility generated by the collection of autonomic system with the use of the meta-manager.

Experimental

The experimental results, presented in figure 7.4, were established by applying the same load profile to the same configured *FrontEndUI* and *Database* ElasticBeanStalk (EBS) applications in the same manner as the baseline results with the principal difference being that in this run, the



((a)) Front End UI

((b)) Database

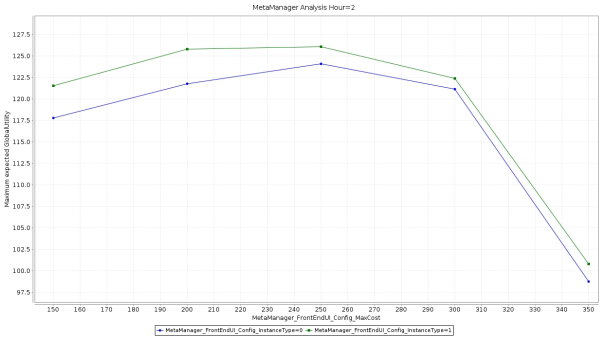
Figure 7.4: AWS Shopping Cart - Experiment Results

systems were also under the control of a meta-manager configured as described in section 7.2 with a 4 hour time horizon.

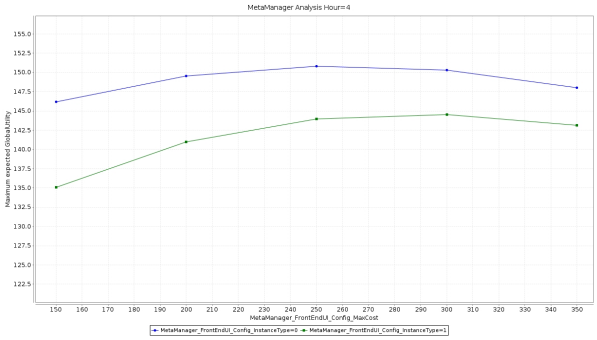
The behavior of the *FrontEndUI* system is presented in figure 7.4(a) shows a different pattern of behavior than the baseline results. Specifically, at hour 2 (time steps 8-12) the meta-manager performs an analysis and determines that the global utility of the collection of autonomic systems is maximized, specifically at 125.118 in figure 7.5(a), by the use of AWS spot instances with a balanced set of resources set at 250 for each system. Hence, the $FrontEndUI$ starts the load cycle using AWS spot instances. However, at hour 4 (time steps 16-20) the meta-manager reruns the analysis and finds that the utility is maximized, specifically at 150.819 in figure 7.5(b), using AWS on-demand instances with a balanced set of resources at 250 for each system. This change causes the EBS manager to begin to swap out the spot instances for on-demand instances which causes the temporary drop in the number of instances. Then at hour 10 (time steps 40-44) the meta-manager determines that the utility is maximized, specifically at 86.982 in figure 7.5(c), by using spot instances for the $FrontEndUI$ system and changing the balance of the resources from even to 200 for the $FrontEndUI$ and 300 for the $Database$ system. This results in the meta-manager changing the maximum number of instances for the $FrontEndUI$ from 10 to 6 and from 10 to 14 for the $Database$ system in the EBS configurations.

The change in available resources for the $Database$ system, specifically the change of the maximum number of available instances from 10 to 14, causes the behavior of the system to deviate from the baseline behavior. Specifically, the $Database$ system scales up the number of instances in use to 13 of the available 14 to ensure it has the capacity to handle the additional load placed on it from hours 12 - 24 (time steps 48-96). Each run of the simulation for the meta-manager takes between 22 and 49 seconds.

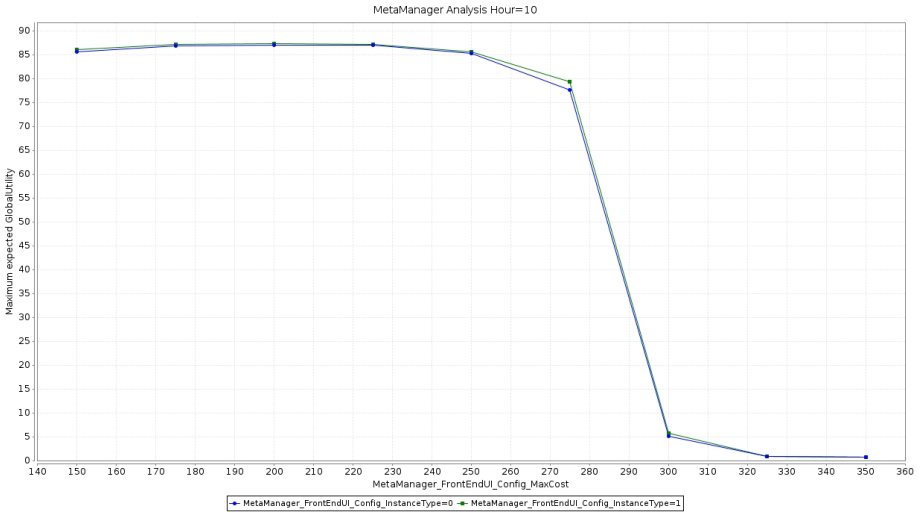
To determine the effectiveness of the meta-manager in improving the homeostatic operations of the AWS Shopping Cart test system, the global utility of the system for both the baseline and experimental test runs was compared and presented in figure 7.6(a). It shows that the meta-manager maintains a modest increase in utility for the first 12 hours (48 time steps) of the daily cycle. Examination of the components that make up the global utility, presented in figures 7.6(b-e), show that the improvement in the global utility during that period is due, principally, to the



((a)) Meta-Manager Analysis Hour = 2



((b)) Meta-Manager Analysis Hour = 4



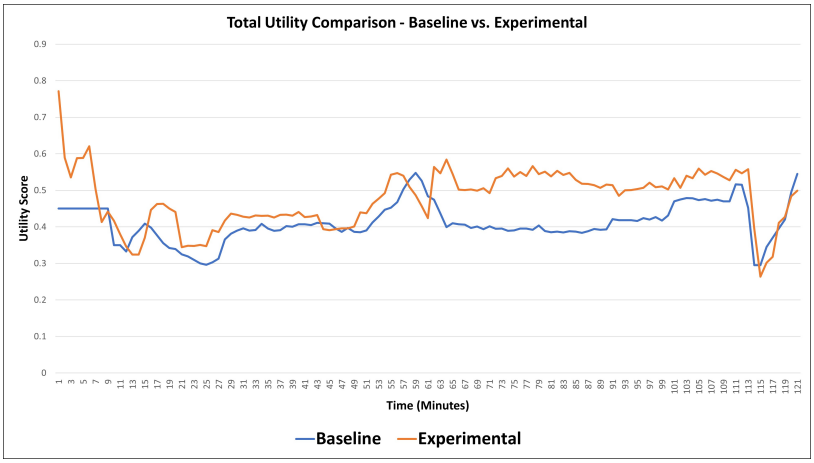
((c)) Meta-Manager Analysis Hour = 10

Figure 7.5: AWS Shopping Cart - Meta-Manager Analysis

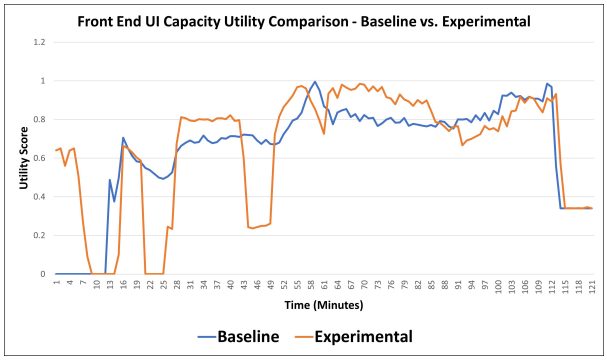
improvement in the cost to operate the system due to the use of spot instances for a portion of the daily load, see figure 7.6(c). This results in an improvement in the cumulative (integral) utility score of 29.3 to 42.4, a 44% improvement.

However, for the second 12 hour period (time steps 49-96) there are additional improvements in the global utility score due to the ability of the $Database$ system to stand up additional instances, as presented in figure 7.6(d). This results in an improvement in the cumulative (integral) utility score of 66.4 to 82.2, a 23.8% improvement.

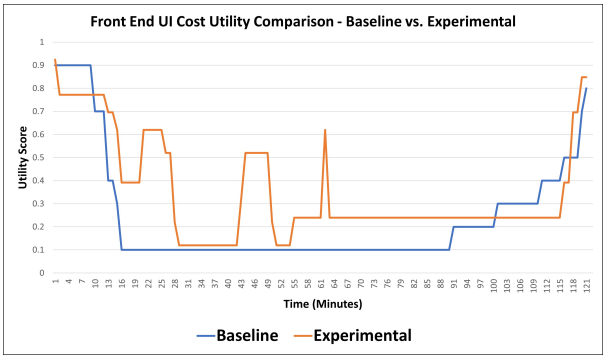
These improvements in the cost utility of the $FrontEndUI$ system and the capacity utility of the $Database$ system contribute to the improvement of the overall global utility score from 49.6 to 57.7, a 16% improvement.



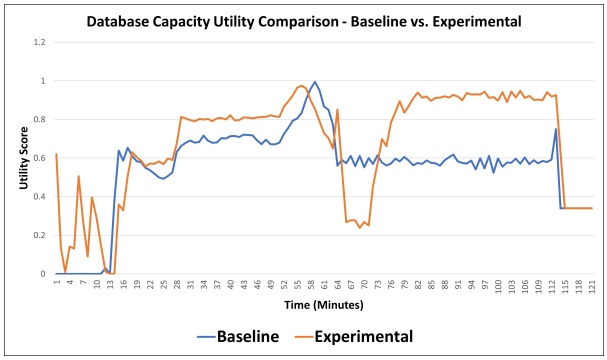
((a)) Total Utility



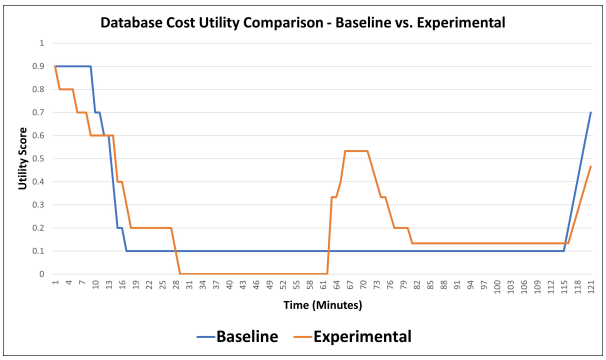
((b)) Front End UI Capacity Utility



((c)) Front End UI Cost Utility



((d)) Database Capacity Utility



((e)) Database UI Cost Utility

Figure 7.6: AWS Shopping Cart - Experiment Results

Chapter 8

Case Study: Google Control Plane

The Google Control Plane case study is based upon an actual incident documented by Google in [46] in which the system that processes changes in the networking configuration for Google Cloud customers failed. The complete information on the validity and selection of this case study can be found in section 1.2. However, this case study was selected because it presented a well documented and specific failure scenario that occurred during the period of the research of this thesis the cause of which was, partially, a result of human-centric management of a collection of autonomic systems.

8.1 Background & Context

On June 2, 2019, Google Cloud Platform (GCP) experienced a major outage in its scope, duration, and impact [52]. The outage caused network packet loss, up to 100%, resulting in an inability to access critical services for over 4 hours and caused a significant degradation of services for major websites including youtube.com, Gmail, GSuite, Nest, Snap, Discord, and Vimeo [112]. This section will detail, to the extent possible given publically available information, the architectural components and requirements of the system determined to be the root cause, how the outage progressed, the critical events during the remediation process, and a review of the preventive actions that resulted from Google's own post-mortem analysis.

Architecture

As is common with cloud providers, each customer has the ability to setup and customize a private networking space, known as an autonomous system (AS) or more commonly as a virtual private cloud (VPC) [118], for their application. Due to the high degree of customization available, changes in the networking configuration of these VPCs are commonplace which mandates that a system is continuously processing these changes and making the relevant updates to the configuration of the physical networking hardware. This system is referred to as the network control plane.

While the technical details of the network control plane are not publicly available, figure 8.1 presents what is believed to be a high-level and functionally accurate representation of the

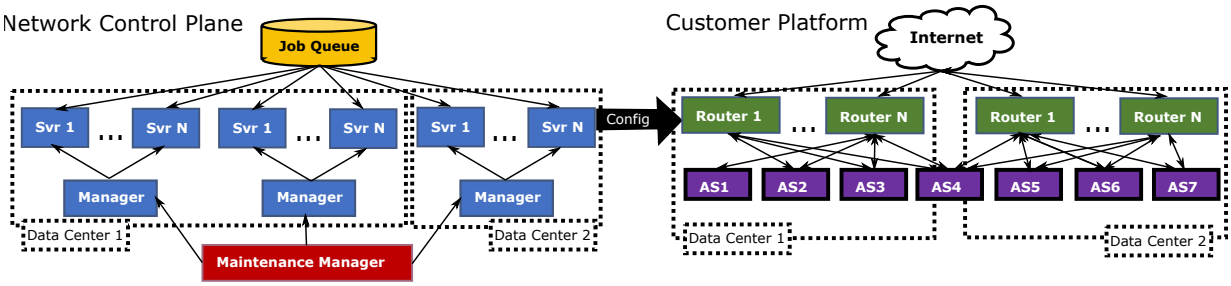


Figure 8.1: GCP Control Plane Architecture

system based on the information provided in the outage details in [52]. The network control plane consists of a set of autonomous clusters composed of individual nodes and managed by cluster management software that enables various cluster control functions, like auto-scaling. Each of the nodes will process jobs to update the network configuration as needed. The exact mechanism behind this is unknown, but it has been represented as a queue. It would not be expected for this queueing system to be dedicated to the network control plane, but instead is an enterprise wide platform serving multiple applications.

These individual clusters are run under a specific GCP service known as Google Compute Engine [51, 112] which leads to the belief that the nodes are individual virtual machines. These clusters are distributed throughout various regions and availability zones to ensure a highly available and responsive network control plane. When the network control plane finishes processing a job, the new network configuration is distributed to the relevant networking equipment to enable proper packet routing to the individual autonomous systems. Due to the expectations of customers and the criticality of changes in network configuration, the network control plane is constantly processing jobs maintaining very low processing time, anecdotally less than 5 seconds.

When maintenance of the network control plane is required, the maintenance system will either move jobs from one individual cluster to another or stop the processing of jobs and resume it after the maintenance on the cluster has been completed. In similar situations, maintenance is typically done in a 'rolling' fashion which allows one cluster to be under maintenance while others continue processing jobs, a strategy that prevents an interruption and degredation of service. Further, in the event of a failure of the entire network control plane, the physical network is setup to be 'fail static', meaning that the network will continue to run normally on the current known good configuration for a period of time to allow administrators to resolve the problem.

Outage Details

The GCP outage was the result of two misconfigurations and a software defect. Specifically, the network control plane jobs and their infrastructure were included in a specific type of automated maintenance event, the instances of the cluster management software were also included in the maintenance event type, and finally, the maintenance software had a defect allowing it to deschedule multiple independent software clusters at once, even if they were in different physical locations.

At 11:45 US/Pacific the maintenance event started and began to shutdown the clusters running the network control plane jobs. Once the network control plane failed, the physical network continued to operate normally for a few minutes. After this, the routing configuration was invalid resulting in significant packet loss, up to 100%, and end users began to be impacted between 11:47-11:49 US/Pacific. The Google engineers were alerted to the networking failures 2 minutes after it began and started their incident management protocols.

Troubleshooting of the failure was significantly hampered by severe network congestion caused by all of the consumers of GCP services which triggered the engineering team to begin another set of incident management protocols to mitigate the tool failures. Specifically, engineers had to travel to the physical data centers and reprioritize network traffic to allow the tooling traffic to take precedence. By 13:01 US/Pacific, the root cause of the incident had been determined and the engineers began to re-enable the network control plane and its supporting infrastructure. Due to the length of the outage and the shutdown of the network control plane instances across different physical locations, the configuration data had been lost and needed to be rebuilt. The rebuilt configuration for the network control plane began to roll out at 14:03 US/Pacific.

As the network control plane started to come back online, new network configurations began to roll out and service started to recover at 15:19 US/Pacific and full service was restored at 16:10 US/Pacific.

Post-Mortem Actions

In the wake of the outage, the GCP administrators took a number of short term actions to prevent an immediate reoccurrence of the problem and a number of planned changes to prevent the problem from reoccurring in the long term. First, the administrators halted the datacenter automation software responsible for descheduling jobs for maintenance events. Second, they hardened the cluster management software to reject requests to de-schedule jobs at multiple physical locations. Third, the network control plane will be reconfigured to handle the maintenance events correctly and persist its configuration so that it will not need to be rebuilt. Finally, the GCP network will be updated to continue in a 'fail static' mode for a longer period of time in the event of a loss of the control plane [52].

8.2 Experiment

To examine the applicability and effectiveness of an automated approach to meta-management, the evaluation created a representative workload against a realistic experimental platform and evaluated three scenarios: normal conditions, maintenance conditions, and maintenance conditions with a meta-manager. To determine and compare the effectiveness of the automated approach to meta-management, two measures were used: (1) the integral of the time of the oldest message in the queue, referred to as the *inverse utility*, and (2) the integral of the cost (i.e., number of servers in use per unit-time). The ideal value for each of these is the minimum that can be achieved. This appropriately evaluates the GCP case study as the control plane has the assumed quality attribute of processing the requested changes to the network configuration with minimal delay consistently over time; an inverse aggregate utility.

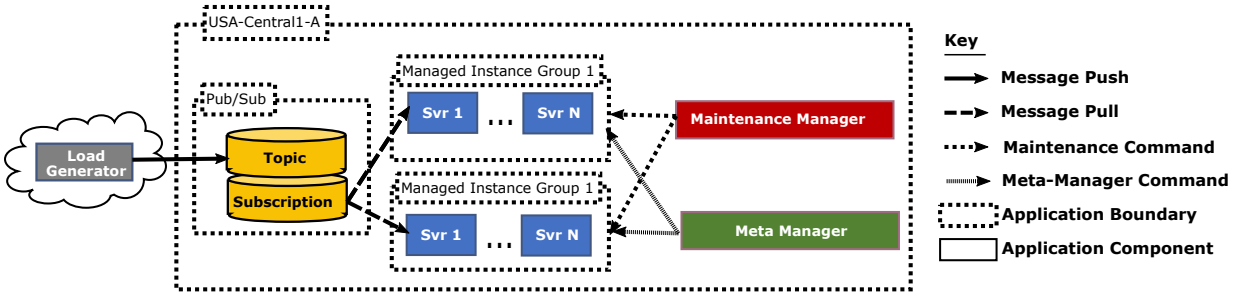


Figure 8.2: Experiment Platform Architecture

Additionally, the evaluation also examines the scalability of the approach by exploiting the fact that each of the network control plane clusters are practically identical. This enables an assumption that the set of meta-tactics appropriate for one cluster is applicable to all clusters. It is then possible to consolidate and simplify the analysis which increases the computational scalability of the approach. To test this, the scenario was re-run with maintenance and a meta-manager to determine if there was a notable change in the aggregate utility as a result of applying the proposed model scaling technique.

Platform Implementation

The experiment was designed to mimic the GCP control plane architecture described in Section 8.1. Therefore, it was instantiated on GCP using as many of the standard features as possible. The technical platform consists of five principal components, as diagramed in Figure 8.2: (1) Load Generator, (2) Pub/Sub Queue, (3) Managed Instance Groups, (4) Maintenance Manager, and (5) the Meta-Manager.

The load generator is a custom developed utility that uses the Google provided SDK to generate random well-formed messages to be placed on the Pub/Sub queue. The rate at which it places messages on the queue and the length of the run are both configurable.

The Pub/Sub queue is a standard GCP product offering [53] which provides message queueing and subscriber capabilities with guaranteed delivery mechanisms. This was configured with a single topic and a single pull subscription.

A managed instance group (MIG) is a cluster of virtual machines (VMs) that are managed by an external controller that will auto-scale the cluster depending upon configured parameters. Each instance group is configured to auto-scale to maintain the Pub/Sub metric *oldest_unacked_message_age* less than or equal to 5 seconds. This metric represents the age of the oldest message in the Pub/Sub queue. Each instance group is also configured, by default, with a 60 second cool down time between auto-scale actions and a minimum of 1 VM and a maximum of 10. Each of these VM instances is created from a base operating system template running a custom utility that is configured to check for and, if present, pull messages from the Pub/Sub queue once every second. Processing of the messages is then simulated by determining a random amount of time, between 250 and 1500 milliseconds, the system pauses before discarding the message and moving to the next.

The maintenance manager is a custom developed utility running on a static virtual machine

that uses the Google SDK to interface with the configuration and state of the MIGs. Specifically, it can place the MIG into a 'Rolling Replace' update in which each VM in the cluster is replaced by a new one.

The meta-manager is an implementation of the approach presented in this thesis that collects current state information from the managers of the MIGs, uses that information to update a model, presented in section 8.2, triggers the analysis of the model to determine the 'best' set of meta-tactics to deploy, and carries out those actions against the managed subsystems.

SEAM Specification

The experiment used a SEAM specification to describe the behaviors of the autonomic subsystems under management, specifically, the $ManagedInstanceGroups$ and the $Environment$. The complete specification can be found in appendix C, but the principal elements will be described in this section.

Global Utility The specification of the global utility for the GCP control plane uses the maximum message time allowed, 2500 ms, to determine a score between 0, the low, and 1, the high, based upon how close the oldest message time is to that maximum. The following is the specification of $GlobalUtility$ for the GCP control plane:

```
{ //Root Node
  "Global": {
    "GlobalUtility": [
      {
        "Predicate":"#$.MIG.CurrentState.OldestTimeMsg# <= 2500",
        "Formula":"(2500 – #$.MIG.CurrentState.OldestTimeMsg#) / 2500",
        "Objective":"Min"
      },
      {
        "Predicate":"#$.MIG.CurrentState.OldestTimeMsg# > 2500",
        "Formula":"0"
      }
    ]
  }
}
```

Listing 8.1: SEAM Specification - Global Utility Definition

The first entry of the global utility structure is the primary definition of the utility function and is gated by a predicate ensuring the $OldestTimeMsg$ is below 2500 ms. The second entry gives a utility score of 0 for any state in which the $OldestTimeMsg$ is above 2500 ms.

Environment The specification of the $Environment$ for the GCP control plane defines the $CurrentState$, $StateSpace$, and $AdaptationPolicy$ for the environment for the GCP control plane. The following is the specification for the $Environment$:

```
{ //Root Node
  "Environment":
  {
    "CurrentState": {
      "QueueLoad": 250
    },
    "StateSpace": {
      "Properties":
      [
        "QueueLoad": {
          "Type":"Numeric",
          "Min":0,
          "Max":500,
          "Step":50
        }
      ]
    },
    "AdaptationPolicies": [
      {
        "ConfigPredicate":"",
        "isDefault":"True",
        "Behaviors": [
          {
            "StatePredicate":"",
            "ResultState":"#$.Environment.CurrentState.QueueLoad# =
              ↪ #$.Environment.CurrentState.QueueLoad# + 250"
          }
        ]
      }
    ]
  }
}
```

Listing 8.2: SEAM Specification - Environment Definition

The $StateSpace$ defines the $QueueLoad$ property of the environment and the $CurrentState$ sets the current value of the $QueueLoad$. Finally, the $AdaptationPolicy$ for the environment adds 250 messages to the current value of the $QueueLoad$.

Managed Instance Group The specification of the managed instance group, MIG, defines the $Subsystem$ element including the $CurrentState$, $CurrentConfig$, $StateSpace$, and the default $AdaptationPolicy$ elements for the GCP control plane. The full specification for the MIG can be found in appendix C, but the following is the specification describing the default $AdaptationPolicy$ element of the full $Subsystem$ element:

```
{ //Root Node
  "MIG": {
    "InstanceCount":2,
    "AdaptationPolicies": [
      {
```

```
6             "ConfigPredicate":"",
7             "isDefault":"True",
8             "Behaviors": [
9               { //Maintenance
10                "StatePredicate":"#$.MIG.CurrentConfig.CanMaintenance# = 1",
11                "ResultState":"#$.MIG.CurrentConfig.CanMaintenance# = 0 &
                    ↪ #$.MIG.CurrentState.ServerCount# = 1"
12              },
13              { //AddCapacity
14                "StatePredicate":"#$.MIG.CurrentState.OldestTimeMsg# >
                    ↪ #$.MIG.CurrentState.MaxOldestTimeMsg# & #$.MIG.CurrentState.CoolDownTime# = 0",
15                "ResultState":"#$.MIG.CurrentState.ServerCount# = #$.MIG.CurrentState.ServerCount# + 1 &
                    ↪ #$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentConfig.CoolDownDuration#"
16              },
17              { //RemoveCapacity
18                "StatePredicate":"#$.MIG.CurrentState.OldestTimeMsg# <=
                    ↪ #$.MIG.CurrentState.MaxOldestTimeMsg# & #$.MIG.CurrentState.CoolDownTime# = 0",
19                "ResultState":"#$.MIG.CurrentState.ServerCount# = #$.MIG.CurrentState.ServerCount# − 1 &
                    ↪ #$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentConfig.CoolDownDuration#"
20              },
21              { //CoolDown
22                "StatePredicate":"#$.MIG.CurrentState.CoolDownTime# > 0",
23                "ResultState":"#$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentState.CoolDownTime# −
                    ↪ 1"
24              },
25              { //Process Jobs
26                "StatePredicate":"",
27                "ResultState":"#$.MIG.Environment.QueueLoad# = #$.MIG.Environment.QueueLoad# − (150 ∗
                    ↪ #$.MIG.CurrentState.ServerCount#)"
28              }
29            ]
30          }
31      ]
32    }
33  }
```

Listing 8.3: SEAM Specification - Managed Instance Group Adaptation Policy Definition

The $AdaptationPolicy$ specifies five behaviors labeled 'Maintenance', 'AddCapacity', 'RemoveCapacity', 'CoolDown', and 'Process Jobs'. The 'Maintenance' behavior establishes the pre-adaptation set of states to the post-adaptation set of states where the $ServerCount$ is 1. The 'AddCapacity' behavior establishes the pre-adaptation states as when the $MaxOldestTimeMsg$ is greater than the $OldestTimeMsg$ and the $CoolDownTime$ is greater then or equal to 0 to the post-adaptation set of states where the $ServerCount$ is increased by 1 and the $CoolDownTime$ is equal to the configured value. The 'RemoveCapacity' behavior is the inverse of the 'AddCapacity' behavior. The 'Cool Down' behavior establishes the pre-adaptation states for anytime the $CoolDownTime$ is greater than 0 and the post-adaptation states where the $CoolDownTime$ is one less than the previous value. Finally, the 'Process Jobs' behavior establishes any pre-adaptation state not established by any other *behavior* entry to the post-adaptation states where the $QueueLoad$ is reduced to simulate the processing of the control plane jobs. The property $InstanceCount$ is also specified at 2 which means that this $subsystem$ specification represents

two physical subsystems which is modified as part of the experiment.

PRISM Model Definition

The model used by the meta-manager for strategy synthesis is defined as a Stochastic Multi-player Game (SMG) that is implemented in PRISM-Games [22] v.2.1. This method was selected because the simplifying assumption that the meta-strategy developed for one managed instance group is applicable to all allows for a reduction of the state space and this method provides for a worst case scenario analysis for a system with high availability and reliability requirements. The meta-manager uses this model to synthesize an adaptation strategy for the meta-manager to deploy. This strategy synthesis is first attempted during the experiment runtime using data from the running system. However, if the analysis of the model cannot be completed in an experimentally relevant time horizon, the synthesis is performed off-line and loaded into the meta-manager for execution using default values. The meta-manager generates the PRISM model by populating a template that contains basic structure to properly run the model. This section will provide the details of the model and will annotate the template provided information.

Global Items The lines in listing 8.4 define two elements that are part of the PRISM template provided with the framework. The first element is the $Model\_Sink$ which provides the end state for all paths of the model. The second element is the $Model\_Max\_Turns$ which provides an upper limit on the number of turns the model can take to ensure that the model will complete.

```
1 global Model_Sink : bool init false; //Template
2 const int Model_Max_Turns = 150; //Template
```

Listing 8.4: Global Items

Reward Structures The reward structures presented in listing 8.5, represent the global utility function, $\mathcal{U}$ presented in equation 4.7. This reward structure is setup to assign the same number of 'points' to each environment state weighted by the time of the oldest message in the queue.

```
1 rewards "GlobalUtility"
2   [EnvAction1] MIG_OldestTimeMsg <= 2500: (2500 – MIG_OldestTimeMsg) / 2500;
3   [EnvAction1] MIG_OldestTimeMsg > 2500: 0;
4 endrewards
```

Listing 8.5: Reward Structures

Control Module The control module provides the administration for the model. Specifically, it tracks what the current turn is, $Model\_Turn$, the turn count, $Model\_TurnCount$, and a series of synchronized actions to update the current turn.

```
module ControlModule

   Model_Turn : [0..2] init 0;
   Model_TurnCount : [0..1000] init 0;

   [EnvAction1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG1Action1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG1Action2] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG1Action3] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG1Action4] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG1Action5] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG2Action1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG2Action2] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG2Action3] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG2Action4] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MIG2Action5] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                ↪ Model_TurnCount + 1);
   [MM] (!Model_Sink) -> (Model_Turn' = 0) & (Model_TurnCount' = Model_TurnCount + 1);

endmodule
```

Listing 8.6: Control Module

The element $Model\_Turn$ tracks which player in the game has the current turn. By default in the template, the $Environment$ has the first turn, the managed system the second, and the Meta-Manager the third. The $Model\_Turn$ element is then reset to 0 to start the process again. The $Model\_TurnCount$ element counts the number of turns to compare against the Max_Model_Turns elements to guarantee the model stops.

Player Definition In the definition of the PRISM model, there are four players: (1) Environment, (2) MIG1, (3) MIG2, and (4) MetaManager. Listing 8.7 shows the model player definitions.

```
player ENV [EnvAction1], ENVMNT endplayer
player MIG1 [MIG1Action1], [MIG1Action2], [MIG1Action3], [MIG1Action4],[MIG1Action5] endplayer
player MIG2 [MIG2Action1], [MIG2Action2], [MIG2Action3], [MIG2Action4],[MIG2Action5] endplayer
player MM [MM] endplayer
```

Listing 8.7: Player Definition

The first player is the environment which is presented in listing 8.8 and defines the element $ENV\_QueueLoad$ which is the number of messages available on the queue. The definition also defines two behaviors. The first is the behavior of the player to add additional messages to the queue. The second is a template behavior to end the model when the maximum number of turns has been reached.

```
1 global ENV_QueueLoad : [0..500] init 250;
2
3 module ENVMNT
4  [EnvAction1] (Model_Turn = 0) & (Model_TurnCount < Model_Max_Turns) -> (ENV_QueueLoad' =
       ↪ ENV_QueueLoad + 250);
5  [](Model_Turn = 0) & (Model_TurnCount >= Model_Max_Turns) & (!Model_Sink) -> (Model_Sink' = true);
       ↪ //Template
6 endmodule
```

Listing 8.8: Environment Definition

The second player, presented in listing 8.9, is the managed instance group ($MIG1$) with 5 possible actions: (1) process jobs from the queue, (2) undertake maintenance, (3) add server capacity, (4) remove server capacity, and (5) cool down after adding or removing capacity. The actions within the module align with the specification of the adaptation policy presented in listing 8.3. The formula elements in the listing set new values for individual properties by ensuring that the new value is in between the min and the max values established for the element.

```
1  const int MIG1_Model_Turn = 1;
2
3  global MIG1_ServerCount : [1..20] init 2;
4  global MIG1_OldestTimeMsg : [0..3000] init 100;
5  global MIG1_MaxOldestTimeMsg : [0..3000] init 150;
6  global MIG1_CoolDownTime : [0..2] init 0;
7  global MIG1_CanMaintenance : [0..1] init 0;
8  global MIG1_CoolDownDuration : [0..2] init 1;
9
10 formula MIG1_Formula_QueueLoad1 = (ENV_QueueLoad - (MIG1_ServerCount * 150) < 0) ? (0) :
       ↪ ((ENV_QueueLoad - (MIG1_ServerCount * 150) > 500) ? (500) : (ENV_QueueLoad -
       ↪ (MIG1_ServerCount * 150)));
11 formula MIG1_Formula_ServerCount1 = (1 < 0) ? (0) : ((1 > 20) ? (20) : (1));
12 formula MIG1_Formula_ServerCount2 = (MIG1_Server_Count + 1 < 0) ? (0) : ((MIG1_Server_Count + 1) ? (20)
       ↪ : (MIG1_Server_Count + 1));
13 formula MIG1_Formula_ServerCount3 = (MIG1_Server_Count - 1 < 0) ? (0) : ((MIG1_Server_Count - 1) ? (20)
       ↪ : (MIG1_Server_Count - 1));
14 formula MIG1_Formula_CoolDown1 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
15 formula MIG1_Formula_CoolDown2 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
16 formula MIG1_Formula_CoolDown3 = (MIG1_CoolDownTime - 1 < 0) ? (0) : ((MIG1_CoolDownTime - 1) ?
       ↪ (2) : (MIG1_CoolDownTime - 1));
17
18 module MIG
19
20  [MIG1Action1] (Model_Turn = MIG1_Model_Turn) -> (ENV_QueueLoad' = MIG1_Formula_QueueLoad1);
```

```
21
22  [MIG1Action2] (Model_Turn = MIG1_Model_Turn) & (MIG1_CanMaintenance = 1) -> (MIG1_Server_Count'
        ↪ = MIG1_Formula_ServerCount1) & (MIG1_CanMaintenance' = 0);
23
24  [MIG1Action3] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg > MIG1_MaxOldestTimeMsg)
        ↪ & (MIG1_Cool_Down_Count = 0) -> (MIG1_Server_Count' = MIG1_Formula_ServerCount2) &
        ↪ (MIG1_CoolDownTime' = MIG1_Formula_CoolDown1);
25
26  [MIG1Action4] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg <=
        ↪ MIG1_MaxOldestTimeMsg) & (MIG1_Cool_Down_Count = 0) -> (MIG1_Server_Count' =
        ↪ MIG1_Formula_ServerCount3) & (MIG1_CoolDownTime' = MIG1_Formula_CoolDown2);
27
28  [MIG1Action5] (Model_Turn = MIG1_Model_Turn) & (MIG1_CoolDownTime > 0) ->
        ↪ (MIG1_CoolDownTime' = MIG1_Formula_CoolDown3);
29
30 endmodule
```

Listing 8.9: MIG Definition

The third player, $MIG2$, is included in the complete specification found in listing D.1 as part of appendix D, however, it is functionally equivalent to $MIG1$.

The fourth player is the *MetaManager* module, presented in Listing 8.10, and it has a set of actions that represent the available meta-tactics. Each of these meta-tactics updates the configuration of *MIG*. The guards on each of these actions are identical and each part of them is strictly to ensure the proper operation of the model, they do not influence which meta-tactic could be selected. This is important, as this is the uncertainty that will be resolved by the tool when it synthesizes a strategy to determine the best use of these meta-tactics; determining the adaptation strategy the meta-manager will use. This process is what is represented by equation 4.7. The tool uses the current state of the environment, the current state of the managed system (*MIG*), and the current configuration to determine which configuration change is going to improve the global utility function, see listing 8.5. The first statement is a pass-through with no effect on other components. This allows for the MetaManager to take 'no action'.

```
1  const int MM_Model_Turn = 3;
2
3  module MetaManager
4
5    [MM](Model_Turn = MM_Model_Turn) -> (Model_Sink' = Model_Sink);
6    [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 0);
7    [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 1);
8    [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 2);
9    [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CanMaintenance' = false);
10   [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CanMaintenance' = true);
11   [MM](Model_Turn = MM_Model_Turn) -> (MIG2_CoolDownDuration' = 0);
12   [MM](Model_Turn = MM_Model_Turn) -> (MIG2_CoolDownDuration' = 1);
13   [MM](Model_Turn = MM_Model_Turn) -> (MIG2_CoolDownDuration' = 2);
14   [MM](Model_Turn = MM_Model_Turn) -> (MIG2_CanMaintenance' = false);
15   [MM](Model_Turn = MM_Model_Turn) -> (MIG2_CanMaintenance' = true);
16
17 endmodule
```

Listing 8.10: MetaManager Definition

Properties This is a rewards based property that causes the tool to search for the strategy that minimizes the reward, see section 8.2, that can be guaranteed by the players *MM* and *MIG*.

```
1 <<MIG,MM>> Rmin=? [ Fc Model_Sink ]
```

Listing 8.11: Properties

8.3 Results

Each of the four scenarios, normal conditions, maintenance conditions, maintenance conditions with a meta-manager, and maintenance conditions with a meta-manager and only one cluster modelled, was run by generating a workload of 250 messages per minute for 30 consecutive minutes with an additional 5 minute warm-up and cool-down period against an enterprise production grade cloud system, not a simulation. All scenarios were run in a single 3 hr. window to control variability in the underlying systems.

The first scenario, presented in figure 8.3, exercises the system under normal operating conditions without the interference of the maintenance manager or the assistance of a meta-manager. This scenario results in an aggregated inverse utility of 1137 with an aggregated cost of 431.

The second scenario, presented in figure 8.4, exercises the system with the interference of a maintenance manager. The maintenance manager is configured to perform a rolling restart of cluster 1 at time step 10 and a rolling restart of cluster 2 at time step 15. This scenario results in an aggregated inverse utility of 3024 with an aggregated cost of 334.

The third scenario, presented in figure 8.5, exercises the system with the interference of a maintenance manager, which is configured identically as the previous scenario, and the assistance of a meta-manager. In this scenario, the meta-manager first attempts to set the cool down period of each of the clusters from 60 seconds, the default, to the minimum available, 15 seconds at time step 7. Then at time step 12, the meta-manager configures each cluster to ignore the requests of the maintenance manager to perform rolling restarts. While the rolling restart has already been started for MIG1, this action does prevent the rolling restart of cluster 2 at time step 10. Due to the state space expansion of modelling both MIGs explicitly, it was necessary to reduce

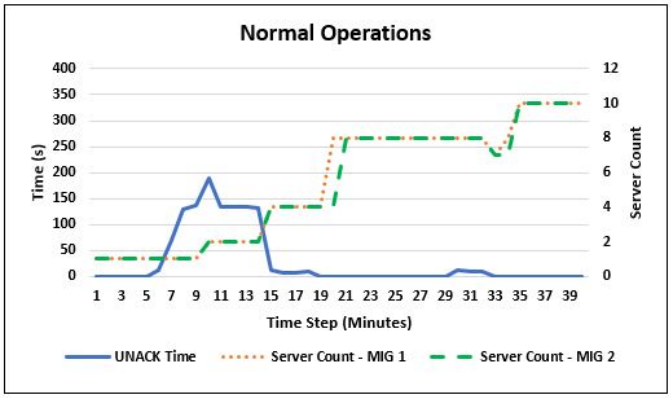


Figure 8.3: Experiment - Normal Operations

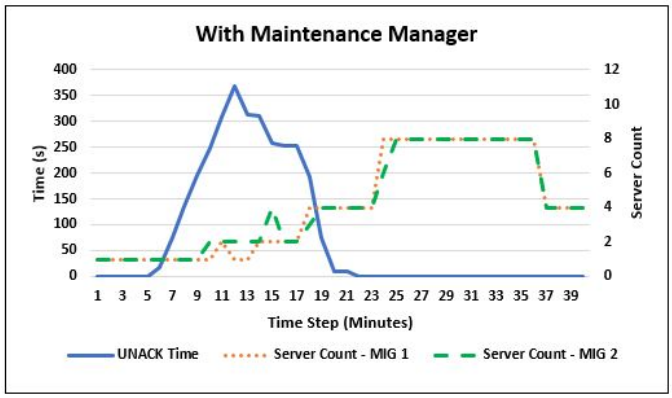


Figure 8.4: Experiment - Maintenance Operations

the *MAX_TURNS* to 40 which took a total of 642 seconds generating 12,299,356 states and 35,734,291 transitions. Because of the amount of time required to run this model, the meta-strategy was precalculated offline and preloaded into the meta-manager. This scenario results in an aggregated inverse utility of 1157 with an aggregated cost of 298.

Finally, the fourth scenario, presented in figure 8.6, exercises the system identically to the third scenario. The difference being that the model used by the meta-manager has only one representative cluster, see listing D.2, with two physical clusters, under the assumption that because the physical clusters are identical, the proposed meta-tactics appropriate for one would be applicable to all others. Similarly to scenario 3, the $MetaManager$ reconfigures the cool down period for each cluster at time step 6 and configures each cluster to ignore the maintenance manager at time step 13, again preventing the rolling restart of cluster 2 at time step 15. The model took a total of 216 seconds to generate a strategy for the $MetaManager$ generating 2,772,379 states with 6,840,217 transitions with a *MAX_TURNS* of 150. The scenario results in an aggregated inverse utility of 1095 with an aggregated cost of 269.

The experimental results show a 61.7% improvement, from 3024 to 1157, in the aggregate inverse utility between scenario 2, maintenance conditions, and scenario 3, maintenance condi-

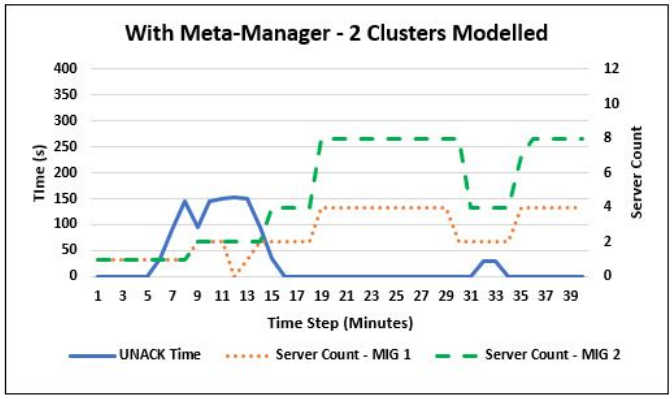


Figure 8.5: Experiment - Meta-Manager Operations - 2 Clusters

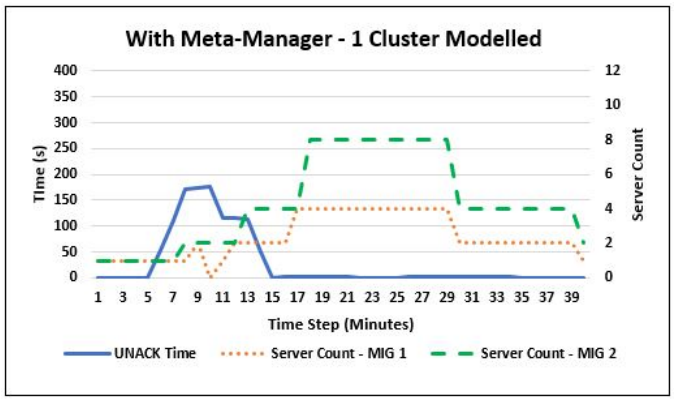


Figure 8.6: Experiment - Meta-Manager Operations - 1 Cluster

tions with a meta-manager. Further, the results also show that the meta-managed scenario had a 10.7% improvement in the aggregated cost. Additionally, this is only a 1.75% difference between the normal operating conditions, scenario 1, and the meta-managed maintenance conditions, scenario 3. Based on these experimental results, it can be reasonably concluded that the automated approach to meta-management would have improved the performance of the network control plane in the GCP case study.

Finally, the difference in aggregate inverse utility between the meta-managed scenarios, scenario 3, with 2 MIGs modelled, and scenario 4, was 5.4% decrease in aggregate inverse utility and 9.7% decrease in aggregated cost. While a 5.4% difference in the aggregated inverse utility could be significant in some contexts, it is believed that this difference is acceptable in a wide variety of contexts and can therefore conclude that the automated approach to meta-management would scale sufficiently to meet the needs of an enterprise system similar to the one presented in the GCP case study.

Chapter 9

Case Study: Electrical Grid Cascade Failure

This case study simulates the Northeast Blackout of 2003 and was selected because it presents an example of a failure of human-centric management of a collection of autonomic systems that was occurred in an industrial context outside of information technology. Please refer to section 1.2 for additional information on the selection of the case study.

9.1 Background & Context

On August 14, 2003, the Northeast of the United States and Eastern Canada experienced a significant power outage that affected 55 million people with an estimated economic impact of $6.4 billion [34, 54]. This incident was caused by a sequence of individually minor events that triggered a cascade failure of multiple electrical grids. This section will detail what caused the cascade failure including information on the basic operating conditions of an electrical grid, the sequence of events that led to the failure, how the failure progressed, and a review of the actions that could have been taken to prevent the outage from occurring. Comprehensive

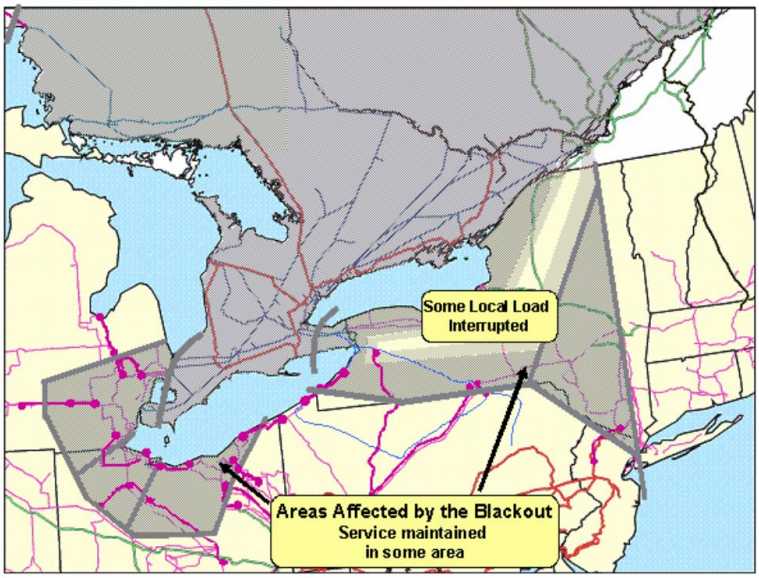


Figure 9.1: Area Affected By The Blackout

information on the failure can be found in [34, 54].

Primer on Electrical Grid Operations

Due to the physics of how energy is produced, consumed, and transmitted, an electrical grid is carefully balanced to provide the amount of electricity required to meet the demand of the customers. Simply, in AC electrical systems, this is due to Ohm's Law [1] which is stated as $V = IZ$ and relates the voltage(V) across a conductor to the current (I) and the impedance(Z) which can vary depending on a number of factors. Intuitively, voltage is the measure of the volume of electricity, current is the speed at which the volume of electricity is flowing, and impedance is a measure of how much resistance there is in transmitting the electricity.

Therefore, if there is an increase in demand for electricity, referred to as load, the generators hold the voltage steady then the impedance of the line will drop which is compensated for by an increase in the current of the system. To hold the voltage steady there are two broad categories of power generation. The first is fixed capacity generation which provides a constant source of electricity and is typically slow to react to changes. The second is reactive generation which is added and removed from the grid as needed.

Small fluctuations in load resulting in changes to voltage, current and impedance are a standard part of electrical grid operations and are routinely handled. However, under some abnormal conditions, when the current on an electrical line increases significantly, the physical properties of the line and the environment can cause the line to heat up which can cause the line to sag and come into contact with neighboring items, like trees and other foliage, and begin transmitting electricity to them; a condition referred to as a 'short circuit'. These conditions can cause damage to the lines which can be prohibitively expensive to replace. Therefore, each line is rated to carry a specific amount of current and voltage at which they are safe to operate and each end of the line is often equipped with an impedance relay. Impedance relays will 'trip' and isolate the line from the electrical grid if they detect a significant drop in the impedance of the line. If the impedance drop is due to a 'short circuit', then the load that line would normally handle is distributed across other available lines and the operations of the grid remain stable.

However, if there is a sudden increase in the demand for electricity and the combined fixed and reactive generators are unable to immediately compensate, there is a drop in the voltage on the line which also causes a drop in impedance and an increase in current. This condition will also trigger the impedance relays to protect the line with one significant difference. Since the conditions on the original line had already reached the critical mass necessary to trigger a failure of the line, as the load is distributed to other lines those conditions are likely to cause failures on those lines as well. This pattern cascades until the electrical grid fails.

Due to the interconnected nature of the individual electrical grids, often operated by different companies with competing interests, there are several independent system operators (ISOs) or regional transmission operators (RTOs) which oversee the electrical grid operations for large regions by monitoring the conditions of the individual grids and coordinate corrective actions as necessary to maintain reliable operations.

Details of the 2003 Northeast Blackout

The following is an abbreviated description of the sequence of events that led to the 2003 Northeast Blackout, a comprehensive description can be found in [54] and [34].

At 12:15pm eastern daylight time (EDT), the Midwest Independent System Operator (MISO) entered incorrect information into their state estimator (SE) and real time contingency analysis (RTCA) tools. A SE tool is used to improve the accuracy of raw sampled data from the electrical grids by mathematically processing it to make it consistent with the electrical grid models. The RTCA runs a simulation of the electrical grid using the current state of the system to evaluate various conditions. It is run on a regular schedule and if the RTCA does not have accurate information or a situation is produced outside of accepted limits an alert is generated. At this time the MISO RTCA solution produced a solution with a high mismatch because it was not configured with the information that the Bloomington-Denois Creek 230v line was out of service. This rendered the analyses produced by the system to be of little or no value to the MISO operators.

At 13:31pm EDT, the Eastlake Unit 5 generation plant (597 MW) went offline in Northern Ohio due to equipment failure. At this time, the Cleveland-Akron area elecrtical grid was able to compensate by drawing reactive power from other regional grids with a total load of 12,080MW of power load and 2,575MW (21% of total) was imported reactive power.

At 14:02pm EDT, the Dayton Power & Light's (DPL) Stuart-Atlanta 345-kV line tripped offline due to a tree contact. Due to the problems with MISO's SE and RTCA systems, the loss of this line was not properly reflected in the calculations which prevented any form of contingency analysis within its reliability zone. At 14:14 - 14:20 EDT, the First Electric supervisory control and decision analysis (SCADA) alarm and logging software failed which caused the failure of several remote EMS consoles and the operators and IT support personnel were unaware of the failure.

At 14:27 EDT, the Star-South Canton 345 kV transmission line tripped between First Energy(FE) and American Electric Power(AEP). AEP contacts FE to confirm the action, but the FE operators discuss the information as not accurate because their alerting system is not registering the failure.

At 15:05 EDT, Harding-Chamberlin 345-kV line tripped due to contact with overgrown trees within the right of in-

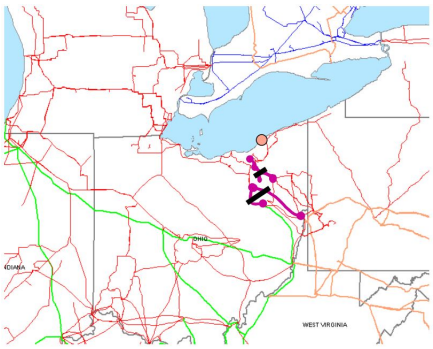


Figure 9.2: 13:31 - Cleveland-Akron Cutoff

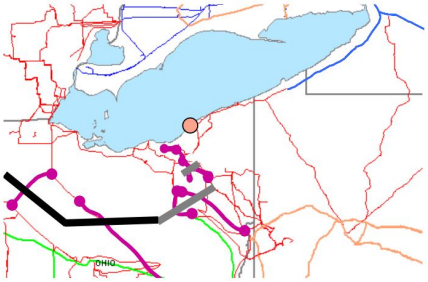


Figure 9.3: 16:08 - Ohio 345-kV Lines Trip

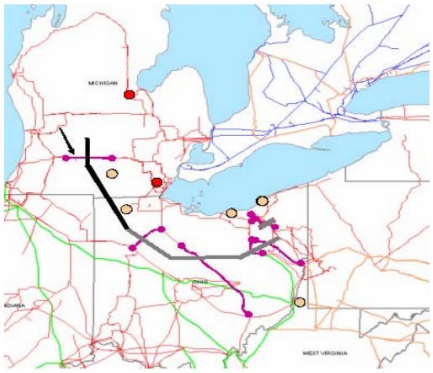


Figure 9.4: 16:10 - Eastern Michigan Trips

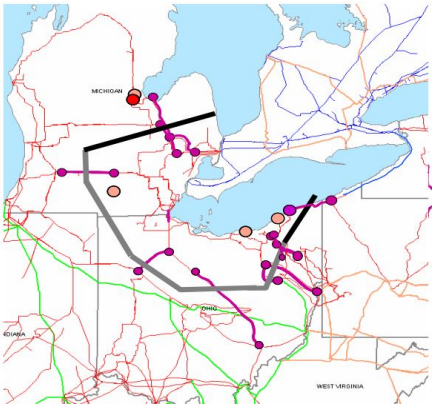


Figure 9.5: 16:11 - Michigan Trips, Ohio Separates from Pennsylvania

cursion at only 45% of its rated load.

At 15:32 EDT, Hanna-Juniper 345-kV line tripped due to contact with trees.

At 15:41 EDT, Star-South Canton 345-kV line tripped due to contact with trees.

Between 15:39 and 15:58 EDT, seven 138-kV lines tripped due to overload and sagging into underlying distribution lines.

At 15:59 EDT, the West Akron bus tripped due to a breaker failure causing another 5 138-kV lines to trip.

From 16:00 to 16:08 EDT, 4 additional 138-kV lines tripped due to overload and the Sammis-Star 345-kV line tripped due to high current and low voltage.

This constitutes the 'point of no return' for the electrical grid cascade. Up until this point, a full 2 hours and 6 minutes after the first line tripped, it might have been possible to enact mitigation measures to prevent the cascade failure. Specifically, if First Engery had shed 1,500-2,500 MW of power from the Cleveland-Akron area, the cascade might have been prevented. However, due to the lack of awareness due to the failure of the monitoring systems, no such effort was made.

From 16:08 - 16:12, the cascade failure moves from Northern Ohio into Michigan and Niagra, New York. Pennsylvannia separates from Ohio and the cascade continues to Ontario, Canada, and through out New York. New England and PJM, servicing New Jersey also separated from New York.

The collapse of the electrical grid results in several 'islands' in which the local electrical grid was disconnected from the others around it and was able to balance generation and demand through the use of under frequency load shedding (UFLS). UFLS results in the emergency disconnect of power to individual consumers in an effort to save service to as many as possible. The following are the automatic UVLS operations that day:

1. Ohio shed 1,883 MVA beginning at 16:10 EDT
2. Michigan shed a total of 2,835 MV
3. New York shed 10,648 MW beginning at 16:10 EDT
4. PJM shed a total of 1,324 MVA at 16:10 EDT
5. Ontario shed 7,800 MW at 16:10 EDT
6. New England shed 1,098 MW.

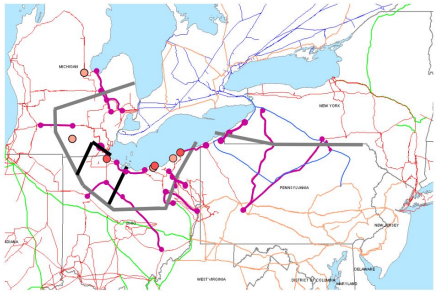


Figure 9.6: 16:11 - Cleveland and Toledo Islanded

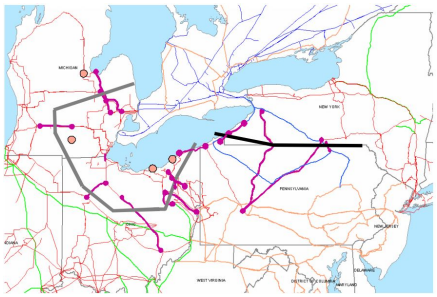


Figure 9.7: 16:11 - Western Pennsylvania Separates from New York

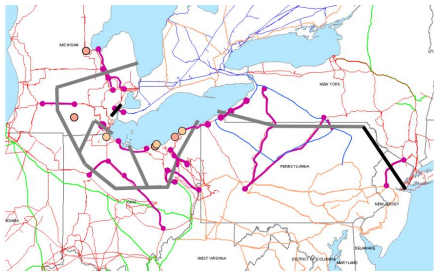


Figure 9.8: 16:12 - Northeast Separates From Eastern Interconnection

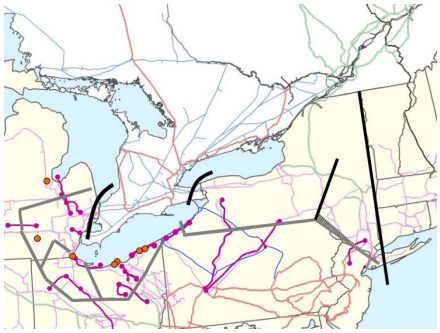


Figure 9.9: 16:13 - New York and New England Separate, Multiple Islands Form

Post-Mortem Actions

Both reports [34, 54] identify multiple recommendations and corrective actions that could prevent a recurrence of the cascade failure. Most of the recommendations involve proper vegetation management, IT system management, and other human-centric process improvements. However, one recommendation states:

> **7. Available system protection technologies were not consistently applied to optimize the ability to slow or stop an uncontrolled cascading failure of the power system.** The effects of zone 3 relays, the lack of under-voltage load shedding, and the coordination of underfrequency load shedding and generator protection are all areas requiring further investigation to determine if opportunities exist to limit or slow the spread of a cascading failure of the system. [54, p. 99]

This recommendation highlights the need for an automated solution, like a meta-manager, to detect and prevent electrical grid cascade failures.

9.2 Experiment

To evaluate the applicability and effectiveness of using a meta-manager in the context of a cascade failure of an electrical grid, a test bed was established that simulated a cascade failure that propagates across multiple electrical grids. A meta-manager was then established to monitor the conditions of the complete electrical grid, similar to the role of an independent system operator (ISO), and take action in the event a cascade failure is detected. This section is organized as follows: (1) a detailed description of the test bed, (2) the details of the electrical grid simulations and their baseline results, (3) detailed description of the implementation of the meta-manager, and (4) the results of the electrical grid simulations with the meta-manager in use.

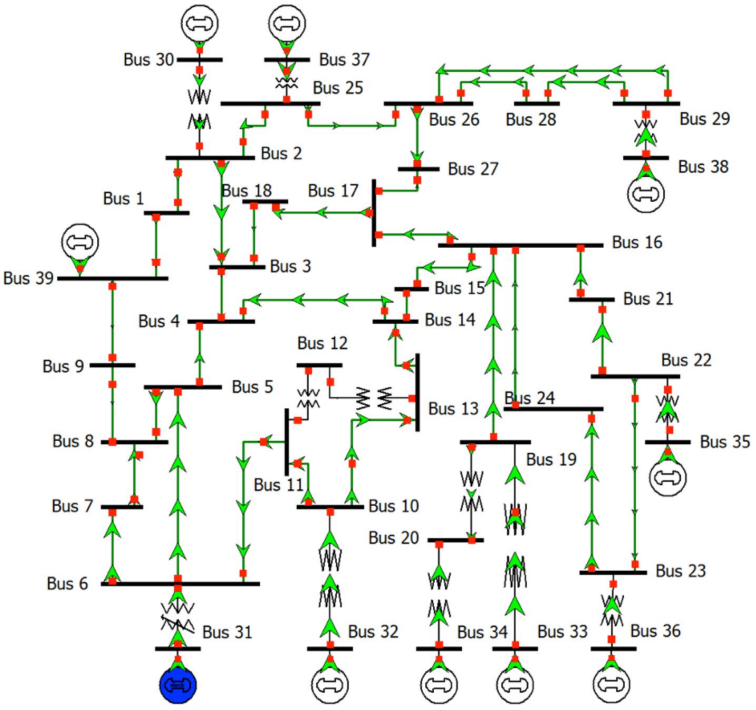


Figure 9.10: IEEE 39 Bus System Topology

Electrical Grid Test Bed

The electrical grid test bed is composed of two elements: (1) the simulation platform and (2) the electrical grid model.

Simulation Platform

The technical platform of the electrical grid test bed was established using MatLab R2023a [60] and was augmented with MatPower [81, 128] a research standard open source set of libraries for electrical grid optimization and simulation and the AC-CFM [86, 87] set of open source libraries to simulate an electrical grid cascade failure.

Electrical Grid Model

The electrical grid test bed also requires a model of an electrical grid. The test bed for this case study uses the IEEE 39 Bus system model [5, 33]. The IEEE 39 bus model is a power network in the New England area of the United States. The system consists of 10-generators, 39 bus bars,

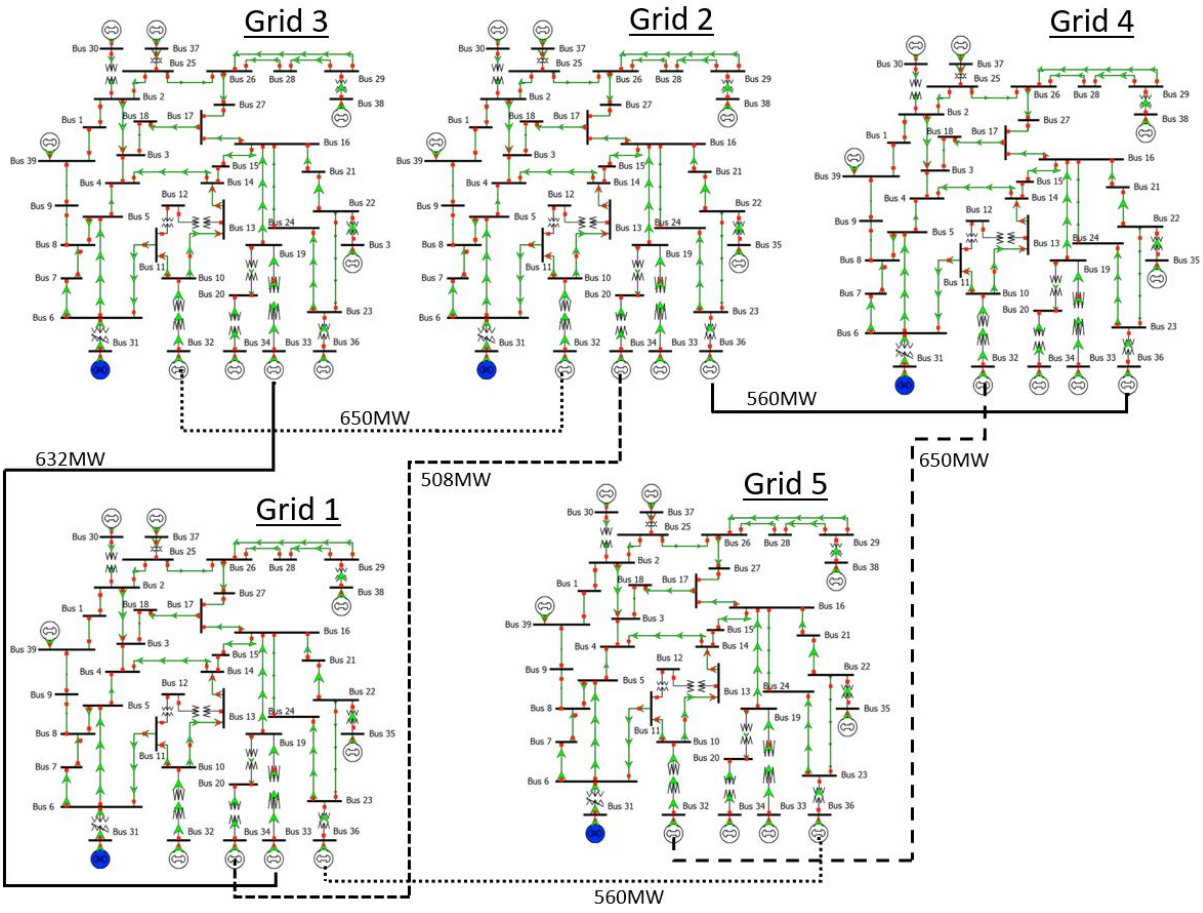


Figure 9.11: Connected Power Grid Test Bed Topology

and 12 transformers. All of the technical information on the IEEE 39 bus system can be found in appendix E but the topology of the network can be found in figure 9.10.

While the IEEE 39 bus system is a well established model of a single electrical grid, it does not represent a larger regional set of interconnected electrical grids operating in different geographic locations with different operating companies. Therefore, five instances of the IEEE 39 bus model were composed together and interconnected by creating a logical link between different power generation sources. The combined topology of the electrical grids and the interconnections between them are presented in figure 9.11.

Simulation Model

The simulation model for the electrical grid test bed is implemented in Matlab 2023a. The complete simulation model can be found in appendix F.

Initial Values The listing 9.1 presents the initial values for the simulation model.

```
%Setting the initial conditions for Grid 1, similar implementations for Grid 2 – 5
grid1 = case39;
enableGrid1 = true;
collapsedGrid1 = false;

%Sets the maximum number of loops
maxLoops = 25;

%Sets the initial conditions for the casade failure simulation
settings = get_default_settings();
settings.verbose = 0;
settings.max_recursion_depth = 1;

%Sets which bus is first to fail in the simulation
initial_contingency1 = ic;
```

Listing 9.1: Power Grid Simulation Model - Initial Values

Each of the five entries for the grids has the same initial setup. For example, $grid1$ is the variable that holds the instance of the IEEE 39 bus model for grid 1, if $enableGrid1$ is $true$ then the grid will be be analyzed for a cascade failure, and if $collapsedGrid1$ is $true$ then the analysis has determined that the electrical grid has collapsed. The $maxLoops$ variable sets the maximum number of loops the script will run before ending. The variable $settings$ establishes the initial configuration for the cascade failure analysis. The $max\_recursion\_depth$ variable of the $settings$ establishes how many iterations of the analysis will be performed. This is set to 1 so the failure analysis of each grid can be run simultaneously as elements from other connected grids fail. Finally, $initial\_contingency1$ establishes which branch of grid number 1, $grid1$, will fail first to potentially start a cascade failure.

Grid Cascade Analysis The listing 9.2 presents the code that performs the cascade failure analysis for electrical grid 1.

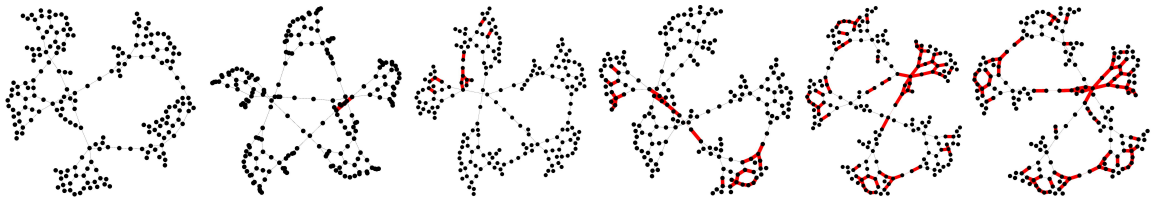
```
1  if enableGrid1 && ~collapsedGrid1
2     grid1 = accfm(grid1, struct('branches', initial_contingency1), settings);
3
4     % 1 to 2 Branch 34, Gen 34
5     if grid1.branch_tripped(34) == 1 || grid1.gen(5,8) == 0
6        enableGrid2 = true;
7        initial_contingency2 = 34;
8     end
9
10    %1 to 3 – Branch 33, Gen 33
11    if grid1.branch_tripped(33) == 1 || grid1.gen(4,8) == 0
12       enableGrid3 = true;
13       initial_contingency3 = 33;
14    end
15
16    %1 to 5 – Branch 39, Gen 36
17    if grid1.branch_tripped(39) == 1 || grid1.gen(7,8) == 0
18       enableGrid5 = true;
19       initial_contingency5 = 39;
20    end
21
22    collapsedGrid1 = nnz(ismember(grid1.G.Nodes.Type, ['success']) == 1);
23 end
```

Listing 9.2: Power Grid Simulation Model - Cascade Analysis

Each of the five electric grids has an entry in the simulation model similar to that presented in listing 9.2. The function $accfm$, on line 2, runs the cascade failure analysis for $grid1$ with the failure of a specific branch in the electrical grid as part of the $initial\_contingency1$. The other entries, on lines 5-8 for example, establish the link between the individual power grids. On line 5, if branch 35 or generator 5 fails then $enableGrid2$ is set to true which enables the cascade failure analysis for grid 2 with the appropriate initial contingency set for the element that has failed. The other lines establish similar connections from Grid 1 to Grid 3 and from Grid 1 to Grid 5. The final line, line 22, determines if the cascade analysis has completed.

Baseline Results

To establish the baseline results for the electrical grid test bed, a simulation was run in which each of the 46 branches of the IEEE 39 bus instance for grid 1 were tripped and then the cascade analysis was allowed to run to determine what the end state of the complete interconnected power grid would be. Table 9.1 gives the results of the simulation. The column 'Grid 1 Branch Tripped' is the identifier of the branch in Grid 1 that was failed to begin the simulation. The 'Total Branch Failures' column is the number of branches that tripped through out the complete electrical grid simulation model. The 'Percentage' column is the percentage of tripped branches through out the simulation model out of a total of 230, 46 branches times 5 grids. The entries in the table represent those branches that resulted in the failure of more than the initial branch.



(a) t = 0 (b) t = 1 (c) t = 3 (d) t = 6 (e) t = 9 (f) t = 12

Figure 9.12: Test Bed Simulation Results, Branch 35 Tripped

To further establish the baseline results for the electrical grid test bed, the simulation was rerun focusing on the initial condition in which the branch 35, the branch between bus 21 and 22 in 9.10, was tripped and the progressive failure of the electrical grid was observed.

The subfigure A in figure 9.12 shows the state of the complete electrical grid prior to the trip of branch 35 where the dots are individual buses and generators and the edges are the individual branches or electrical lines connecting them and black represents normal operation and red represents the failure of that component. Subfigure B shows the state of the electrical grid one time step after branch 35 is failed, this causes the failure of 3 additional branches. Subfigure C shows the failure of additional branches at time step 3 and the cascade has started to spread to another interconnected electrical grid. Subfigure D shows the state of the electrical grid at time step 6 and the branch failures are significant but still contained; this would constitute a reasonable 'point of no return'. Subfigure E shows the state of electrical grid at time step 9, the first time step in which there is at least one branch failure in all five electrical grids. Finally, subfigure F shows the final state of the electrical grid in which all five electrical grids have suffered a significant number of branch failures and multiple islands have formed due to under voltage load shedding balancing with the local generation available.

Grid 1 Branch Tripped	**Total Branch Failures**	**Percentage**
9	9	3.91%
13	13	5.65%
14	13	5.65%
18	30	13.04%
19	33	14.34%
20	28	12.17%
23	36	15.65%
27	25	10.86%
28	28	12.17%
32	28	12.17%
33	42	18.26%
34	36	15.65%
35	72	31.30%
37	20	8.69%
38	29	12.60%
39	38	16.52%
42	3	1.34%
46	18	7.82%

Table 9.1: Electric Grid Baseline Simulation Results

Meta-Manager

SEAM Specification

The experiment used a SEAM specification to describe the behaviors of the individual electrical grids including the use of global knowledge and probability distributions. The complete spec-

ification can be found in appendix G, but the principal elements, the electrical grid, the global knowledge, and the global utility will be described in this section.

Electrical Grid The SEAM specification only includes a single definition of an electrical grid, but as the adaptation policies for each of the 5 simulated grids are the same, the specification uses the $InstanceCount$ property to create the full compliment of electrical grids. The SEAM specification of the electrical grid abstracts the operations of the individual electrical grids to the percentage of the total electrical branches down, $PerBranchesDown$, which can vary between 0 and 30 percent. Under normal electrical grid operations branches can and do fail, or be taken out of service for maintenance, without significant consequences to the electric grid (e.g., 5% of branches). However, if the total number of failures of the branches reaches a critical mass (e.g., 20%), then the complete electrical grid will fail with significant indiscriminate power outages. The level of power outages are represented in the model by the $CustomerOutageLevel$ which can very between 0 (no outages) to 5 (wide spread failure). Additionally, the $CurrentConfig$ element defines one property, $EmergencyLoadShed$, which causes an electrical grid to drop pre-specified load in response to extreme conditions. These elements are defined as part of the $CurrentState$ of the electrical grid as specified here:

```
"Grid":
{
   "InstanceCount":5,
   "CurrentState": {
      "PerBranchesDown": 10,
      "CustomerOutageLevel": 0
   },
   "CurrentConfig": {
      "EmergencyLoadShed": "0"
   }
}
```

Listing 9.3: Electrical Grid SEAM Specification - Current State and Instance Count

There are two adaptation policies defined for the electrical grid. The first describes the adaptive behavior of the grid when the grid is not configured for $EmergencyLoadShed$ conditions:

```
"Grid":
{
   "AdaptationPolicies": [
      {
         "ConfigPredicate":"#$.Grid.CurrentConfig.EmergencyLoadShed# = 0",
         "isDefault":"True",
         "Behaviors": [
            {
               "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 10",
               "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                  ↪ N(#$.Grid.CurrentState.PerBranchesDown#, 0.5) &
                  ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 0"
            },
```

```
12            {
13               "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 20 &
                     ↪ #$.Grid.CurrentState.PerBranchesDown# > 10",
14               "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                     ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 0.5) &
                     ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 3"
15            },
16            {
17               "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# > 20",
18               "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                     ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, −1) &
                     ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 5"
19            }
20         ]
21      }
22   ]
23 }
```

Listing 9.4: Electrical Grid SEAM Specification - Default Adaptation Policy

The first defined behavior describes the behavior of the electrical grid with less than 10% of electrical branches down. In this case it is expected that the electrical grid will maintain the percentage of lines down as described by a normal distribution with a mean around the previous value and variance of 0.5. The second behavior describes the behavior of the electrical grid when the percentage of branches down is between 10% and 20%. Specifically, the resulting percentage of branches down will be the result of an asymmetric Gaussian distribution which is skewed to increasing the percentage of branches out. This skewed probability distribution reflects the physical nature of the electrical grid which tends to fail faster as the percentage of tripped electrical branches increases. Additionally, the increased percentage of electrical branches tripped also results in the substantial level of customer outage. The final behavior describes how the electrical grid reacts when the percentage of tripped electrical lines is greater than 20%. The resulting percentage of electrical grid lines tripped is again the result of a more severely skewed distribution and a more severe state of customer outage.

The second adaptation policy defines the behavior of the electrical grid when it is under emergency load shed conditions:

```
1  "Grid":
2  {
3     "AdaptationPolicies": [
4        {
5           "ConfigPredicate":"#$.Grid.CurrentConfig.EmergencyLoadShed# = 1",
6           "isDefault":"False",
7           "Behaviors": [
8              {
9                 "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 10",
10                "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                      ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                      ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 1"
11             },
```

```
            {
               "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 20 &
                    ↪ #$.Grid.CurrentState.PerBranchesDown# > 10",
               "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                    ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                    ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 2"
            },
            {
               "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# > 20",
               "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                    ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                    ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 3"
            }
         ]
      }
   ]
}
```

Listing 9.5: Electrical Grid SEAM Specification - Emergency Load Shed Adaptation Policy

The defined behaviors for the second adaptation policy are similar to the behaviors specified for the first adaptation policy with two differences. The first is the probability distributions are defined with a skew towards lowering the percentage of electrical branches that have been tripped and less severe customer outages. This is a result of an electrical grid taking deliberate actions to reduce load to prevent the collapse of additional electrical branches. The second difference is that the severity of the customer outages is reduced due to the deliberate action of the electrical grid.

Global Knowledge The SEAM specification includes the definition of global knowledge. Specifically, the global knowledge defines 5 correlations between the percentage of electrical branches down between the individual grids. These correlations represent the interconnections between the grids as specified here:

```
{
   "Global": {
      "GlobalKnowledge": [
         "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid2.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid1.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
         },
         "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid3.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid1.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
         },
         "Relation": {
            "Type":"Correlation",
```

```
            "Target":#$.Grid2.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid3.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          },
          "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid4.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid2.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          },
          "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid5.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid4.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          },
          "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid1.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid5.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          }
        ]
    }
}
```

Listing 9.6: Electrical Grid SEAM Specification - Global Knowledge

Each of the correlations specifies that the percentage of electrical branches down in one grid is, in part, related to the number of branches down in another grid. This is established by specifying that the branches in one grid (e.g., Grid1) are the result of a normal probability distribution on the value of the percentage of electrical branches down on another grid (e.g., Grid2). For this particular case study, only one way relationships are defined, but a two way relationship could be defined through the specification of another global knowledge correlation. Additionally, there are multiple correlations defined which influence the same target (e.g., Grid1 and Grid3 both influence Grid2).

Global Utility The SEAM specification also defines the global utility function based on the $CustomerOutageLevel$ in which a wide spread failure of the grid (i.e., $CustomerOutageLevel = 5$) has a utility score of 0 and normal operations (i.e., $CustomerOutageLevel = 0$) has a utility score of 1 and is scaled for the values in between. The following is the SEAM specification of the global utility:

```
{
  "Global": {
    "GlobalUtility": [
      {
        "Predicate":"",
        "Formula":"(5 – #$.Grid.CurrentState.CustomerOutageLevel#) / 5"
```

```
        }
      ]
    }
}
```

Listing 9.7: Electrical Grid SEAM Specification - Global Utility

Meta-Manager Analysis and Synthesis

For each time step of each run of the experimental simulation, the meta-manager generates two sets of discrete time Markov chain (DTMC) transition matrices, a matrix in which each entry is the probability of moving from one state of the system to another. The first set is one DTMC transition matrix for each electrical grid under normal operating conditions. The second set is one DTMC transition matrix for each electrical grid under emergency load shed conditions. Each DTMC transition matrix is 42 by 42 and is generated by determining the probability of moving from the current state of the individual electrical grid to the other potential states of the electrical grid by reconciling the probability distributions as specified in the SEAM specification for each of the electrical grids. Additionally, the effect on the state of each electrical grid due to the interconnections between them, as represented in the global knowledge using probability distributions in the SEAM specification, is also considered. This also allows for the specification of the $timedelay$ in the global knowledge to be considered by generating a different generation matrix the specified number of time steps in the future.

Using Matlab 2023a [60], each DTMC transition matrix is then subjected to a Monte Carlo analysis with 500 runs with a depth of 10 time steps, each beginning with the current state of the electrical grid in that time step of the simulation. The Monte Carlo analysis for each of the electrical grids varies between 44 seconds and 1 minute 39 seconds for each electrical grid on standard desktop PC hardware. The results of each Monte Carlo analysis for each grid are then analyzed to determine the most frequently occurring end state for each electrical grid. The most frequently occurring end state is then evaluated to determine the utility score for that state. Finally, if the utility score for the most frequently occurring end state for the normal operations matrix is higher than that for the emergency load shed matrix then the meta-manager sets normal operations for the electrical grid. Otherwise, the meta-manager sets emergency load shed operations for the electrical grid.

The experimental simulation was run twice, each run using different load shedding strategies. The first is referred to as 'spot load shedding' in which the electrical grid management system eliminates specific low-priority electrical demand (e.g., lower priority commercial areas instead of a hospital), in this case buses 7 and 24 which represents about 10% of total load. The second is 'Broad Load Shedding' in which the electrical grid management system eliminates a specific percentage, in this case 10% from all demand sources, of electrical demand across the grid. These are 2 of 6 potential load shedding mitigation strategies that have been identified in the context of electrical grid cascade failures [66].

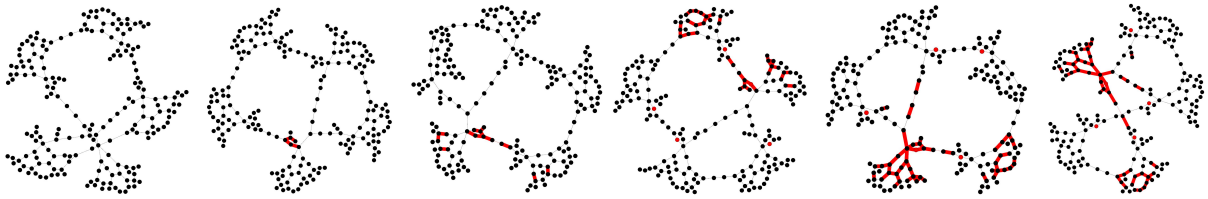
9.3 Results

The results of the simulation are presented in table 9.2 which presents the baseline results for the test bed simulation for each initial branch tripped that resulted in additional electrical lines tripped and the same results with the meta-manager running with the electrical grids using 'spot load shedding' and 'broad load shedding'. As can be observed there are four cases in which the meta-manager provides a net benefit, In case 13, a 7% improvement in the number of lines tripped, in case 18, there is a 66% improvement, and in case 19 a 63-72% improvement depending on the load shedding method. In the most severe number of total branch failures from the original test bed simulations, case 35, there is a 38 - 61% improvement. Additionally, in the other cases the introduction of a meta-manager did not either help nor harm the ability of the electrical grids to stop the cascade failure. It is possible that with additional mitigation techniques targeted to the specific nature of the failures in those cases improvement can be found.

			Spot Load Shedding		**Broad Load Shedding**	
Grid 1 Branch Tripped	**Total Branch Failures**	**Initial Failure %**	**Total Branch Failures**	**Failure %**	**Total Branch Failures**	**Failure %**
9	9	3.91%	9	3.91%	9	3.91%
13	13	5.65%	**12**	**5.21%**	**12**	**5.21%**
14	13	5.65%	13	5.65%	13	5.65%
18	30	13.04%	**10**	**4.34%**	**10**	**4.34%**
19	33	14.34%	**9**	**3.91%**	**12**	**5.21%**
20	28	12.17%	28	12.17%	28	12.17%
23	36	15.65%	36	15.65%	36	15.65%
27	25	10.86%	25	10.86%	25	10.86%
28	28	12.17%	28	12.17%	28	12.17%
32	28	12.17%	28	12.17%	28	12.17%
33	42	18.26%	42	18.26%	42	18.26%
34	36	15.65%	36	15.65%	36	15.65%
35	72	31.30%	**45**	**19.56%**	**28**	**12.71%**
37	20	8.69%	20	8.69%	20	8.69%
38	29	12.60%	29	12.60%	29	12.60%
39	38	16.52%	38	16.52%	38	16.52%
42	3	1.34%	3	1.34%	3	1.34%
46	18	7.82%	18	7.82%	18	7.82%

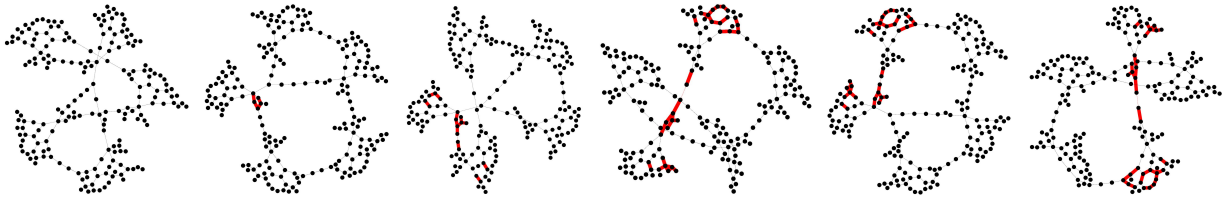
Table 9.2: Electric Grid Experimental Simulation Results

Further, figure 9.13 shows the progression of the electrical cascade failure for case 35 with the spot load shedding strategy. At time step 1 the initial branches fail and at time step 3 the cascade spreads through grid 1. At time step 6, the meta-manager has determined that all of the



(a) t = 0 (b) t = 1 (c) t = 3 (d) t = 6 (e) t = 9 (f) t = 12

Figure 9.13: Test Bed Experiment Results, Branch 35 Tripped, Spot Load Shedding



(a) t = 0 (b) t = 1 (c) t = 3 (d) t = 6 (e) t = 9 (f) t = 12

Figure 9.14: Test Bed Experiment Results, Branch 35 Tripped, Broad Load Shedding

electric grids should emergency shed load as can be seen by the small red dots in the network. At time step 9 the cascade continues to spread through grid 1, and some additional failures in grid 2. However, in time step 12 the cascade failure has stopped and electrical grids 3, 4, and 5, are largely unaffected.

Finally, figure 9.14 shows the progression of the electrical cascade failure for case 35 with the broad load shedding strategy. At time step 1 the initial branches fail and at time step 3 the cascade spreads through grid 1. At time step 6, the meta-manager has configured all electrical grids into emergency load shedding and the failure processes to grid 2. However, at time step 9 and step 12 the electrical grid cascade has been mitigated and grids 3, 4, and 5 are unaffected and both grid 1 and grid 2 have only been moderately impacted.

Chapter 10

Validation

This chapter will examine the claims of the thesis as presented in chapter 1 and provide answers to the research questions presented in chapter 2. These claims will be validated through argumentation based on the examination of the three case studies provided in chapters 7, 8, and 9. Therefore, this chapter is organized as follows: section 10.1 will examine and discuss the evidence for each of the claims, and section 10.2 will provide answers to each of the research questions.

10.1 Claims

This section will examine and provide the evidence in support of each of the claims made in section 1.1. This section is organized as follows: section 10.1.1 will examine the practicality claim, section 10.1.2 will examine the effectiveness claim, section 10.1.3 will examine the applicability claim, and finally, section 10.1.4 will examine the remaining claims of the thesis statement.

10.1.1 Practicality

Ease of Use

> *Ease of Use.* The framework will allow individuals with standard state-of-the-practice knowledge in software engineering to use it to instantiate an automated solution to manage an applicable collection of autonomic systems.

This claim is supported by two key features of SEAM. The first is that SEAM does not introduce any concepts that would be unfamiliar to a standard software engineer. Specifically, SEAM uses the same concepts in object oriented structure, data representation, boolean algebra, and basic set theory as many other general purpose programming languages such as C++, Java, the .NET languages, and others familiar to software engineers and is a standard part of education in computer science and software engineering.

Additionally, SEAM also leverages basic concepts in statistics such as probability distributions and correlations that are commonly used in software engineering projects, especially those

dealing with data and analytics. However, the selection of the appropriate meta-analysis and meta-strategy synthesis technique requires more advanced knowledge in statistics. The provided taxonomy of analysis techniques in chapter 6 can guide this decision based upon key dimensions and concerns relevant to the implementing software engineer facilitating the selection of an appropriate technique.

The second key feature of SEAM is that it provides a layer of abstraction between the implementing software engineer or administrator and the underlying meta-analysis and strategy synthesis technique and toolset. This layer of abstraction allows the software engineer to focus on the primary concerns in managing collection of autonomic systems (e.g., ensuring the performance) as opposed to having to learn additional domain-specific languages and accompanying toolsets. For example, the SEAM compiler can ensure the that the variables in PRISM can never be assigned a value outside of their defined limits, a common run-time error in the use of PRISM and PRISM-Games. Additionally, this abstraction also allows an engineer to potentially change which meta-analysis and meta-synthesis technique and toolset with minimal changes to the SEAM specification.

Human Feasible Configuration

> *Human Feasible Configuration.* The framework will provide methods that will allow a human administrator to specialize a meta-manager to a particular collection of autonomic systems including the specification of adaptive behavior for each subsystem, referred to as an adaptation policy.

This claim has two parts that must be supported. The first is that the framework is capable of specializing a meta-manager to a particular collection of autonomic systems. The second is that the method of specializing the meta-manager is feasible for a human administrator. Both parts of the claim are supported through argument based upon an examination of the SEAM specification of the three provided case studies.

AWS Shopping Cart The complete SEAM specification for the AWS Shopping Cart case study can be found in appendix A. It includes 296 total lines of SEAM specification of which 124 lines are starting and ending brackets separated into individual lines to facilitate human readability leaving 172 lines of functional SEAM content. Within those 172 lines of SEAM content are four principal objects specialized to this collection of autonomic systems. The first is the specification of the $Environment$, see listing 7.3, in which the behavior of the environment on the current load of both the $FrontEndUI$ and $Database$ subsystems is specified through the use of statistical distributions that define the behavior of the environment for different hours of the day. The second object is the $FrontEndUI$, see listing 7.4, which specifies the adaptive behavior of the subsystem based upon the state of the managed system, specifically the current load, and how the subsystem adapts to ensure performance against its individual defined QoS objectives. Similar to the $FrontEndUI$, the $Database$ subsystem object, see listing 7.5, also defines a similar, but different set of autonomic behavior based upon the state of the managed system. While both systems will add capacity in response to increased load, the nature of the

capacity added is different because of the stateful nature of the $Database$ subsystem. Finally, the fourth object is the $MetaManager$ which contains the $GlobalUtility$ specification that provides the QoS objective for the collection of autonomic systems and the definition of the meta-tactics specific to this collection of autonomic systems.

GCP Control Plane The complete specification for the Google Control Plane case study can be found in appendix C. It includes 142 total lines of SEAM specification of which 51 lines are starting or ending brackets separated into individual lines to facilitate human readability leaving 91 lines of functional SEAM content. Those 91 lines contain the specification of three objects specialized to this collection of autonomic systems. The first is the $Environment$, see listing 8.2, which describes the addition of new network control plane messages onto a queue as described in the Google incident report [52]. The second is the MIG, see listing 8.3, or managed instance group which specifies the autonomic response of an individual cluster to the increase or decrease in the number of messages in the queue. The third is the $MetaManager$, see listing 8.1, which specifies the $GlobalUtility$ object that provides the QoS definition and meta-tactics specific to this collection of autonomic systems.

Electrical Grid Cascade Failure The complete specification for the Electrical Grid Cascade Failure case study can be found in appendix G. It includes 131 total lines of code of which 48 lines are starting or ending brackets separated into individual lines to facilitate human readability leaving 83 lines of functional SEAM content. Those 83 lines contain the specification of two objects specialized to this collection of autonomic systems. The first is the $Grid$, see listings 9.4 and 9.5, in which the autonomic behavior of the electical grid is specified under two different configurations of its autonomic manager; Normal Operations and Emergency Load Shed. The second object is the $MetaManager$, which defines the $GlobalUtility$ specific to this collection of autonomic systems, but more importantly defines the $GlobalKnowledge$, see listing 9.6, which contains the relationships between the individual subsystems (i.e., electrical grids). It is also important to note that in this case, the environment plays no role in this model of the autonomic behaviors of the individual electrical grid, and therefore it is not included in the SEAM specification.

In each of the three case studies, the meta-manager was specialized through the use of a SEAM specification for the collection of autonomic systems it was intended to manage, supporting the first part of the claim. In [65], Caspers Jones evaluated numerous development methodologies (e.g., RUP, XP, Agile, Waterfall) and programming languages over thousands of projects and determined that programmers write between 325 and 750 lines of code per month. The lines of SEAM code required to specialize a meta-manager to a particular collection of autonomic systems ranged between 83 and 172, well within what would be considered feasible for a single human administrator to be able to accomplish, supporting the second part of the claim.

Scalability

Scalability. The framework will be capable of scaling to handle systems of practical industrial size.

The scalability of the approach is dependent upon the meta-synthesis technique used to generate the meta-strategy. This is addressed by:

- The simplification of the state space that occurs when an autonomic manager is introduced to the managed systems
- Allowing for the selection of the meta-synthesis technique used to generate the meta-strategy
- The automated approach to meta-management presented in this thesis does not subsume control of the autonomic control of the individual subsystems
- The approach can take advantage of optimizations available in the context

Each of the case studies supports the scalability claim by leveraging these elements on a production grade system or research grade simulation to select and use a meta-synthesis technique that was able to generate a meta-strategy on a time scale appropriate for the context.

AWS Shopping Cart The experimental platform is established using AWS provided code for a shopping cart and runs on AWS cloud based systems that are, by default, ready for industrial production operations. Additionally, increasing the scale, either horizontally or vertically, would have little impact on the ability of the meta-manager to optimize the homeostatic operations of the platform. Finally, the meta-analysis and meta-strategy synthesis technique, a DTMC with simulation, took between 22 seconds and 49 seconds in each of the runs to produce a result. Therefore, even if autonomic subsystems and/or configuration options were added to the model to handle additional industrial concerns, it is reasonable to conclude that either the current technique would still be acceptable or the current technique could be tuned to align with the specific needs of the collection of autonomic systems under management (e.g., reducing the number of simulation runs).

Google Control Plane Similar to the AWS Shopping Cart, the experimental platform is established using the likely GCP services, or similar, used by the actual GCP control plane as supported in the GCP incident report [122] which are, by default, ready for industrial production operations. Additionally, increasing the number of control plane clusters would have little impact on the ability of the meta-manager to handle the instability caused by the maintenance system in this case study. Unlike the AWS Shopping Cart, since each of the autonomic subsystems is practically identical, it is reasonable to assert that the meta-strategy generated for one subsystem would be applicable to all. This means it is reasonable to conclude that adding additional subsystems would have little impact on the ability of the meta-manager to effectively manage the collection of autonomic control plane clusters. Finally, the meta-analysis and meta-strategy synthesis activities completes in 3 minutes and 36 seconds (216 seconds). In this situation, even if the meta-manager is unable to prevent any negative effects within the 4 minutes between initial action and full impact, due to being configured for homeostatic operations, it can perform mitigation actions to limit the damage well before human administrators which took 1 hour and 13 minutes. Further, the use of different analysis techniques (e.g., a DTMC with simulation vs. a Stochastic Multi-Player Game) which might be more effective for handling instabilities

in the Google Control Plane could further optimize the ability of the meta-manager to handle instabilities in this case study.

Electrical Grid Cascade Failure The experimental platform is established using simulation and analysis techniques and frameworks that are commonplace in actual electrical grid operations. It is documented in [34, 54] that Independent System Operators (ISOs) and Regional Transmission Operators frequently run simulations of electrical grid operations over models of the managed electrical grid because it was a misconfiguration of one of those systems that contributed to the 2003 Northeast Blackout. While the models of actual electrical grids are unavailable, due to security concerns, the IEEE 39 bus model is reasonably representative of an electrical grid of practical industrial size. However, since the meta-analysis and meta-strategy phases of the experimental runs varied between 44 seconds and 1 minute and 39 seconds, even if the models were increased in both size and complexity, it would be expected that the meta-manager could complete the necessary analysis in the time frame required to be effective given the period of 'hours' that often precede the 'point of no return' in electrical grid cascade failures [34, 54].

10.1.2 Effectiveness

Improved Performance

> *Improved Performance.* A collection of autonomic systems managed by an autonomic manager will experience improved performance against defined global quality objectives over human-based management.

Each of the three case studies supports the claim because in all three the meta-manager performed actions that could have and/or should have been done by human administrators and improves the global utility over the baseline results presented in each of the case studies.

AWS Shopping Cart The purpose of the AWS Shopping Cart case study was to evaluate the effectiveness of the meta-manager to improve the homeostatic operations of a collection of autonomic systems, specifically an exemplar shopping cart system. The test system was subjected to a load profile that mimics a daily cycle of load. Specifically, the first 12 hours of the day, representing the period between 6am and 6pm, the $FrontEndUI$ and the $Database$ experience a steady increase in load up to a peak at 12pm and then a steady decrease in load to 6pm. However, at 6pm, the $Database$ system experiences a constant load representing the nightly processing of database jobs for the remaining 12 hours of the 24 hour day.

This load was applied for two scenarios. The first was the baseline scenario which established the performance of the collection of autonomic systems under the applied load without the benefit of the meta-manager. The second was the experimental scenario in which the collection of autonomic systems was under the control of a meta-manager. The meta-manager was able to improve the configuration of the $FrontEndUI$ system during the first 12 hour period by using a combination of spot instances and on-demand instances to improve the cost dimension of the $FrontEndUI$ by 44%, from a cumulative (integral) utility score of 29.3 to 42.4. In the

second 12 hour period, the meta-manager was able to improve the capacity dimension of the $Database$ system by reallocating resources from the $FrontEndUI$ to the $Database$ system. This allowed the $Database$ system to deploy 13 instances, instead of the baseline 10, which allowed it to better meet the QoS objectives of the system. Specifically, the capacity dimension of the $Database$ system improved 23.8%, from a cumulative (integral) utility score of 66.4 to 82.2. These improvements contributed to a 16% improvement in the overall utility of the system from a cumulative (integral) utility score of 49.6 to 57.7 over the baseline scenario.

Google Control Plane In the Google Control Plan case study four scenarios were run each trying to minimize the the inverse utility score representing the time in queue of the oldest message and the aggregated cost. The first baseline scenario, presented in figure 8.3, was the baseline scenario under normal maintenance operations and generated a utility score of 1137 with a cost of 431. The second baseline scenario, presented in figure 8.4, runs the system with the interference of the maintenance manager resulting in a utility score of 3024 and a cost of 334.

The first experimental scenario has a meta-manager, configured with two managed instance groups, operating on the collection of control plane clusters which are also subject to interference from the maintenance manager. This scenario resulted in a utility score of 1157 with an aggregated cost of 298. This is a 61.7% improvement in utility and a 10.7% improvement in cost over the second baseline scenario. This first experimental run also has only a 1.75% difference between the system running normally without a maintenance manager and the meta-managed scenario in which the maintenance manager is active.

The second experimental scenario has a meta-manager, configured with only one managed instance group specified, operating on the collection of control plane clusters which are also subject to interference from the maintenance manager. This scenario is investigating the validity of the assertion that multiple practically identical systems can be modeled as one entity and the resulting meta-strategy can be applied to all physical cluster instances and still be effective. This scenario generated an inverse aggregate utility of 1095, a 5.4% decrease, with an aggregated cost of 269, a 9.7% decrease from the scenario with 2 MIGs modelled. In some scenarios, the 5.4% decrease in utility might be significant, but this thesis argues that this difference would be an acceptable tradeoff in many contexts to provide additional scaling options for the meta-analysis and meta-strategy synthesis techniques.

Electrical Grid Cascade Failure In the Electrical Grid Cascade Failure case study three scenarios were run, the details of which are presented in table 9.2. In each scenario the goal of the meta-manager was to maximize the global utility by reducing the number of electrical grid branches that are tripped as part of the cascade failure. The first was a baseline simulation in which each of the branches in Grid 1 were tripped to determine if they would cause a cascade failure. The experiment found that the failure of 18 branches caused additional branches to fail ranging from 2 (1.34%) to 71 (31.30%) additional branches.

In the two experimental scenarios, the same simulation was run, this time with a meta-manager evaluating the state of the electrical grids with the option of triggering an emergency load shed to bring the electrical grids into balance and prevent further spread of the cascade failure. In the first experimental scenario, the meta-manager implemented a ‘spot load shedding’

strategy in which lower priority load is eliminated. This results in an improvement in four cases with improvement ranging from a 7% to 66%. In the most severe failure case, branch 35, the number of branches tripped from 72 to 45 resulting in a 38% improvement.

In the second experimental scenario, the meta-manager implemented a 'broad load shedding' strategy in which 10% of load was reduced from all sources. This results in an improvement in the same four cases with improvement ranging from 7% to 72% improvement. In the most severe failure case, branch 35, the number of branches tripped drops from 72 to 28 resulting in a 72% improvement. In none of the cases in either experimental run did the actions of the meta-manager cause additional harm to the electrical grid.

Timeliness and Assurance

> *Timeliness and Assurance.* Because the framework and approach do not mandate a specific synthesis technique, an engineer implementing the framework can select a synthesis technique that best fits the level of timeliness and assurance required for the context.

This claim is supported through argument based upon an examination of the three case studies. Specifically, each of the case studies uses a meta-analysis and meta-strategy synthesis technique that meets both the assurance and timeliness requirements of the context.

AWS Shopping Cart The collection of autonomic subsystems that form the AWS Shopping Cart operate in an environment with potentially large and frequent changes in load that will vary amongst the individual subsystems. In this context, the consequences of less-than-ideal operation are economic by causing higher costs than are strictly necessary or reduction in revenue. Therefore, a DTMC with simulation was selected as it can analyze more complicated models relatively quickly by approximating the most likely result. This approximation is sufficient as the speed of analysis is more important than the level of assurance the technique would provide.

Google Control Plane In the context of an infrastructure IT system, the collection of autonomic subsystems does not have large or frequent changes in its load due to environmental actions. However, the system must have a high degree of reliability so a conservative approach to adaptation is appropriate to minimize the potential for disruption. As such, the meta-manager in the Google Control Plane case study is configured to use a Stochastic Multi-Player Game to determine what the best achievable global utility would be assuming the environment takes every action to try and minimize it. This method of meta-strategy synthesis is more sensitive to the size of the model and, consequently, the number of states it generates. As such, it generally takes more time to determine the most appropriate strategy, but it also provides a high degree of assurance in the result as it is an exhaustive state space approach. This trade-off is acceptable and fits the timeliness and assurance requirements of this context.

Electrical Grid Cascade Failure In the context of an electrical grid failure, the meta-manager is concerned with handling a system instability and the potentially extreme consequences. How-

ever, the size of the electrical grids and the models required to describe them would be impractical for some meta-synthesis methods (e.g., Stochastic Multi-Player Game). Therefore, a discrete time Markov chain (DTMC) with a Monte Carlo simulation was used. This has the advantage that the time taken to do the analysis is manageable even for large model sizes and would be possible within the 'hours' time frame of the initial stages of a cascade failure. However, a DTMC with Monte Carlo simulation samples the state space to give the most likely result, which is a lower level of assurance than other techniques (e.g., Stochastic Multi-Player Game). Given the nature and consequences of electrical grid failures, if the likelihood of one is sufficiently high to merit meta-adaptation, then a controlled outage, like emergency load shedding, is preferable to the uncontrolled outage even if the grid might not have eventually experienced a cascade failure.

10.1.3 Applicability

> The framework will be applicable to a significant subset of collections of autonomic systems with the following characteristics:
>
> - Each subsystem is non-adversarial in nature.
> - Each of the subsystems provides an interface to adjust the configuration parameters of the autonomic managers.
> - It is possible to elaborate the adaptive behavior that each autonomic subsystem will employ for a given state of the environment under a set of configuration parameters.

This claim will be supported through argumentation based upon the examination of the variation across the key properties of the case studies as presented in table 1.1.

Functional Area The three case studies represent two areas of IT-centric systems, consumer facing and backend systems, and one industrial system in the electrical grid. It is reasonable to assert that the functional requirements of each of these systems are shared by a significant subset of other collections of autonomic systems in other contexts. For example, in a manufacturing environment (e.g., automobile) the meta-manager could be managing a set of practically identical robots, similar to the GCP Control Plane example, in which it is configured to handle system instabilities because of limited optimization potential, similar to the electrical grid case study. Further, in bespoke manufacturing (e.g., manufacturing of drones) it is necessary to power up multiple machines to produce potentially a single custom part and there is significant opportunity to continuously improve the power consumption operations of different machines working together in a single process, similar to the AWS Shopping Cart case study. Therefore, one can conclude that the functional properties of the three case studies represent a significant subset of autonomic systems.

Instability & Homeostatic The three case studies examine 3 of the potential 4 combinations between the meta-manager's ability to improve performance against instabilities or improve homeostatic operations. Specifically, the AWS Shopping Cart case study examines homeostatic

operations, the Google Control Plane examines handling of an instability in the context of homeostatic operations, and the Power Grid Cascade Failure only examines the handling of a system instability. In the fourth possible combination, it would not be valid to expect a meta-manager to improve homeostatic operations while being configured for the handling of a system instability. Therefore, it is reasonable to conclude that the ability of a meta-manager to handle both homeostatic operations and system instabilities makes the approach applicable to a large subset of collection of autonomic systems.

Autonomic Manager Across the three case studies, the meta-manager operates over four autonomic managers: specifically, the AWS ElasticBeanStalk and DAX Autoscaling in the Shopping Cart case study and in the Managed Instance Groups in the GCP Control Plane case study. In all three cases the meta-manager interacted with the actual production services available to customers of both AWS and GCP which provide autonomic capabilities for thousands of their customers. While the actions of the SCADA system were simulated in the Power Grid case study, the SCADA control systems have, or should have, the ability to perform the simulated actions as mentioned in [34, 54] and that SCADA systems are popular in many industrial contexts. Due to the diversity in the autonomic managers and the individual popularity of those managers, one can conclude that a meta-manager is capable of operating over a sufficiently large number of autonomic management systems to make the approach applicable to a significant subset of collections of autonomic systems.

Synthesis Method & Toolset Each of the three case studies uses a different combination of analysis technique and toolset. Both the AWS shopping cart and the Electrical Grid Cascade Failure use a discrete time Markov chain (DTMC) with simulation. However, the AWS Shopping cart uses PRISM [73] and the Electrical Grid case study uses Matlab 2023a [60]. Additionally, the Google Control Plane case study uses a Stochastic Multi-player Game (SMG) in PRISM-Games [22]. The variation of the synthesis techniques and the toolsets used across the three case studies supports the assertion that the automated approach to meta-management presented in this thesis does not mandate a specific analysis technique or tool set. Therefore, it is reasonable to conclude that since an administrator can select the analysis method and tool set appropriate to their context, the approach is applicable to a significant subset of collections of autonomic systems.

10.1.4 Thesis Statement

This section provides the support for the individual claims of the thesis statement that are not part of practicality, effectiveness, or applicability.

> We can provide engineers the ability to establish an automated solution...

This statement is validated by the individual case studies presented in chapters 7, 8, and 9 because in each of them an engineer was able to establish an automated solution to provide meta-management to a collection of autonomic systems.

1. An automated approach to the management of collections of autonomic systems.

The automated approach to meta-management is provided in chapter 4 and was implemented as part of the case studies presented in chapters 7, 8, and 9.

2. A domain specific language used to abstract and represent the adaptation behavior for each autonomic subsystem.

The domain specific language, SEAM, is provided in chapter 5 and was used as part of the case studies presented in chapters 7, 8, and 9.

3. Guidance to determine the appropriate strategy synthesis technique for the context in which the collection of autonomic system is operating.

The guidance to determine the appropriate strategy synthesis technique is provided as part of the taxonomy of analysis and synthesis methods presented in chapter 6.

4. A reusable software framework that simplifies the development of a meta-manager.

In addition to the case studies presented in chapters 7, 8, and 9, the approach was also implemented in conjunction with the Rainbow [24] framework, as Rainbow has been used in numerous research efforts across multiple domains, see [15, 16, 17, 19, 83], this provides evidence for the claim of the framework being reusable.

10.2 Research Questions

Research Question 1: How to provide assurance on the behavior of the collection of autonomic systems?

The assurance on the behavior of the collection of autonomic systems is provided by the selection of the meta-analysis and meta-strategy synthesis technique and the required level of assurance will vary depending on the context in which the collection of autonomic systems is operating. More information on the selection of the appropriate technique can be found in chapter 6.

Research Question 2: How to enable the practical analysis of adaptation policies given the uncertainty in the future state of the managed system?

Uncertainty in the outcomes of the adaptive behaviors is handled by leveraging the specifications of probability distributions. This thesis examines four potential candidates: Normal Distribution(N), Synchronous General Gaussian Distribution (SGGD), Asynchronous General Gaussian Distribution (AGGD), and explicitly defined. Each of these are straightforward to specify and provide a mechanism to inform the meta-analysis and meta-synthesis technique of the characterization of the uncertainty. Please see chapter 5 for more information.

Research Question 3: How to enable the practical specification of adaptation policies for individual autonomic subsystems?

The specification of the adaptation policies for each subsystem is facilitated by using predicate statements to describe subsets of the state space that define the conditions under which an adaptation will occur and maps to another subset that specifies the state of the subsystem after the adaptation, see chapter 5 for more information. As long as the number of subsets is reasonable, see chapter 11, the specification of adaptation policies for each of the autonomic subsystems is practical.

Research Question 4: How to synthesize a plan of changes to the configurations of the autonomic subsystems that improves the performance of the collection of autonomic systems?

A plan of changes to the configuration of the autonomic subsystems to improve the performance of the collection is done through the selection of a meta-analysis and meta-strategy synthesis technique that is appropriate for the context by providing timely analysis at the level of assurance required, see chapter 6 for more information.

Research Question 5: How to synthesize a plan of changes that balances competing organizational priorities?

The configuration of the meta-manager as represented in a SEAM specification, see chapter 5, includes the specification for a global utility function that can be guarded by predicates. This allows for a meta-analysis and meta-strategy synthesis technique to score the potential states that could occur as a result of changing the configuration of the autonomic subsystems. The definition of this global utility function can weight various properties of the environment (e.g., performance and cost) to represent the competing organizational priorities.

Research Question 6: How to synthesize a plan of changes on a time scale appropriate to the context?

The administrator establishing the meta-manager can choose which meta-analysis and meta-strategy synthesis technique is most appropriate for their collection of autonomic subsystems, see chapter 6 for more information. While there are edge cases in which no technique will satisfy the requirements of the context, see chapter 11 for further discussion, it is expected that there is a large number of contexts in which at least one meta-analysis and meta-strategy technique is appropriate.

Research Question 7: How to leverage the knowledge about the structure of the system and environments to improve the effectiveness of managing a collection of autonomic systems?

The knowledge about the structure of the system and their environments is specified as *Global-Knowledge* in the SEAM specification which provides the primary source of configuration to the meta-manager, see chapter 5 for more information.

This chapter examines the claims of the thesis presented in chapter 1 and the research questions presented in chapter 2. These claims were validated through argumentation based on the examination of the evidence provided by the three case studies. However, there are several key assumptions that underpin the automated approach to meta-management presented in this thesis. The next chapter will discuss each of these assumptions, the limitations of the approach, and potential future work to address these limitations and further develop the approach.

Chapter 11

Discussion & Future Work

This chapter will reexamining the key assumptions in section 11.1, and elaborating on areas of future work in section 11.2.

11.1 Assumptions

This section examines the key assumptions made throughout the thesis to address how they impact the approach and its applicability to collections of autonomic systems.

Ability to Specify Adaptation Policy

The automated approach to meta-management presented in chapter 4 supported by the domain specific language, SEAM, presented in chapter 5 and utilized throughout the case studies in chapters 7, 8, and 9 provides for the human feasible specification of the adaptation policies of the individual subsystems in a substantial set of collections of autonomic systems. However, there are autonomic subsystems for which the human feasible specification of the adaptation policies is not practical.

Specifically, it is possible to have a system that is so sensitive to changes in the environment or the managed system that the effort of specifying its adaptive behavior is intractable for humans. For example, referring back to the scenario presented in chapter 2, if the shopping cart system had the autonomic behavior of adding additional granular virtual CPU (vCPU) capacity at each 1/10th of a second of page response time between 2 and 4 seconds, adding an additional 5% each time, that would result in a human administrator needing to specify a minimum of 20 adaptation policies for each relevant variation of the configuration settings. This could potentially result in the specification of hundreds of autonomic behaviors.

However, a single subsystem that has adaptive behaviors too complex to specify or analyze represents one end of the spectrum of the scalability of human specification of adaptation policies. The other end of the spectrum is reasonably represented in the Google Control Plane case study in which the autonomic systems under management are practically identical. This means the meta-manager could potentially handle a large number of subsystems because the diversity of the autonomic behaviors amongst the collection of autonomic systems is low. In the Google

Control Plane case study it was possible to make the assumption that the meta-strategy synthesized for one system would be effective on all. However, there are likely to be collections of autonomic systems with a low degree of complexity in their adaptive behaviors in which that assumption does not apply. Then the question becomes; what is the upward limit on the number of autonomic subsystems that can be specified by a human?

Future Work This thesis focuses on the scalability of the meta-analysis and meta-strategy synthesis techniques by examining the complexity of the models and the time necessary to generate a meta-strategy. However, this thesis does not directly examine the scalability of a human's ability to specify the adaptation policies for the individual autonomic subsystems. Specifically, how the number of subsystems and the combined complexity of the adaptation actions influence the scalability of the human effort to generate the specifications of the adaptation policies.

Expected Level of Effort: **High (> 18 mo.)**

Another challenge to the assumption that the autonomic behavior of the subsystem can be specified can be found in the use of human-in-the-loop adaptation in the autonomic subsystem. A human-in-the-loop adaptive system is one where a human plays a part in the control loop [44, 75]. An example of such a system would be an autonomous car in which the human still needs to drive in certain situations like complex driving situations or emergencies [44]. Due to the inherent inconsistency in the actions of human based adaptation, it might not be practical to specify the adaptive behaviors of the subsystem or it might not be possible to effectively reason about the autonomic behavior of the subsystem limiting the ability of a meta-manager to synthesize a meta-strategy.

Adherence to Specified Adaptive Behavior

An assumption of this thesis and the approach to meta-management is that each of the subsystems adhere to the specification of their adaptive behavior. If the autonomic subsystems do not adhere to their specified behavior then the level of assurance that can be provided by the meta-manager on the performance on the collection of adaptive systems is, at best, uncertain. A similar problem is adapting the meta-management approach to a collection of autonomic systems in which one or more subsystems might be actively working against the global objectives. This could occur in open networks of adaptive systems (e.g., corporate supply chain systems) in which members of the collection of autonomic system can join or leave as they wish, anyone of which could potentially be a rogue actor. Additionally, the autonomic subsystem may not have to be actively working against the collection of autonomic systems, but due to a fault or other potential conditions is no longer allowing the meta-manager to make adjustments to the configuration of the autonomic manager to improve the collections performance against global objectives.

These systems were intentionally placed out-of-scope for this thesis as there is another significant research effort in determining that an autonomic subsystem is not adhering to the specification of their adaptive behavior. Referring to the exemplar scenario presented in chapter 2, the shopping cart system adapting to add a server to its capacity and having that adaptation fail is part of the specified behavior. However, the probability of that failure might not be correctly

specified. Making that determination is non-trivial because a separate type of statistical analysis, referred to as a run test [12], would be required to determine if the behavior is random or if there is an underlying process to the results and if so what probability distribution or parameters of a given probability distribution best represents the data. Additionally, these tests are dependent upon having a sufficient number of samples of a potentially rare occurrence. It might also be possible to use various machine learning techniques, like reinforcement learning, to determine if the behavior is within a normal range or potentially anomalous. Finally, these efforts become potentially more complicated when the system is intentionally trying to avoid detection. For example, if an autonomic subsystem has been compromised, its autonomic behavior might change to generate some advantage for the responsible party, but the responsible party would not want the compromised system to be discovered.

Future Work It is possible for subsystems to become non-responsive to the meta-manager either due to a fault or other administrative action. In the most extreme case, it is also possible that an autonomic subsystem is actively working against the objectives of the collection of autonomic systems because it has been compromised or is a rogue actor in the collection of autonomic systems. A potential area of future work is to expand this approach to meta-management into zero-trust environments in which the individual autonomic subsystems might not implicitly trust each other or the meta-manager which would likely include the ability to identify and mitigate adaptive subsystems that are non-cooperative, adversarial, or compromised.

Expected Level of Effort: **High (> 18 mo.)**

Interface to Adjust Configuration Parameters

An assumption of this thesis is that the autonomic managers for each of the subsystems provides an interface to adjust their configuration parameters. For the purposes of this thesis, an interface is understood to mean a set of configuration parameters that can tune the autonomic behaviors of the subsystem to within a desired range. Referring to the exemplar system presented in chapter 2, the shopping cart system has the $MaximumCost$ and $CapacityBuffer$ configuration options. It is assumed that either the autonomic manager of the subsystem has an application programming interface (API) with which to manipulate the configuration options or that one could be built with reasonable engineering effort.

In the case studies presented in chapters 7, 8, and 9 different autonomic managers are in use, each of which has a set of configuration options available. As discussed in 10.1.3, this is used as evidence of the claim that the approach to meta-management presented in this thesis is applicable to a significant subset of collections of autonomic systems. However, there are autonomic systems in which the configuration options are not accessible. For example, managed cloud services in use in the AWS Shopping Cart Case Study, see chapter 7.

However, the autonomic managers controlling the managed subsystems that do not have an interface to adjust their configuration parameters represent a minority of the available autonomic managers. For example, many of the unmanaged cloud services like EC2 at AWS [102], virtual machines at Azure [109], and compute engine at Google Cloud [91] all have autonomic control interfaces. Additionally, a review of the SEAMS artifact repository [96] shows that the following

research artifacts also have autonomic managers with configuration options: RTX [95], mRubis [119], TRAPP [43], Self-Adaptive Video Encoder [77], UNDERSEA [42], DeltaIoT [58], and, while not part of the SEAMS artifact repository, Rainbow [24].

Given the number and variety of autonomic managers examined and the presence of available configuration options on the autonomic manager itself, it is reasonable to make the assumption that a significant majority of the autonomic subsystems that are part of a collection will have an interface available to adjust their configuration parameters.

Statistical Independence

One of the engineering decisions of the approach presented in this thesis is to make the assumption that the individual properties of the managed systems are statistically independent of each other. This is unlikely to be true in practical examination of a real-world collection of autonomic systems. For example, referring back to the exemplar scenario presented in chapter 2 and as depicted in listing 5.8, there is a natural dependency between the number of servers deployed, $ServerCount.$ and the average page response time, $AvgPageRespTime$; the more servers there are, the lower the average page response time down to a minimum.

The outcomes of adaptive actions that are intended to impact these properties would be more accurately represented by defining a conditional probability distribution between the variables. However, the representation of a conditional probability distributions for the purposes of specification in SEAM is a non-trivial problem as a single specification might call for elaborating multiple distributions depending on the relationship between the properties. Referring back to the example, it might be necessary to specify a different probability distribution for the $AvgPageRespTime$ for each value of the $ServerCount.$ Therefore, the choice was made to support only the specification of probability distributions on individual properties by assuming the statistical independence of the properties.

This approach intentionally compromises the accuracy of the specification for a practical engineering concern. However, the impact of this trade-off can be minimized by calculating the joint probability distribution of the assumed statistically independent properties. This is accomplished with the following calculation:

$$\Pr\left(\bigcap_{i=0}^{n} A_i\right) = \prod_{i=0}^{n} \Pr(A_i) \tag{11.1}$$

where the probability of a given *state* occurring can be determined by multiplying the probability of the values of all properties occurring.

This calculation is a straightforward process that can be performed by multiple toolsets and frameworks. Therefore, this thesis argues that the reduction in the accuracy of not being able to specify conditional probability distributions in favor of the practical engineering concerns of human feasible specification with the mitigation of being able to calculate the joint probability distributions is an acceptable trade-off.

Production Grade Analysis and Synthesis Tools

An implicit assumption of this thesis is that the analysis and synthesis tools used are of sufficient quality to be used in production grade operations. While PRISM [73] and PRISM-Games [22] are capable research tools, their suitability for production grade operations is uncertain as they are untested in such situations. Use of them in this thesis presented some operational challenges which required lower level management of memory and other unexpected and undocumented failure conditions that would not be practical in production grade environments. The production grade worthiness of the tools is important as it influences the scalability of the automated meta-management technique presented in this thesis.

However, not only can the production readiness of PRISM and PRISM-Games be readily improved, there are other potential libraries and frameworks which could serve as the basis for production grade analysis and synthesis. For example, [46, 47, 48] are all Python libraries that can assist with game theory based analysis, and [45] can perform analysis of Markov Decision Processes.

Continuous Behavior

Another assumption of this thesis is that the behavior of the autonomic subsystems is best described by discrete time models. As demonstrated in the case studies presented in chapters 7, 8, and 9 this approach is applicable to a significant subset of collections of autonomic systems. However, there are collections of autonomic systems which would be better described by continuous or dense time models. For example, the collection of autonomic systems that control the operations of an autonomous vehicle, aircraft, satellite, or complex robotics systems might all be better described using continuous time models. This limits the potential applicability of this approach to meta-management.

Future Work It is possible to expand upon this automated approach to meta-management to include autonomic subsystems that are better described by continuous time models. A potential area of future work is to enhance the approach, see chapter 4, the SEAM specification, see chapter 5, and the taxonomy, see chapter 6, to include the necessary items to support continuous time systems. For example, the use of continuous time models will necessitate a different set of meta-analysis and strategy synthesis techniques and an additional set of profiles for the timeliness, scalability, and assurance and several new objects, like the notion of a clock, will need to be added to SEAM.

Expected Level of Effort: **High (> 18 mo.)**

Hierarchical Control

Through the case studies presented in chapters 7, 8, and 9, this thesis has demonstrated the applicability of the automated approach to meta-management for a significant set of collections of autonomic systems which are subject to hierarchical control meaning the individual subsystems do not self-coordinate adaptive actions. However, there are collections of autonomic systems,

often referred to as multi-agent systems, in which the individual autonomic subsystems communicate with each other to coordinate their adaptive actions. Additional information about these systems is presented in section 3.2, but 'platoons' of self-driving cars [113] and fleets of drones [78] are examples of collections of autonomic system for which this approach might not be directly applicable.

Future Work A potential area of future work might be to evaluate the various coordination methods and protocols of multi-agent systems to determine what the role of a meta-manager might be to positively influence the performance of the collections of autonomic systems.

Expected Level of Effort: **High (> 18 mo.)**

11.2 Future Work

In addition to the areas of future work identified above, this section will outline the potential future work, both theoretical and engineering, applicable to the automation of the management of collections of autonomic systems.

Automated Assistance

There are several areas in this approach to meta-management that could benefit from additional tooling to provide automated assistance. Specifically, while this thesis demonstrated that writing a SEAM specification is human-feasible and the compiler does provide various checks and implementations of best practices for the individual synthesis techniques and tools, it does require the human administrator to have to consider and cross-check multiple items (e.g., coverage of the defined state space against the adaptation policy definitions) to ensure the consistency and completeness of the model. Therefore, tooling that could assist a human administrator with such tasks as ensuring the defined adaptation policies cover the complete defined state space would be beneficial to further practical adoption.

Additionally, it might be possible to automate the discovery of the adaptation policies of the individual autonomic subsystems by evaluating logs of their autonomic behavior and the environmental conditions that are likely to exist in practical industrial settings. Similarly, it might also be possible to automate the discovery of global knowledge, specifically the interrelationships between systems, through the examination of various application and system logs through the use of various machine learning techniques.

Expected Level of Effort: **Medium (6 - 18 mo.)**

Hybrid Planning & Latency Aware Adaptation

The automated approach to meta-management presented in this thesis builds upon prior work done in the area of providing autonomic capabilities to a single managed system, see 3.1. Therefore, there is other work in that area which might be applicable, with possible adaptations, to meta-management. For example, latency aware adaptation, see [82], in which the time an adaptation tactic is expected to take is factored into the analysis and planning phases of the MAPE-K

loop to allow for predictive and potentially concurrent adaptation actions. This could potentially be applied to allow a meta-manager to understand the effects of how long a particular meta-adaptation might take to show results and use that information to improve its effectiveness.

Additionally, hybrid planning [89], in which alternative analysis and planning processed are started. The first is intended to be fast but provides less assurance on the outcomes and the second provides a higher degree of assurance but will take longer to reach a result. This results in sub-optimal improvement in the short term and a higher degree of improvement in the long term. With appropriate modifications, this approach might be applicable and allow the meta-manager to better handle instabilities in the collections of autonomic systems by providing immediate short term changes that might not be ideal, but would improve results while a longer planning process attempts to determine the best course of action.

Expected Level of Effort: **Medium (6 - 18 mo.)**

Dynamic Adaptation Policies

This approach to meta-management allows for online dynamic generation of the meta-adaptation plans and the offline calculation, storage, and retrieval of them as appropriate for the context of collections of autonomic systems. However, as previously discussed, this approach does require the subsystem adaptation policies to be static for the purposes of meta-analysis and meta-strategy synthesis. A potential area of improvement and future work is for the adaptation policies from each subsystem to be dynamic and periodically communicated to the meta-manager. This could potentially enhance the applicability and effectiveness of the meta-manager by accounting for adaptive subsystems with frequently changing adaptation policies. Additionally, work on dynamic adaptation policies might also include extensions to account for uncertainties in the autonomic behavior itself, not just the outcomes. This would be relevant in human-in-the-loop situations in which the selection of the adaptation policy itself is likely to be non-deterministic.

Expected Level of Effort: **High (> 18 mo.)**

Homeostatic and Instability Configuration

The case studies in this thesis are geared towards either maintaining homeostatic operations of a collection of autonomic systems or handling a significant instability in the operations. However, as demonstrated in Google Control Plane Case Study presented in chapter 8, it is possible for a system configured for maintaining homeostatic operations to, at least partially, mitigate a significant instability. While the actions of a meta-manager in both situations are intended to improve the global utility, some instabilities in the operations of the collection of autonomic systems are known to the human administrators and have a preferred adaptation policy in such situations. Therefore, the effectiveness of a meta-manager in similar situations is unknown. It might be possible to expand upon the work presented in this thesis to include the ability of the meta-manager to identify significant instabilities that can be anticipated by administrators and use an alternative meta-manager configuration that has been specified to handle such situations. This would result in the meta-manager having the ability to handle both homeostatic operations and significant system instabilities in a deliberate manner. This would further expand the practicality and applicability of the approach to meta-management as there are systems that could

need meta-management for both situations and prevent the counter-intuitive need to potentially deploy two meta-managers.

Expected Level of Effort: **High (> 18 mo.)**

Chapter 12

Conclusion

This dissertation presents an automated approach to improve the performance of a collection of autonomic systems. This approach provides a formal basis for reasoning about changes to the configurations of the autonomic subsystems which tune the autonomic behavior to within a desired range to improve the collective performance of the collection of autonomic systems. The key idea of the approach is that the fact that each of the subsystems is autonomic provides three advantages to enable an automated approach to meta-management: (1) the simplification of the state space, (2) a reduction in the variance of the results of adaptation, and (3) the abstraction of the underlying managed system.

These advantages can be exploited to enable the creation of a meta-manager; a higher level autonomic control system that is specialized to the management of collections of autonomic systems. This is facilitated by (1) the creation of a domain specific language, SEAM, specialized to enable the specification of the adaptation policy for each subsystems, (2) the taxonomy of analysis and synthesis techniques that elaborate on each method's key properties to enable selection of the appropriate technique for the context in which the collection of autonomic systems is operating, (3) the definition of a MAPE-K control loop specialized for the purposes of managing a collection of autonomic systems, and (4) the creation of a method of specifying and using information that is unknown, or only partially known, to the individual autonomic subsystems, referred to as *global knowledge*.

This approach and framework are then instantiated as part of the Rainbow framework for self-adaptive systems and evaluated in three realistic case studies: (1) AWS Shopping Cart, (2) Google Control Plane, and (3) Power Grid Cascade Failure. In aggregate, the approach was demonstrated to be practical, effective, and applicable.

This thesis provides the following contributions to the theory of self-adaptive systems:

- A formal model characterizing an approach to meta-management of collection of autonomic systems;
- The definition of a Meta-MAPE-K Loop, an architecture pattern specialized to the needs of managing a collection of autonomic systems;
- A taxonomy of analysis and synthesis techniques that provides guidance on how their key properties align to the needs of a particular context.

This thesis provides the following contributions to the practice of self-adaptive systems:

- SEAM: A domain specific language specialized to the needs of meta-management including the specification of adaptation polices for each subsystem, global knowledge, and the global utility function;
- An implementation framework that can be used to instantiate a meta-manager;
- A demonstration of the effectiveness of a meta-manager in a variety of contexts and its ability to improve both homeostatic operations and significant system instabilities.

In addition to providing these contributions, this thesis sets the stage for future work in areas such as (1) managing subsystems that are adversarial or are in an error state, (2) controlling subsystems that self-coordinate adaptive actions (e.g., multi-agent systems), (3) including other features of self-adaptation including hybrid planning and latency aware adaptation, (3) handling continuous time systems (e.g., embedded systems), and (4) using machine learning and other artificial intelligence techniques to provide automated assistance in the creation of adaption policies and global knowledge.

Appendix A

SEAM Specification for AWS Shopping Cart Case Study

```
{
  "MetaManager": {
    "GlobalUtility": [
      {
        "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) > 0.66 &
            ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
            ↪ <= 1 & (#$.FrontEndUI.Database.CurrentLoad# /
            ↪ #$.FrontEndUI.Database.CurrentCapacity#) > 0.66 &
            ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) <= 1",
        "Formula":"0.25 * (1 - (#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) - 0.66) + 0.25 *
            ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
            ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 -
            ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#) -
            ↪ 0.66) + 0.25 * ((#$.Database.CurrentConfig.MaxCost# -
            ↪ #$.Database.CurrentState.CurrentCost#)/ #$.Database.CurrentConfig.MaxCost#)"
      },
    {
        "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) <= 0.66 &
            ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
            ↪ >= 0 & (#$.FrontEndUI.Database.CurrentLoad# /
            ↪ #$.FrontEndUI.Database.CurrentCapacity#) <= 0.66 &
            ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) >= 0",
        "Formula":"0.25 * (1 - (0.66 - #$.FrontEndUI.CurrentState.CurrentLoad# /
            ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#)) + 0.25 *
            ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
            ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 - (0.66 -
            ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#)) +
            ↪ 0.25 * ((#$.Database.CurrentConfig.MaxCost# - #$.Database.CurrentState.CurrentCost#)/
            ↪ #$.Database.CurrentConfig.MaxCost#)"
      },
    {
```

```
13            "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
                  ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) <= 0.66 &
                  ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                  ↪ >= 0 & (#$.FrontEndUI.Database.CurrentLoad# /
                  ↪ #$.FrontEndUI.Database.CurrentCapacity#) > 0.66 &
                  ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) <= 1",
14            "Formula":"0.25 * (1 - (0.66 - (#$.FrontEndUI.CurrentState.CurrentLoad# /
                  ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) - 0.66)) + 0.25 *
                  ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
                  ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 -
                  ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#) -
                  ↪ 0.66) + 0.25 * ((#$.Database.CurrentConfig.MaxCost# -
                  ↪ #$.Database.CurrentState.CurrentCost#)/ #$.Database.CurrentConfig.MaxCost#)"
15          },
16    {
17            "Predicate":"(#$.FrontEndUI.CurrentState.CurrentLoad# /
                  ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) > 0.66 &
                  ↪ (#$.FrontEndUI.CurrentState.CurrentLoad# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                  ↪ <= 1 & (#$.FrontEndUI.Database.CurrentLoad# /
                  ↪ #$.FrontEndUI.Database.CurrentCapacity#) <= 0.66 &
                  ↪ (#$.FrontEndUI.Database.CurrentLoad# / #$.FrontEndUI.Database.CurrentCapacity#) >= 0",
18            "Formula":"0.25 * (1 - (#$.FrontEndUI.CurrentState.CurrentLoad# /
                  ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) - 0.66) + 0.25 *
                  ↪ ((#$.FrontEndUI.CurrentConfig.MaxCost# - #$.FrontEndUI.CurrentState.CurrentCost#)/
                  ↪ #$.FrontEndUI.CurrentConfig.MaxCost#) + 0.25 * (1 - (0.66 -
                  ↪ (#$.Database.CurrentState.CurrentLoad# / #$.Database.CurrentState.CurrentCapacity#)) +
                  ↪ 0.25 * ((#$.Database.CurrentConfig.MaxCost# - #$.Database.CurrentState.CurrentCost#)/
                  ↪ #$.Database.CurrentConfig.MaxCost#)"
19          }
20        ],
21    "GlobalKnowledge": [
22      {
23        "Relation":
24        {
25          "Type":"Constraint",
26          "Predicate":"#$.Database.CurrentConfig.MaxCost# = 500 - #$.FrontEndUI.CurrentConfig.MaxCost#"
27        }
28      }
29    ],
30    "CurrentState": {
31      "HourOfDay": 10
32    },
33    "StateSpace": {
34      "Properties":
35      {
36        "NumOfHrs": {
37          "Type": "Numeric",
38          "Min": 1,
39          "Max": 24,
40          "Step": 1
41        },
42        "InitHour": {
```

```
        "Type": "Numeric",
        "Min": 1,
        "Max": 24,
        "Step": 1
       },
       "HourOfDay": {
        "Type": "Numeric",
        "Min": 1,
        "Max": 24,
        "Step": 1,
        "Intervals":12,
        "TimeCount":true
       }
    }
  },
  "AdaptationPolicies": [
   {
          "Behaviors": [
             {
               "StatePredicate":"",
               "ConfigUpdate":"#$.FrontEndUI.CurrentConfig.InstanceType#"
             },
         {
               "StatePredicate":"",
               "ConfigUpdate":"#$.FrontEndUI.CurrentConfig.MaxCost#"
             },
         {
               "StatePredicate":"",
               "ConfigUpdate":"#$.Database.CurrentConfig.MaxCost#"
             }
          ]
     }
  ]
   },
   "Environment": {
  "AdaptationPolicies": [
         {
        "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# <= 6",
          "Behaviors": [
             {
               "StatePredicate":"",
               "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
                 ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 3, 1, −0.5) &
                 ↪ #$.Database.CurrentState.CurrentLoad# = 0.8 ∗
                 ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 3, 1, −0.5)"
             }
          ]
   },
   {
      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 6 & #$.Global.CurrentState.HourOfDay# <= 12",
          "Behaviors": [
             {
```

```
92                  "StatePredicate":"",
93                  "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
                      ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 1, 3, 0.5) &
                      ↪ #$.Database.CurrentState.CurrentLoad# = 0.8 ∗
                      ↪ AGGD(#$.FrontEndUI.CurrentState.CurrentLoad#, 1, 3, 0.5)"
94                }
95              ]
96    },
97    {
98      "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 12 & #$.Global.CurrentState.HourOfDay# <=
                    ↪ 18",
99              "Behaviors": [
100               {
101                 "StatePredicate":"",
102                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
                      ↪ N(#$.FrontEndUI.CurrentState.CurrentLoad#, 0.2) &
                      ↪ #$.Database.CurrentState.CurrentLoad# = 150 + #$.FrontEndUI.CurrentState.CurrentLoad#"
103               }
104             ]
105   },
106   {
107     "ConfigPredicate":"#$.Global.CurrentState.HourOfDay# > 18 & #$.Global.CurrentState.HourOfDay# <=
                    ↪ 24",
108             "Behaviors": [
109               {
110                 "StatePredicate":"",
111                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentLoad# =
                      ↪ N(#$.FrontEndUI.CurrentState.CurrentLoad#, 0.2) &
                      ↪ #$.Database.CurrentState.CurrentLoad# = 150 + #$.FrontEndUI.CurrentState.CurrentLoad#"
112               }
113             ]
114   }
115 ]
116   },
117   "FrontEndUI": {
118     "CurrentState": {
119       "CurrentLoad": 66,
120       "CurrentCapacity": 100,
121       "CurrentCost": 100,
122       "AdaptDelay":0
123 },
124 "CurrentConfig": {
125   "MaxCost": 250,
126   "InstanceType": 0
127 },
128 "StateSpace": {
129   "Properties":
130   {
131     "CurrentLoad": {
132           "Type": "Numeric",
133           "Min": 0,
134           "Max": 500,
```

```
135             "Step": 10
136         },
137         "CurrentCapacity": {
138                 "Type": "Numeric",
139                 "Min": 0,
140                 "Max": 500,
141                 "Step": 10
142         },
143         "CurrentCost": {
144                 "Type": "Numeric",
145                 "Min": 0,
146                 "Max": #$.FrontEndUI.CurrentConfig.MaxCost#,
147                 "Step": 1
148         },
149         "AdaptDelay": {
150                 "Type": "Numeric",
151                 "Min": 0,
152                 "Max": 3,
153                 "Step": 1
154         }
155     },
156     "Configuration":
157     {
158         "MaxCost": {
159                 "Type": "Numeric",
160                 "Min": 0,
161                 "Max": 500,
162                 "Step": 1
163         },
164         "InstanceType": {
165                 "Type": "Numeric",
166                 "Min": 0,
167                 "Max": 1,
168                 "Step": 1
169          }
170       }
171   },
172   "AdaptationPolicies": [
173           {
174             "ConfigPredicate":"#$.FrontEndUI.Configuration.InstanceType# = 0",
175             "Behaviors": [
176                 {
177                   "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                        ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                        ↪ >= 0.75",
178                   "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                        ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# + 10 &
                        ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# + 10
                        ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
179                 },
180                 {
```

```
181                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                      ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                      ↪ <= 0.50",
182                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# − 10 &
                      ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# − 10
                      ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
183               },
184               {
185                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                      ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                      ↪ > 0.50 & (#$.FrontEndUI.CurrentState.CurrentCost# /
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) < 0.75",
186                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#"
187               },
188               {
189                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# > 0",
190                 "ResultState":"#$.FrontEndUI.CurrentState.AdaptDelay# =
                      ↪ #$.FrontEndUI.CurrentState.AdaptDelay# − 1"
191               }
192             ]
193   },
194   {
195     "ConfigPredicate":"#$.FrontEndUI.Configuration.InstanceType# = 1",
196           "Behaviors": [
197               {
198                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                      ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                      ↪ >= 0.75",
199                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                      ↪ [0.2|0,0.8|#$.FrontEndUI.CurrentState.CurrentCapacity# + 10] &
                      ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# + 1
                      ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
200               },
201               {
202                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                      ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                      ↪ <= 0.50",
203                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# − 10 &
                      ↪ #$.FrontEndUI.CurrentState.CurrentCost# = #$.FrontEndUI.CurrentState.CurrentCost# − 1
                      ↪ & #$.FrontEndUI.CurrentState.AdaptDelay# = 3"
204               },
205               {
206                 "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# = 0 &
                      ↪ (#$.FrontEndUI.CurrentState.CurrentCost# / #$.FrontEndUI.CurrentState.CurrentCapacity#)
                      ↪ > 0.50 & (#$.FrontEndUI.CurrentState.CurrentCost# /
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity#) < 0.75",
207                 "ResultState":"#$.FrontEndUI.CurrentState.CurrentCapacity# =
                      ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# & #$.FrontEndUI.CurrentState.AdaptDelay#
                      ↪ = 3"
```

```
            },
            {
              "StatePredicate":"#$.FrontEndUI.CurrentState.AdaptDelay# > 0",
              "ResultState":"#$.FrontEndUI.CurrentState.AdaptDelay# =
                ↪ #$.FrontEndUI.CurrentState.AdaptDelay# − 1"
            }
          ]
  }
 ]
  },
  "Database": {
    "CurrentState": {
      "CurrentLoad": 100,
      "CurrentCapacity": 100,
      "CurrentCost": 100
 },
    "CurrentConfig": {
      "MaxCost":250
 },
 "StateSpace": {
  "Properties":
  {
    "CurrentLoad": {
          "Type": "Numeric",
          "Min": 0,
          "Max": 500,
          "Step": 10
    },
    "CurrentCapacity": {
          "Type": "Numeric",
          "Min": 0,
          "Max": 500,
          "Step": 10
    },
    "CurrentCost": {
          "Type": "Numeric",
          "Min": 0,
          "Max": #$.Database.CurrentConfig.MaxCost#,
          "Step": 10
    }
  },
  "Configuration":
  {
    "MaxCost": {
          "Type": "Numeric",
          "Min": 0,
          "Max": 500,
          "Step": 1
    }
  }
 },
 "AdaptationPolicies": [
```

```
259         {
260           "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# != 12 &
                  ↪ #$.MetaManager.CurrentState.HourOfDay# != 1",
261           "Behaviors": [
262             {
263               "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                    ↪ #$.Database.CurrentState.CurrentCapacity#) >= 0.75 &
                    ↪ (#$.Database.CurrentState.CurrentCost# < #$.Database.Configuration.MaxCost#) + 10",
264               "ResultState":"#$.Database.CurrentState.CurrentCost# = #$.Database.CurrentState.CurrentCost# +
                    ↪ 10 & #$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.Database.CurrentState.CurrentCapacity# + 10"
265             },
266             {
267               "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                    ↪ #$.Database.CurrentState.CurrentCapacity#) <= 0.50 &
                    ↪ (#$.Database.CurrentState.CurrentCost# < #$.Database.Configuration.MaxCost#) + 10",
268               "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.Database.CurrentState.CurrentCapacity# − 10 & #$.Database.CurrentState.CurrentCost#
                    ↪ = #$.Database.CurrentState.CurrentCost# − 10"
269             },
270             {
271               "StatePredicate":"(#$.Database.CurrentState.CurrentCost# /
                    ↪ #$.Database.CurrentState.CurrentCapacity#) > 0.50 &
                    ↪ (#$.Database.CurrentState.CurrentCost# / #$.Database.CurrentState.CurrentCapacity#) <
                    ↪ 0.75",
272               "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.Database.CurrentState.CurrentCapacity#"
273             }
274           ]
275   },
276         {
277           "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# = 12",
278           "Behaviors": [
279             {
280               "StatePredicate":"",
281               "ResultState":"#$.Database.CurrentState.CurrentCapacity# = 200 &
                    ↪ #$.Database.CurrentState.CurrentCost# = 200"
282             }
283           ]
284    },
285         {
286     "ConfigPredicate":"#$.MetaManager.CurrentState.HourOfDay# = 1",
287           "Behaviors": [
288             {
289               "StatePredicate":"",
290               "ResultState":"#$.Database.CurrentState.CurrentCapacity# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCapacity# & #$.Database.CurrentState.CurrentCost# =
                    ↪ #$.FrontEndUI.CurrentState.CurrentCost#"
291             }
292           ]
293   }
294     ]
```

```
295     }
296 }
```

Listing A.1: Shopping Cart SEAM Specification

Appendix B

PRISM Specification for AWS Shopping Cart Case Study

```
1
2 dtmc
3
4 // ----- Rewards -----
5
6 rewards "GlobalUtility"
7  ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.66)  & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) <= 1)  & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) > 0.66) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 1): (0.25 * ( 1 - ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) -0.66))) +
                    ↪ (0.25 * ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * ( 1 - ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) -0.66))) + (0.25 ((MetaManager_Database_Config_MaxCost -
                    ↪ Database_CurrentCost) / MetaManager_Database_Config_MaxCost));
8
9  ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) >= 0) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) <= 0.66) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) >= 0): (0.25 * (1 - (0.66 - (FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity)))) + (0.25 * ((MetaManager_FrontEndUI_Config_MaxCost -
                    ↪ FrontEndUI_CurrentCost) / MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * (1 -
                    ↪ (0.66 - (Database_CurrentLoad / Database_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_Database_Config_MaxCost - Database_CurrentCost) /
                    ↪ MetaManager_Database_Config_MaxCost));
10
11  ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) >= 0) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) > 0.66) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 1): (0.25 * (1 - (0.66 - (FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity)))) +
                    ↪ (0.25 * ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * ( 1 - ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) -0.66))) + (0.25 * ((MetaManager_Database_Config_MaxCost -
                    ↪ Database_CurrentCost) / MetaManager_Database_Config_MaxCost));
12
```

```
13  ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.66) & ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) <= 1) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) <= 0.66) & ((Database_CurrentLoad /
                    ↪ Database_CurrentCapacity) >= 0): (0.25 * ( 1 - ((FrontEndUI_CurrentLoad /
                    ↪ FrontEndUI_CurrentCapacity) -0.66))) + (0.25 *
                    ↪ ((MetaManager_FrontEndUI_Config_MaxCost - FrontEndUI_CurrentCost) /
                    ↪ MetaManager_FrontEndUI_Config_MaxCost)) + (0.25 * (1 - (0.66 -
                    ↪ (Database_CurrentLoad / Database_CurrentCapacity)))) + (0.25 *
                    ↪ ((MetaManager_Database_Config_MaxCost - Database_CurrentCost) /
                    ↪ MetaManager_Database_Config_MaxCost));
14  endrewards
15
16  // ----- Model Control ----
17
18  global MODEL_Turn : [0..3] init 0;
19  global MODEL_Sink : bool init false;
20  global MODEL_TurnCount : [0..100] init 0;
21  const int MODEL_MaxTurns = MetaManager_NumOfHrs * MetaManager_NumOfIntervals;
22
23
24  // ----- Env. Module ----
25
26  formula ENVMNT_Formula_CurrentLoad_Lower_2 = (FrontEndUI_CurrentLoad - 2 < 10) ? (10) :
                    ↪ FrontEndUI_CurrentLoad - 2;
27  formula ENVMNT_Formula_CurrentLoad_Upper_2 = (FrontEndUI_CurrentLoad + 2 > 200) ? (200) :
                    ↪ FrontEndUI_CurrentLoad + 2;
28  formula ENVMNT_Formula_CurrentLoad_Lower_5 = (FrontEndUI_CurrentLoad - 5 < 10) ? (10) :
                    ↪ FrontEndUI_CurrentLoad - 5;
29  formula ENVMNT_Formula_CurrentLoad_Upper_5 = (FrontEndUI_CurrentLoad + 5 > 200) ? (200) :
                    ↪ FrontEndUI_CurrentLoad + 5;
30
31  const int ENVMNT_Turn = 0;
32
33  module ENVMNT
34
35
36    [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
                    ↪ (MetaManager_HourOfDay <= 6) ->
37      0.10 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' =
                    ↪ Database_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) & (MODEL_TurnCount' =
                    ↪ MODEL_TurnCount + 1)
38      + 0.55 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
                    ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) & (MODEL_Turn'
                    ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
39      + 0.08 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
                    ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) & (MODEL_Turn'
                    ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
40      + 0.25 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
                    ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) & (MODEL_Turn'
                    ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
41      + 0.02 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
                    ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) & (MODEL_Turn'
```

```
                ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
42
43
44  [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
                ↪ (MetaManager_HourOfDay > 6) & (MetaManager_HourOfDay <= 12) ->
45    0.10 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' =
                ↪ Database_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) & (MODEL_TurnCount' =
                ↪ MODEL_TurnCount + 1)
46    + 0.08 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
                ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) & (MODEL_Turn'
                ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
47    + 0.55 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
                ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) & (MODEL_Turn'
                ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
48    + 0.02 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
                ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) & (MODEL_Turn'
                ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
49    + 0.25 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
                ↪ (Database_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) & (MODEL_Turn'
                ↪ = FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
50
51
52  [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
                ↪ (MetaManager_HourOfDay > 12) & (MetaManager_HourOfDay <= 18) ->
53    0.70 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' = 150 +
                ↪ FrontEndUI_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) &
                ↪ (MODEL_TurnCount' = MODEL_TurnCount + 1)
54    + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
55    + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
56    + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
57    + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
58
59
60  [] (MODEL_TurnCount < MODEL_MaxTurns) & (MODEL_Turn = ENVMNT_Turn) &
                ↪ (MetaManager_HourOfDay > 18) & (MetaManager_HourOfDay <= 24) ->
61    0.70 : (FrontEndUI_CurrentLoad' = FrontEndUI_CurrentLoad) & (Database_CurrentLoad' = 150 +
                ↪ FrontEndUI_CurrentLoad) & (MODEL_Turn' = FrontEndUI_Turn) &
                ↪ (MODEL_TurnCount' = MODEL_TurnCount + 1)
62    + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_2) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
63    + 0.10 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_2) &
                ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
```

```
64    + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Upper_5) &
                    ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                    ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1)
65    + 0.05 : (FrontEndUI_CurrentLoad' = ENVMNT_Formula_CurrentLoad_Lower_5) &
                    ↪ (Database_CurrentLoad' = 150 + FrontEndUI_CurrentLoad) & (MODEL_Turn' =
                    ↪ FrontEndUI_Turn) & (MODEL_TurnCount' = MODEL_TurnCount + 1);
66
67  endmodule
68
69  //------ FrontEndUI ------
70
71  global FrontEndUI_CurrentLoad : [0..500] init 66;
72  global FrontEndUI_CurrentCapacity : [0..500] init 100;
73  global FrontEndUI_CurrentCost : [0..MetaManager_FrontEndUI_Config_MaxCost] init 100;
74  global FrontEndUI_Config_InstanceType : [0..1] init MetaManager_FrontEndUI_Config_InstanceType;
75  global FrontEndUI_AdaptDelay : [0..3] init 0;
76
77  formula FrontEndUI_Formula_CurrentCapacity_Lower_10 = (FrontEndUI_CurrentCapacity − 10 < 0) ? (0) :
                    ↪ FrontEndUI_CurrentCapacity − 10;
78  formula FrontEndUI_Formula_CurrentCapacity_Upper_10 = (FrontEndUI_CurrentCapacity + 10 > 200) ? (200) :
                    ↪ FrontEndUI_CurrentCapacity + 10;
79
80  formula FrontEndUI_Formula_CurrentCost_Lower_10 = (FrontEndUI_CurrentCost − 10 < 0) ? (0) :
                    ↪ FrontEndUI_CurrentCost − 10;
81  formula FrontEndUI_Formula_CurrentCost_Upper_10 = (FrontEndUI_CurrentCost + 10 >
                    ↪ MetaManager_FrontEndUI_Config_MaxCost) ?
                    ↪ (MetaManager_FrontEndUI_Config_MaxCost) : FrontEndUI_CurrentCost + 10;
82
83  formula FrontEndUI_Formula_CurrentCost_Lower_1 = (FrontEndUI_CurrentCost − 1 < 0) ? (0) :
                    ↪ FrontEndUI_CurrentCost − 1;
84  formula FrontEndUI_Formula_CurrentCost_Upper_1 = (FrontEndUI_CurrentCost + 1 >
                    ↪ MetaManager_FrontEndUI_Config_MaxCost) ?
                    ↪ (MetaManager_FrontEndUI_Config_MaxCost) : FrontEndUI_CurrentCost + 1;
85
86  const int FrontEndUI_Turn = 1;
87
88  module FrontEndUI
89
90    [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) >= 0.75) &
                    ↪ (FrontEndUI_Config_InstanceType = 0) −>
91      1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Upper_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Upper_10) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
92
93    [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.50) &
                    ↪ (FrontEndUI_Config_InstanceType = 0) −>
94      1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Lower_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Lower_10) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
95
```

```
96  [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.50) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) < 0.75) &
                    ↪ (FrontEndUI_Config_InstanceType = 0) ->
97    1: (FrontEndUI_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (MODEL_Turn' = Database_Turn);
98
99  [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay > 0) &
                    ↪ (FrontEndUI_Config_InstanceType = 0) ->
100   1: (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = FrontEndUI_AdaptDelay - 1);
101
102
103 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) >= 0.75) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
104   0.80: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Upper_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Upper_1) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3)
105   + 0.20: (FrontEndUI_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (FrontEndUI_CurrentCost' =
                    ↪ FrontEndUI_CurrentCost) & (MODEL_Turn' = Database_Turn) &
                    ↪ (FrontEndUI_AdaptDelay' = 3);
106
107 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) <= 0.50) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
108   1: (FrontEndUI_CurrentCapacity' = FrontEndUI_Formula_CurrentCapacity_Lower_10) &
                    ↪ (FrontEndUI_CurrentCost' = FrontEndUI_Formula_CurrentCost_Lower_1) &
                    ↪ (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = 3);
109
110 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay = 0) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) > 0.50) &
                    ↪ ((FrontEndUI_CurrentLoad / FrontEndUI_CurrentCapacity) < 0.75) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
111   1: (FrontEndUI_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (MODEL_Turn' = Database_Turn);
112
113 [] (!MODEL_Sink) & (MODEL_Turn = FrontEndUI_Turn) & (FrontEndUI_AdaptDelay > 0) &
                    ↪ (FrontEndUI_Config_InstanceType = 1) ->
114   1: (MODEL_Turn' = Database_Turn) & (FrontEndUI_AdaptDelay' = FrontEndUI_AdaptDelay - 1);
115
116 endmodule
117
118 //----- Database -----
119
120 global Database_CurrentLoad : [0..500] init 66;
121 global Database_CurrentCapacity : [0..500] init 100;
122 global Database_CurrentCost : [0..MetaManager_Database_Config_MaxCost] init 100;
123
124 formula Database_Formula_CurrentCapacity_Lower_10 = (Database_CurrentCapacity - 10 < 0) ? (0) :
                    ↪ Database_CurrentCapacity - 10;
125 formula Database_Formula_CurrentCapacity_Upper_10 = (Database_CurrentCapacity + 10 > 500) ? (500) :
                    ↪ Database_CurrentCapacity + 10;
126
```

```
127 formula Database_Formula_CurrentCost_Lower_10 = (Database_CurrentCost − 10 < 0) ? (0) :
                    ↪ Database_CurrentCost − 10;
128 formula Database_Formula_CurrentCost_Upper_10 = (Database_CurrentCost + 10 >
                    ↪ MetaManager_Database_Config_MaxCost) ? (MetaManager_Database_Config_MaxCost) :
                    ↪ Database_CurrentCost + 10;
129
130 const int Database_Turn = 2;
131
132 module Database
133   [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay != 12)  &
                    ↪ (MetaManager_HourOfDay != 1) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ >= 0.75) & (Database_CurrentCost < Database_Formula_CurrentCost_Upper_10)−>
134     1: (Database_CurrentCapacity' = Database_Formula_CurrentCapacity_Upper_10) & (Database_CurrentCost'
                    ↪ = Database_Formula_CurrentCost_Upper_10) & (MODEL_Turn' = MetaManager_Turn);
135
136
137   [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay != 12) &
                    ↪ (MetaManager_HourOfDay != 1) & ((Database_CurrentLoad / Database_CurrentCapacity)
                    ↪ <= 0.50) & (Database_CurrentCost < Database_Formula_CurrentCost_Upper_10) −>
138     1: (Database_CurrentCapacity' = Database_Formula_CurrentCapacity_Lower_10) & (Database_CurrentCost'
                    ↪ = Database_Formula_CurrentCost_Lower_10) & (MODEL_Turn' = MetaManager_Turn);
139
140   [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay != 12) &
                    ↪ (MetaManager_HourOfDay != 1) & ((Database_CurrentLoad / Database_CurrentCapacity) >
                    ↪ 0.50) & ((Database_CurrentLoad / Database_CurrentCapacity) < 0.75) −>
141     1: (MODEL_Turn' = MetaManager_Turn);
142
143   [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay = 12) −>
                    ↪ (Database_CurrentCapacity' = 200) & (Database_CurrentCost' = 200) & (MODEL_Turn' =
                    ↪ MetaManager_Turn);
144   [] (!MODEL_Sink) & (MODEL_Turn = Database_Turn) & (MetaManager_HourOfDay = 1) −>
                    ↪ (Database_CurrentCapacity' = FrontEndUI_CurrentCapacity) & (Database_CurrentCost' =
                    ↪ FrontEndUI_CurrentCost) & (MODEL_Turn' = MetaManager_Turn);
145
146 endmodule
147
148 // −−−−− MetaManager Module −−−−
149
150 global MetaManager_HourOfDay: [1..24] init MetaManager_InitHour;
151
152 const int MetaManager_NumOfHrs;
153 const int MetaManager_NumOfIntervals;
154 const int MetaManager_InitHour;
155 const int MetaManager_FrontEndUI_Config_InstanceType;
156 const int MetaManager_FrontEndUI_Config_MaxCost;
157 const int MetaManager_Database_Config_MaxCost = 500 − MetaManager_FrontEndUI_Config_MaxCost;
158
159 formula MetaManager_Formula_HourOfDay_Upper_1 = (MetaManager_HourOfDay + 1 > 24) ? (1) :
                    ↪ MetaManager_HourOfDay + 1;
160
161 const int MetaManager_Turn = 3;
162
```

```
163 module MetaManager
164
165  [] (!MODEL_Sink) & (MODEL_Turn = MetaManager_Turn) & (MODEL_TurnCount < MODEL_MaxTurns)
                    ↪ & (mod(MODEL_TurnCount, MetaManager_NumOfIntervals) = 0) ->
                    ↪ (MetaManager_HourOfDay' = MetaManager_Formula_HourOfDay_Upper_1) &
                    ↪ (MODEL_Turn' = ENVMNT_Turn);
166  [] (!MODEL_Sink) & (MODEL_Turn = MetaManager_Turn) & (MODEL_TurnCount < MODEL_MaxTurns)
                    ↪ & (mod(MODEL_TurnCount, MetaManager_NumOfIntervals) > 0) -> (MODEL_Turn' =
                    ↪ ENVMNT_Turn);
167  [] (!MODEL_Sink) & (MODEL_Turn = MetaManager_Turn) & (MODEL_TurnCount >= MODEL_MaxTurns)
                    ↪ -> 1:(MODEL_Sink' = true);
168
169 endmodule
```

Listing B.1: Shopping Cart PRISM DTMC Specification

Appendix C

SEAM Specification for Google Control Plane Case Study

```
{ //Root Node
  "MetaManager": {
    "GlobalUtility": [
      {
        "Predicate":"#$.MIG.CurrentState.OldestTimeMsg# <= 2500",
        "Formula":"(2500 – #$.MIG.CurrentState.OldestTimeMsg#) / 2500",
        "Objective":"Min"
      },
      {
        "Predicate":"#$.MIG.CurrentState.OldestTimeMsg# > 2500",
        "Formula":"0"
      }
    ],
    "AdaptationPolicies": [
  {
        "Behaviors": [
          {
            "StatePredicate":"",
            "ConfigUpdate":"#$.MIG.CurrentConfig.CoolDownDuration#"
          },
      {
            "StatePredicate":"",
            "ConfigUpdate":"#$.MIG.CurrentConfig.CanMaintenance#"
          }
        ]
   }
 ]
  },
  "Environment":
  {
    "CurrentState": {
      "QueueLoad": 250
    },
    "StateSpace": {
```

```
        "Properties":
        [
          "QueueLoad": {
            "Type":"Numeric",
            "Min":0,
            "Max":500,
            "Step":50
          }
        ]
      },
      "AdaptationPolicies": [
        {
          "ConfigPredicate":"",
          "isDefault":"True",
          "Behaviors": [
            {
              "StatePredicate":"",
              "ResultState":"#$.Environment.CurrentState.QueueLoad# =
                ↪ #$.Environment.CurrentState.QueueLoad# + 250"
            }
          ]
        }
      ]
    },
    "MIG": {
      "InstanceCount":2,
      "CurrentState": {
        "ServerCount":2,
        "OldestTimeMsg":100,
        "MaxOldestTimeMsg":150,
        "CoolDownTime":0
      },
      "CurrentConfig": {
        "CanMaintenance":"False",
        "CoolDownDuration": 1
      },
      "StateSpace": {
        "Properties":
        [
          "ServerCount": {
            "Type":"Numeric",
            "Min":1,
            "Max":20,
            "Step":1
          },
          "OldestTimeMsg": {
            "Type":"Numeric",
            "Min":0,
            "Max":3000,
            "Step":100
          },
          "MaxOldestTimeMsg": {
```

```
            "Type":"Numeric",
            "Min":0,
            "Max":3000,
            "Step":100
          },
          "CoolDownTime": {
            "Type":"Numeric",
            "Min":0,
            "Max":2,
            "Step":1
          }
        ],
        "Configuration":
        [
          "CanMaintenance": {
            "Type":"Numeric",
            "Min":0,
            "Max":1,
            "Step":1
          },
          "CoolDownDuration": {
            "Type":"Numeric",
            "Min":0,
            "Max":2,
            "Step":1
          }
        ]
      },
      "AdaptationPolicies": [
        {
          "ConfigPredicate":"",
          "isDefault":"True",
          "Behaviors": [
            { //Maintenance
              "StatePredicate":"#$.MIG.CurrentConfig.CanMaintenance# = 1",
              "ResultState":"#$.MIG.CurrentConfig.CanMaintenance# = 0 &
                ↪ #$.MIG.CurrentState.ServerCount# = 1"
            },
            { //AddCapacity
              "StatePredicate":"#$.MIG.CurrentState.OldestTimeMsg# >
                ↪ #$.MIG.CurrentState.MaxOldestTimeMsg# & #$.MIG.CurrentState.CoolDownTime# = 0",
              "ResultState":"#$.MIG.CurrentState.ServerCount# = #$.MIG.CurrentState.ServerCount# + 1 &
                ↪ #$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentConfig.CoolDownDuration#"
            },
            { //RemoveCapacity
              "StatePredicate":"#$.MIG.CurrentState.OldestTimeMsg# <=
                ↪ #$.MIG.CurrentState.MaxOldestTimeMsg# & #$.MIG.CurrentState.CoolDownTime# = 0",
              "ResultState":"#$.MIG.CurrentState.ServerCount# = #$.MIG.CurrentState.ServerCount# − 1 &
                ↪ #$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentConfig.CoolDownDuration#"
            },
            { //CoolDown
              "StatePredicate":"#$.MIG.CurrentState.CoolDownTime# > 0",
```

```
133                 "ResultState":"#$.MIG.CurrentState.CoolDownTime# = #$.MIG.CurrentState.CoolDownTime# −
                      ↪ 1"
134               },
135               { //Process Jobs
136                 "StatePredicate":"",
137                 "ResultState":"#$.MIG.Environment.QueueLoad# = #$.MIG.Environment.QueueLoad# − (150 ∗
                      ↪ #$.MIG.CurrentState.ServerCount#)"
138               }
139             ]
140           }
141       ]
142     }
143 }
```

Listing C.1: GCP Control Plane - SEAM Specification

Appendix D

PRISM Games Model for Google Control Plane Case Study

```
1
2  smg
3
4  // ----- Players -----
5
6  player ENV [EnvAction1], ENVMNT endplayer
7  player MIG1 [MIG1Action1], [MIG1Action2], [MIG1Action3], [MIG1Action4],[MIG1Action5] endplayer
8  player MIG2 [MIG2Action1], [MIG2Action2], [MIG2Action3], [MIG2Action4],[MIG2Action5] endplayer
9  player MM [MM] endplayer
10
11 // ----- Rewards -----
12
13 rewards "GlobalUtility"
14    [EnvAction1] MIG_OldestTimeMsg <= 2500: (2500 - MIG_OldestTimeMsg) / 2500;
15    [EnvAction1] MIG_OldestTimeMsg > 2500: 0;
16 endrewards
17
18 // ----- Global Variables -----
19
20 global Model_Sink : bool init false;
21 const int Model_Max_Turns = 150;
22
23
24 // ----- Control Module -----
25
26 module ControlModule
27
28    Model_Turn : [0..2] init 0;
29    Model_TurnCount : [0..1000] init 0;
30
31    [EnvAction1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
32    [MIG1Action1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
```

```
33    [MIG1Action2] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
34    [MIG1Action3] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
35    [MIG1Action4] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
36    [MIG1Action5] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
37    [MIG2Action1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
38    [MIG2Action2] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
39    [MIG2Action3] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
40    [MIG2Action4] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
41    [MIG2Action5] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
42    [MM] (!Model_Sink) -> (Model_Turn' = 0) & (Model_TurnCount' = Model_TurnCount + 1);
43
44  endmodule
45
46  // ----- Env. Module -----
47
48  global ENV_QueueLoad : [0..500] init 250;
49
50  module ENVMNT
51
52    [EnvAction1] (Model_Turn = 0) & (Model_TurnCount < Model_Max_Turns) -> (ENV_QueueLoad' =
                  ↪ ENV_QueueLoad + 250);
53    [](Model_Turn = 0) & (Model_TurnCount >= Model_Max_Turns) & (!Model_Sink) -> (Model_Sink' = true);
54
55  endmodule
56
57  // ----- MIG1-----
58
59  const int MIG1_Model_Turn = 1;
60
61  global MIG1_ServerCount : [1..20] init 2;
62  global MIG1_OldestTimeMsg : [0..3000] init 100;
63  global MIG1_MaxOldestTimeMsg : [0..3000] init 150;
64  global MIG1_CoolDownTime : [0..2] init 0;
65  global MIG1_CanMaintenance : [0..1] init 0;
66  global MIG1_CoolDownDuration : [0..2] init 1;
67
68  formula MIG1_Formula_QueueLoad1 = (ENV_QueueLoad - (MIG1_ServerCount * 150) < 0) ? (0) :
                  ↪ ((ENV_QueueLoad - (MIG1_ServerCount * 150) > 500) ? (500) : (ENV_QueueLoad -
                  ↪ (MIG1_ServerCount * 150)));
69  formula MIG1_Formula_ServerCount1 = (1 < 0) ? (0) : ((1 > 20) ? (20) : (1));
70  formula MIG1_Formula_ServerCount2 = (MIG1_Server_Count + 1 < 0) ? (0) : ((MIG1_Server_Count + 1) ? (20)
                  ↪ : (MIG1_Server_Count + 1));
```

```
71 formula MIG1_Formula_ServerCount3 = (MIG1_Server_Count − 1 < 0) ? (0) : ((MIG1_Server_Count − 1) ? (20)
       ↪ : (MIG1_Server_Count − 1));
72 formula MIG1_Formula_CoolDown1 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
73 formula MIG1_Formula_CoolDown2 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
74 formula MIG1_Formula_CoolDown3 = (MIG1_CoolDownTime − 1 < 0) ? (0) : ((MIG1_CoolDownTime − 1) ?
       ↪ (2) : (MIG1_CoolDownTime − 1));
75
76 module MIG
77
78    [MIG1Action1] (Model_Turn = MIG1_Model_Turn) −> (ENV_QueueLoad' = MIG1_Formula_QueueLoad1);
79
80    [MIG1Action2] (Model_Turn = MIG1_Model_Turn) & (MIG1_CanMaintenance = 1) −>
       ↪ (MIG1_Server_Count' = MIG1_Formula_ServerCount1) & (MIG1_CanMaintenance' = 0);
81
82    [MIG1Action3] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg >
       ↪ MIG1_MaxOldestTimeMsg) & (MIG1_Cool_Down_Count = 0) −> (MIG1_Server_Count' =
       ↪ MIG1_Formula_ServerCount2) & (MIG1_CoolDownTime' = MIG1_Formula_CoolDown1);
83
84    [MIG1Action4] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg <=
       ↪ MIG1_MaxOldestTimeMsg) & (MIG1_Cool_Down_Count = 0) −> (MIG1_Server_Count' =
       ↪ MIG1_Formula_ServerCount3) & (MIG1_CoolDownTime' = MIG1_Formula_CoolDown2);
85
86    [MIG1Action5] (Model_Turn = MIG1_Model_Turn) & (MIG1_CoolDownTime > 0) −>
       ↪ (MIG1_CoolDownTime' = MIG1_Formula_CoolDown3);
87
88 endmodule
89
90 // −−−−− MIG2−−−−−
91
92 const int MIG2_Model_Turn = 2;
93
94 global MIG2_ServerCount : [1..20] init 2;
95 global MIG2_OldestTimeMsg : [0..3000] init 100;
96 global MIG2_MaxOldestTimeMsg : [0..3000] init 150;
97 global MIG2_CoolDownTime : [0..2] init 0;
98 global MIG2_CanMaintenance : [0..1] init 0;
99 global MIG2_CoolDownDuration : [0..2] init 1;
100
101 formula MIG2_Formula_QueueLoad1 = (ENV_QueueLoad − (MIG2_ServerCount ∗ 150) < 0) ? (0) :
       ↪ ((ENV_QueueLoad − (MIG2_ServerCount ∗ 150) > 500) ? (500) : (ENV_QueueLoad −
       ↪ (MIG2_ServerCount ∗ 150)));
102 formula MIG2_Formula_ServerCount1 = (1 < 0) ? (0) : ((1 > 20) ? (20) : (1));
103 formula MIG2_Formula_ServerCount2 = (MIG2_Server_Count + 1 < 0) ? (0) : ((MIG2_Server_Count + 1) ? (20)
       ↪ : (MIG2_Server_Count + 1));
104 formula MIG2_Formula_ServerCount3 = (MIG2_Server_Count − 1 < 0) ? (0) : ((MIG2_Server_Count − 1) ? (20)
       ↪ : (MIG2_Server_Count − 1));
105 formula MIG2_Formula_CoolDown1 = (MIG2_CoolDownDuration < 0) ? (0) : ((MIG2_CoolDownDuration > 2)
       ↪ ? (2) : (MIG2_CoolDownDuration));
106 formula MIG2_Formula_CoolDown2 = (MIG2_CoolDownDuration < 0) ? (0) : ((MIG2_CoolDownDuration > 2)
       ↪ ? (2) : (MIG2_CoolDownDuration));
```

```
formula MIG2_Formula_CoolDown3 = (MIG2_CoolDownTime − 1 < 0) ? (0) : ((MIG2_CoolDownTime − 1) ?
    ↪ (2) : (MIG2_CoolDownTime − 1));

module MIG

    [MIG2Action1] (Model_Turn = MIG2_Model_Turn) −> (ENV_QueueLoad' = MIG2_Formula_QueueLoad1);

    [MIG2Action2] (Model_Turn = MIG2_Model_Turn) & (MIG2_CanMaintenance = 1) −>
    ↪ (MIG2_Server_Count' = MIG2_Formula_ServerCount1) & (MIG2_CanMaintenance' = 0);

    [MIG2Action3] (Model_Turn = MIG2_Model_Turn) & (MIG2_OldestTimeMsg >
    ↪ MIG2_MaxOldestTimeMsg) & (MIG2_Cool_Down_Count = 0) −> (MIG2_Server_Count' =
    ↪ MIG2_Formula_ServerCount2) & (MIG2_CoolDownTime' = MIG2_Formula_CoolDown1);

    [MIG2Action4] (Model_Turn = MIG2_Model_Turn) & (MIG2_OldestTimeMsg <=
    ↪ MIG2_MaxOldestTimeMsg) & (MIG2_Cool_Down_Count = 0) −> (MIG2_Server_Count' =
    ↪ MIG2_Formula_ServerCount3) & (MIG2_CoolDownTime' = MIG2_Formula_CoolDown2);

    [MIG2Action5] (Model_Turn = MIG2_Model_Turn) & (MIG2_CoolDownTime > 0) −>
    ↪ (MIG2_CoolDownTime' = MIG2_Formula_CoolDown3);

endmodule


// −−−−− Meta−Manager −−−−−

const int MM_Model_Turn = 3;

module MetaManager

    [MM](Model_Turn = MM_Model_Turn) −> (Model_Sink' = Model_Sink);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG1_CoolDownDuration' = 0);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG1_CoolDownDuration' = 1);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG1_CoolDownDuration' = 2);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG1_CanMaintenance' = false);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG1_CanMaintenance' = true);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG2_CoolDownDuration' = 0);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG2_CoolDownDuration' = 1);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG2_CoolDownDuration' = 2);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG2_CanMaintenance' = false);
    [MM](Model_Turn = MM_Model_Turn) −> (MIG2_CanMaintenance' = true);

endmodule
```

Listing D.1: GCP Control Plane - PRISM Specification - 2 Managed Instance Groups

```

smg

// −−−−− Players −−−−−

player ENV [EnvAction1], ENVMNT endplayer
player MIG1 [MIG1Action1], [MIG1Action2], [MIG1Action3], [MIG1Action4],[MIG1Action5] endplayer
```

```
8  player MM [MM] endplayer
9
10 // ----- Rewards -----
11
12 rewards "GlobalUtility"
13    [EnvAction1] MIG_OldestTimeMsg <= 2500: (2500 - MIG_OldestTimeMsg) / 2500;
14    [EnvAction1] MIG_OldestTimeMsg > 2500: 0;
15 endrewards
16
17 // ----- Global Variables -----
18
19 global Model_Sink : bool init false;
20 const int Model_Max_Turns = 150;
21
22
23 // ----- Control Module -----
24
25 module ControlModule
26
27    Model_Turn : [0..2] init 0;
28    Model_TurnCount : [0..1000] init 0;
29
30    [EnvAction1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
31    [MIG1Action1] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
32    [MIG1Action2] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
33    [MIG1Action3] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
34    [MIG1Action4] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
35    [MIG1Action5] (!Model_Sink) -> (Model_Turn' = Model_Turn + 1) & (Model_TurnCount' =
                  ↪ Model_TurnCount + 1);
36    [MM] (!Model_Sink) -> (Model_Turn' = 0) & (Model_TurnCount' = Model_TurnCount + 1);
37
38 endmodule
39
40 // ----- Env. Module -----
41
42 global ENV_QueueLoad : [0..500] init 250;
43
44 module ENVMNT
45
46    [EnvAction1] (Model_Turn = 0) & (Model_TurnCount < Model_Max_Turns) -> (ENV_QueueLoad' =
                  ↪ ENV_QueueLoad + 250);
47    [](Model_Turn = 0) & (Model_TurnCount >= Model_Max_Turns) & (!Model_Sink) -> (Model_Sink' = true);
48
49 endmodule
50
51 // ----- MIG1-----
52
```

```
53 const int MIG1_Model_Turn = 1;
54
55 global MIG1_ServerCount : [1..20] init 2;
56 global MIG1_OldestTimeMsg : [0..3000] init 100;
57 global MIG1_MaxOldestTimeMsg : [0..3000] init 150;
58 global MIG1_CoolDownTime : [0..2] init 0;
59 global MIG1_CanMaintenance : [0..1] init 0;
60 global MIG1_CoolDownDuration : [0..2] init 1;
61
62 formula MIG1_Formula_QueueLoad1 = (ENV_QueueLoad − (MIG1_ServerCount ∗ 150) < 0) ? (0) :
       ↪ ((ENV_QueueLoad − (MIG1_ServerCount ∗ 150) > 500) ? (500) : (ENV_QueueLoad −
       ↪ (MIG1_ServerCount ∗ 150)));
63 formula MIG1_Formula_ServerCount1 = (1 < 0) ? (0) : ((1 > 20) ? (20) : (1));
64 formula MIG1_Formula_ServerCount2 = (MIG1_Server_Count + 1 < 0) ? (0) : ((MIG1_Server_Count + 1) ? (20)
       ↪ : (MIG1_Server_Count + 1));
65 formula MIG1_Formula_ServerCount3 = (MIG1_Server_Count − 1 < 0) ? (0) : ((MIG1_Server_Count − 1) ? (20)
       ↪ : (MIG1_Server_Count − 1));
66 formula MIG1_Formula_CoolDown1 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
67 formula MIG1_Formula_CoolDown2 = (MIG1_CoolDownDuration < 0) ? (0) : ((MIG1_CoolDownDuration > 2)
       ↪ ? (2) : (MIG1_CoolDownDuration));
68 formula MIG1_Formula_CoolDown3 = (MIG1_CoolDownTime − 1 < 0) ? (0) : ((MIG1_CoolDownTime − 1) ?
       ↪ (2) : (MIG1_CoolDownTime − 1));
69
70 module MIG
71
72    [MIG1Action1] (Model_Turn = MIG1_Model_Turn) −> (ENV_QueueLoad' = MIG1_Formula_QueueLoad1);
73
74    [MIG1Action2] (Model_Turn = MIG1_Model_Turn) & (MIG1_CanMaintenance = 1) −>
       ↪ (MIG1_Server_Count' = MIG1_Formula_ServerCount1) & (MIG1_CanMaintenance' = 0);
75
76    [MIG1Action3] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg >
       ↪ MIG1_MaxOldestTimeMsg) & (MIG1_Cool_Down_Count = 0) −> (MIG1_Server_Count' =
       ↪ MIG1_Formula_ServerCount2) & (MIG1_CoolDownTime' = MIG1_Formula_CoolDown1);
77
78    [MIG1Action4] (Model_Turn = MIG1_Model_Turn) & (MIG1_OldestTimeMsg <=
       ↪ MIG1_MaxOldestTimeMsg) & (MIG1_Cool_Down_Count = 0) −> (MIG1_Server_Count' =
       ↪ MIG1_Formula_ServerCount3) & (MIG1_CoolDownTime' = MIG1_Formula_CoolDown2);
79
80    [MIG1Action5] (Model_Turn = MIG1_Model_Turn) & (MIG1_CoolDownTime > 0) −>
       ↪ (MIG1_CoolDownTime' = MIG1_Formula_CoolDown3);
81
82 endmodule
83
84
85 // −−−−− Meta−Manager −−−−−
86
87 const int MM_Model_Turn = 2;
88
89 module MetaManager
90
91    [MM](Model_Turn = MM_Model_Turn) −> (Model_Sink' = Model_Sink);
```

```
92     [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 0);
93     [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 1);
94     [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CoolDownDuration' = 2);
95     [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CanMaintenance' = false);
96     [MM](Model_Turn = MM_Model_Turn) -> (MIG1_CanMaintenance' = true);
97
98 endmodule
```

Listing D.2: GCP Control Plane - PRISM Specification - 1 Managed Instance Group

Appendix E

IEEE 39 Bus System Technical Information

UnitNo.	RatedPower	H	Ra	x'd	x'q	xd	xq	T'do	T'qo	xl
1	10000	5.000	0.000	0.600	0.800	2.000	1.900	7.000	0.700	0.300
2	1000	3.030	0.000	0.697	1.700	2.950	2.820	6.560	1.500	0.350
3	1000	3.580	0.000	0.531	0.876	2.495	2.370	5.700	1.500	0.304
4	1000	2.860	0.000	0.436	1.660	2.620	2.580	5.690	1.500	0.295
5	600	4.333	0.000	0.792	0.996	4.020	3.720	5.400	0.440	0.324
6	1000	3.480	0.000	0.500	0.814	2.540	2.410	7.300	0.400	0.224
7	1000	2.640	0.000	0.490	1.860	2.950	2.920	5.660	1.500	0.322
8	1000	2.430	0.000	0.570	0.911	2.900	2.800	6.700	0.410	0.280
9	1000	3.450	0.000	0.570	0.587	2.106	2.050	4.790	1.960	0.298
10	1000	4.200	0.000	0.310	0.080	1.000	0.690	10.200	0.000	0.125

Table E.1: Generator Parameter Values

Bus No.	Rs	Xls	Xd	Xq	Rfd	Rkd	Rkq	Xlfd	Xlkd	Xlkq
39	0.0002	0.3	2	1.9	0.00078	0.08842	0.07805	0.36429	0.30000	0.16552
31	0.0002	0.35	2.95	2.82	0.00121	0.08680	0.11626	0.40044	0.26421	0.15970
32	0.0002	0.304	2.495	2.37	0.00114	0.08339	0.09828	0.25324	0.40916	0.15710
33	0.0002	0.295	2.62	2.58	0.00115	0.22949	0.10351	0.15010	0.09017	0.05636
34	0.0002	0.324	4.02	3.72	0.00208	0.03440	0.17167	0.53585	1.07442	0.48710
35	0.0002	0.224	2.54	2.41	0.00096	0.06964	0.10510	0.31334	0.48576	0.19141
36	0.0002	0.322	2.95	2.92	0.00132	0.16917	0.11841	0.17947	0.14560	0.08041
37	0.0002	0.28	2.9	2.8	0.00117	0.07570	0.11947	0.32609	0.41083	0.18230
38	0.0002	0.298	2.106	2.05	0.00118	0.08605	0.08481	0.32017	0.34453	0.16644
30	0.0002	0.125	1	0.69	0.00029	0.09301	0.03207	0.23460	0.38542	0.16051

Table E.2: Generator Bus Values

		R1(pu)	X1(pu)	B1(pu)	km	R1(ohm/km)	X1(ohm/km)	B1(uS/km)	R0(ohm/km)	X0(ohm/km)	B0(uS/km)
1	2	0.0035	0.0411	0.6987	275.5	0.032	0.373	1.015	0.318	1.119	0.609
1	39	0.001	0.025	0.75	167.6	0.015	0.373	1.790	0.149	1.119	1.074
2	3	0.0013	0.0151	0.2572	101.2	0.032	0.373	1.017	0.321	1.119	0.610
2	25	0.007	0.0086	0.146	57.6	0.304	0.373	1.013	3.036	1.119	0.608
3	4	0.0013	0.0213	0.2214	142.8	0.023	0.373	0.620	0.228	1.119	0.372
3	18	0.0011	0.0133	0.2138	89.1	0.031	0.373	0.959	0.308	1.119	0.576
4	5	0.0008	0.0128	0.1342	85.8	0.023	0.373	0.626	0.233	1.119	0.375
4	14	0.0008	0.0129	0.1382	86.5	0.023	0.373	0.639	0.231	1.119	0.384
5	6	0.0002	0.0026	0.0434	17.4	0.029	0.373	0.996	0.287	1.119	0.598
5	8	0.0008	0.0112	0.1476	75.1	0.027	0.373	0.786	0.266	1.119	0.472
6	7	0.0006	0.0092	0.113	61.7	0.024	0.373	0.733	0.243	1.119	0.440
6	11	0.0007	0.0082	0.1389	55.0	0.032	0.373	1.011	0.318	1.119	0.607
7	8	0.0004	0.0046	0.078	30.8	0.032	0.373	1.012	0.324	1.119	0.607
8	9	0.0023	0.0363	0.3804	243.3	0.024	0.373	0.625	0.236	1.119	0.375
9	39	0.001	0.025	1.2	167.6	0.015	0.373	2.865	0.149	1.119	1.719
10	11	0.0004	0.0043	0.0729	28.8	0.035	0.373	1.012	0.347	1.119	0.607
10	13	0.0004	0.0043	0.0729	28.8	0.035	0.373	1.012	0.347	1.119	0.607
13	14	0.0009	0.0101	0.1723	67.7	0.033	0.373	1.018	0.332	1.119	0.611
14	15	0.0018	0.0217	0.366	145.4	0.031	0.373	1.007	0.309	1.119	0.604
15	16	0.0009	0.0094	0.171	63.0	0.036	0.373	1.086	0.357	1.119	0.651
16	17	0.0007	0.0089	0.1342	59.7	0.029	0.373	0.9	0.293	1.119	0.54
16	19	0.0016	0.0195	0.304	130.7	0.031	0.373	0.93	0.306	1.119	0.558
16	21	0.0008	0.0135	0.2548	90.5	0.022	0.373	1.126	0.221	1.119	0.676
16	24	0.0003	0.0059	0.068	39.5	0.019	0.373	0.688	0.19	1.119	0.413
17	18	0.0007	0.0082	0.1319	55	0.032	0.373	0.96	0.318	1.119	0.576
17	27	0.0013	0.0173	0.3216	116	0.028	0.373	1.109	0.28	1.119	0.666
21	22	0.0008	0.014	0.2565	93.8	0.021	0.373	1.093	0.213	1.119	0.656
22	23	0.0006	0.0096	0.1846	64.3	0.023	0.373	1.148	0.233	1.119	0.689
23	24	0.0022	0.035	0.361	234.6	0.023	0.373	0.616	0.234	1.119	0.369
25	26	0.0032	0.0323	0.513	216.5	0.037	0.373	0.948	0.37	1.119	0.569
26	27	0.0014	0.0147	0.2396	98.5	0.036	0.373	0.973	0.355	1.119	0.584
26	28	0.0043	0.0474	0.7802	317.7	0.034	0.373	0.982	0.338	1.119	0.589
26	29	0.0057	0.0625	1.029	418.9	0.034	0.373	0.983	0.34	1.119	0.59
28	29	0.0014	0.0151	0.249	101.2	0.035	0.373	0.984	0.346	1.119	0.59

Table E.3: Transmission Line Data

		Rated power (MVA)	Primary voltagekV	Secondary voltagekV	R	X
11	12	100	500	25	0.0016	0.0435
13	12	100	500	25	0.0016	0.0435
6	31	100	500	20	0	0.025
10	32	100	500	20	0	0.02
19	33	100	500	20	0.0007	0.0142
20	34	100	500	20	0.0009	0.018
22	35	100	500	20	0	0.0143
23	36	100	500	20	0.0005	0.0272
25	37	100	500	20	0.0006	0.0232
2	30	100	500	20	0	0.0181
29	38	100	500	20	0.0008	0.0156
19	20	100	500	500	0.0007	0.0138

Table E.4: Transformer Data

Bus No.	P (MW)	Q (MVar)
1	97.6	44.2
3	322	2.4
4	500	184
7	233.8	84
8	522	176
12	8.5	88
15	320	153
16	329	32.3
18	158	30
20	680	103
21	274	115
23	247.5	84.6
24	308.6	-92.2
25	224	47.2
26	139	17
27	281	75.5
28	206	27.6
29	283.5	26.9
31	9.2	4.6
39	1104	250

Table E.5: Load Data

Appendix F

Matlab Power Grid Simulation Model

```
1
2  function [time, gridA, gridB, gridC, gridD, gridE, iCon] = ConnectedGridModel(ic, baseDir, useMetaMgr,
                ↪ spotShed)
3
4    interConnect = [6, 5];
5
6    %1 to 2
7    interConnect(1,1) = 1;
8    interConnect(1,2) = 34;
9    interConnect(1,3) = 2;
10   interConnect(1,4) = 34;
11   interConnect(1,5) = 1;
12
13   %1 to 3
14   interConnect(2,1) = 1;
15   interConnect(2,2) = 33;
16   interConnect(2,3) = 3;
17   interConnect(2,4) = 33;
18   interConnect(2,5) = 1;
19
20   %1 to 5
21   interConnect(3,1) = 1;
22   interConnect(3,2) = 36;
23   interConnect(3,3) = 5;
24   interConnect(3,4) = 36;
25   interConnect(3,5) = 1;
26
27   %2 to 3
28   interConnect(4,1) = 2;
29   interConnect(4,2) = 32;
30   interConnect(4,3) = 3;
31   interConnect(4,4) = 32;
32   interConnect(4,5) = 1;
33
34   %2 to 4
35   interConnect(5,1) = 2;
```

```
36    interConnect(5,2) = 36;
37    interConnect(5,3) = 4;
38    interConnect(5,4) = 36;
39    interConnect(5,5) = 1;
40
41    %4 to 5
42    interConnect(6,1) = 4;
43    interConnect(6,2) = 32;
44    interConnect(6,3) = 5;
45    interConnect(6,4) = 32;
46    interConnect(6,5) = 1;
47
48
49    % Setting Initial Values
50    tic;
51
52    emergencyLoadShed = false;
53
54    grid1 = case39;
55    enableGrid1 = true;
56    collapsedGrid1 = false;
57
58    grid2 = case39;
59    enableGrid2 = false;
60    collapsedGrid2 = false;
61
62    grid3 = case39;
63    enableGrid3 = false;
64    collapsedGrid3 = false;
65
66    grid4 = case39;
67    enableGrid4 = false;
68    collapsedGrid4 = false;
69
70    grid5 = case39;
71    enableGrid5 = false;
72    collapsedGrid5 = false;
73
74    maxLoops = 25;
75    settings = get_default_settings();
76    settings.verbose = 0;
77    settings.max_recursion_depth = 1;
78
79    initial_contingency1 = ic;
80
81    while maxLoops > 0
82
83        %Meta-Manager
84        if useMetaMgr
85            if emergencyLoadShed
86                if spotShed
87                    grid1.bus(24, 11) = 0;
```

```
88                  grid2.bus(24, 11) = 0;
89                  grid3.bus(24, 11) = 0;
90                  grid4.bus(24, 11) = 0;
91                  grid5.bus(24, 11) = 0;
92
93                  grid1.bus(7, 3) = 0;
94                  grid2.bus(7, 3) = 0;
95                  grid3.bus(7, 3) = 0;
96                  grid4.bus(7, 3) = 0;
97                  grid5.bus(7, 3) = 0;
98              else
99                  grid1 = ShedLoad(grid1, 0.10);
100                 grid2 = ShedLoad(grid2, 0.10);
101                 grid3 = ShedLoad(grid3, 0.10);
102                 grid4 = ShedLoad(grid4, 0.10);
103                 grid5 = ShedLoad(grid5, 0.10);
104             end
105         end
106     end
107
108     if enableGrid1 && ~collapsedGrid1
109
110         grid1 = accfm(grid1, struct('branches', initial_contingency1), settings);
111
112         % 1 to 2 Branch 34, Gen 34
113         if grid1.branch_tripped(34) == 1 || grid1.gen(5,8) == 0
114             enableGrid2 = true;
115             initial_contingency2 = 34;
116             interConnect(1,5) = 0;
117         end
118
119         %1 to 3 - Branch 33, Gen 33
120         if grid1.branch_tripped(33) == 1 || grid1.gen(4,8) == 0
121             enableGrid3 = true;
122             initial_contingency3 = 33;
123             interConnect(2,5) = 0;
124         end
125
126         %1 to 5 - Branch 39, Gen 36
127         if grid1.branch_tripped(39) == 1 || grid1.gen(7,8) == 0
128             enableGrid5 = true;
129             initial_contingency5 = 39;
130             interConnect(3,5) = 0;
131         end
132
133         collapsedGrid1 = nnz(ismember(grid1.G.Nodes.Type, ['success']) == 1);
134         gridA = grid1;
135     end
136
137     if enableGrid2 && ~collapsedGrid2
138
139         grid2 = accfm(grid2, struct('branches', initial_contingency2), settings);
```

```
140
141         % 2 to 1 Branch 34, Gen 34
142         if grid2.branch_tripped(34) == 1 || grid2.gen(5,8) == 0
143             enableGrid1 = true;
144             initial_contingency1 = 34;
145             interConnect(1,5) = 0;
146         end
147
148         %2 to 3 – Branch 20, Gen 32
149         if grid2.branch_tripped(20) == 1 || grid2.gen(3,8) == 0
150             enableGrid3 = true;
151             initial_contingency3 = 20;
152             interConnect(4,5) = 0;
153         end
154
155         %2 to 4 – Branch 39, Gen 36
156         if grid2.branch_tripped(39) == 1 || grid2.gen(7,8) == 0
157             enableGrid4 = true;
158             initial_contingency4 = 39;
159             interConnect(5,5) = 0;
160         end
161
162         collapsedGrid2 = nnz(ismember(grid2.G.Nodes.Type, ['success']) == 1);
163         gridB = grid2;
164     end
165
166     if enableGrid3 && ~collapsedGrid3
167
168         grid3 = accfm(grid3, struct('branches', initial_contingency3), settings);
169
170         % 3 to 1 Branch 33, Gen 33
171         if grid3.branch_tripped(33) == 1 || grid3.gen(4,8) == 0
172             enableGrid1 = true;
173             initial_contingency1 = 33;
174             interConnect(2,5) = 0;
175         end
176
177         %3 to 2 – Branch 20, Gen 32
178         if grid3.branch_tripped(20) == 1 || grid3.gen(3,8) == 0
179             enableGrid2 = true;
180             initial_contingency2 = 20;
181             interConnect(4,5) = 0;
182         end
183
184         collapsedGrid3 = nnz(ismember(grid3.G.Nodes.Type, ['success']) == 1);
185         gridC = grid3;
186     end
187
188     if enableGrid4 && ~collapsedGrid4
189
190         grid4 = accfm(grid4, struct('branches', initial_contingency4), settings);
191
```

```
192         % 4 to 2 Branch 39, Gen 36
193         if grid4.branch_tripped(39) == 1 || grid4.gen(7,8) == 0
194             enableGrid2 = true;
195             initial_contingency2 = 39;
196             interConnect(5,5) = 0;
197         end
198 
199         %4 to 5 - Branch 20, Gen 32
200         if grid4.branch_tripped(20) == 1 || grid4.gen(3,8) == 0
201             enableGrid5 = true;
202             initial_contingency5 = 20;
203             interConnect(6,5) = 0;
204         end
205 
206         collapsedGrid4 = nnz(ismember(grid4.G.Nodes.Type, ['success']) == 1);
207         gridD = grid4;
208     end
209 
210     if enableGrid5 && ~collapsedGrid5
211 
212         grid5 = accfm(grid5, struct('branches', initial_contingency5), settings);
213 
214         % 5 to 4 Branch 20, Gen 32
215         if grid5.branch_tripped(20) == 1 || grid5.gen(3,8) == 0
216             enableGrid4 = true;
217             initial_contingency4 = 20;
218             interConnect(6,5) = 0;
219         end
220 
221         %5 to 1 - Branch 39, Gen 36
222         if grid5.branch_tripped(39) == 1 || grid5.gen(7,8) == 0
223             enableGrid1 = true;
224             initial_contingency1 = 39;
225             interConnect(3,5) = 0;
226         end
227 
228         collapsedGrid5 = nnz(ismember(grid5.G.Nodes.Type, ['success']) == 1);
229         gridE = grid5;
230     end
231 
232     if useMetaMgr
233         emergencyLoadShed = MetaManager(grid1, grid2, grid3, grid4, grid5);
234     end
235 
236     maxLoops = maxLoops - 1;
237 
238     if ~enableGrid1 && ~enableGrid2 && ~enableGrid3 && ~enableGrid4 && ~enableGrid5
239         maxLoops = 0;
240     end
241 
242     if collapsedGrid1 && collapsedGrid2 && collapsedGrid3 && collapsedGrid4 && collapsedGrid5
243         maxLoops = 0;
```

```
244         end
245
246         fprintf('\n *** Max Loop: %i% \n', maxLoops);
247     end
248
249     time = toc;
250     gridA = grid1;
251     gridB = grid2;
252     gridC = grid3;
253     gridD = grid4;
254     gridE = grid5;
255     iCon = interConnect;
256 end
257
258 function [gridA] = ShedLoad(grid, per)
259     busSize = size(grid.bus);
260
261     for i=1:busSize(1)
262         if grid.bus(i, 3) > 0
263             grid.bus(i, 3) = grid.bus(i, 3) * (1 - per);
264         end
265
266         if grid.bus(i, 4) > 0
267             grid.bus(i, 4) = grid.bus(i, 4) * (1 - per);
268         end
269     end
270
271     gridA = grid;
272 end
273
274 function [perTripped] = BranchesTripped(grid1)
275
276     branchSize = size(grid1.branch);
277
278     if(branchSize(2) > 13)
279         branchLoad = round(mean([sqrt(grid1.branch(:, 14).^2 + grid1.branch(:, 15).^2) sqrt(grid1.branch(:, 16).^2 +
                ↪ grid1.branch(:, 17).^2)], 2), 5);
280         branchCap = grid1.branch(:, 6);
281         branchCapRatio = branchLoad ./ branchCap;
282         branchesTripped = find(branchCapRatio >= 0.75);
283         perTripped = numel(branchCapRatio(branchesTripped));
284     else
285         perTripped = 0;
286     end
287 end
288
289 function [eLoadShed] = MetaManager(grid1, grid2, grid3, grid4, grid5)
290     eLoadShed = false;
291
292     g1Tripped = BranchesTripped(grid1);
293     if g1Tripped > 15
294         eLoadShed = true;
```

```
295     end
296
297     g2Tripped = BranchesTripped(grid2);
298     if g2Tripped > 15
299         eLoadShed = true;
300     end
301
302     g3Tripped = BranchesTripped(grid3);
303     if g3Tripped > 15
304         eLoadShed = true;
305     end
306
307     g4Tripped = BranchesTripped(grid4);
308     if g4Tripped > 15
309         eLoadShed = true;
310     end
311
312     g5Tripped = BranchesTripped(grid5);
313     if g5Tripped > 15
314         eLoadShed = true;
315     end
316 end
```

Listing F.1: Electrical Grid SEAM Specification

Appendix G

SEAM Specification for Electrical Grid Cascade

```
{
  "MetaManager": {
    "GlobalUtility": [
      {
        "Predicate":"",
        "Formula":"(5 − #$.Grid.CurrentState.CustomerOutageLevel#) / 5"
      }
    ],
    "GlobalKnowledge": [
      "Relation": {
        "Type":"Correlation",
        "Target":#$.Grid2.CurrentState.PerBranchesDown#,
        "Formula":"N(#$.Grid1.CurrentState.PerBranchesDown#, 0.5)",
        "Timedelay": 2
      },
      "Relation": {
        "Type":"Correlation",
        "Target":#$.Grid3.CurrentState.PerBranchesDown#,
        "Formula":"N(#$.Grid1.CurrentState.PerBranchesDown#, 0.5)",
        "Timedelay": 2
      },
      "Relation": {
        "Type":"Correlation",
        "Target":#$.Grid2.CurrentState.PerBranchesDown#,
        "Formula":"N(#$.Grid3.CurrentState.PerBranchesDown#, 0.5)",
        "Timedelay": 2
      },
      "Relation": {
        "Type":"Correlation",
        "Target":#$.Grid4.CurrentState.PerBranchesDown#,
        "Formula":"N(#$.Grid2.CurrentState.PerBranchesDown#, 0.5)",
        "Timedelay": 2
      },
      "Relation": {
```

```
            "Type":"Correlation",
            "Target":#$.Grid5.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid4.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          },
          "Relation": {
            "Type":"Correlation",
            "Target":#$.Grid1.CurrentState.PerBranchesDown#,
            "Formula":"N(#$.Grid5.CurrentState.PerBranchesDown#, 0.5)",
            "Timedelay": 2
          }
        ],
        "AdaptationPolicies": [
    {
            "Behaviors": [
              {
                "StatePredicate":"",
                "ConfigUpdate":"#$.Grid.CurrentConfig.EmergencyLoadShed#"
              }
            ]
      }
  ]
    },
    "Grid": {
      "InstanceCount":5,
      "CurrentState": {
        "PerBranchesDown": 10,
        "CustomerOutageLevel": 0
      },
      "CurrentConfig": {
        "EmergencyLoadShed": "0"
      },
      "StateSpace": {
        "Properties":
        {
          "PerBranchesDown": {
            "Type": "Numeric",
            "Min": 0,
            "Max": 30,
            "Step": 5
          },
          "CustomerOutageLevel": {
            "Type": "Numeric",
            "Min": 0,
            "Max": 5,
            "Step": 1
          }
        },
        "Configuration":
        {
          "EmergencyLoadShed": {
            "Type": "Numeric",
```

```
            "Min": 0,
            "Max": 1,
            "Step": 1
          }
        }
    },
    "AdaptationPolicies": [
      {
        "ConfigPredicate":"#$.Grid.CurrentConfig.EmergencyLoadShed# = 0",
        "isDefault":"True",
        "Behaviors": [
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 10",
            "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                ↪ N(#$.Grid.CurrentState.PerBranchesDown#, 0.5) &
                ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 0"
          },
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 20 &
                ↪ #$.Grid.CurrentState.PerBranchesDown# > 10",
            "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 0.5) &
                ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 3"
          },
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# > 20",
            "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, −1) &
                ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 5"
          }
        ]
      },
      {
        "ConfigPredicate":"#$.Grid.CurrentConfig.EmergencyLoadShed# = 1",
        "isDefault":"False",
        "Behaviors": [
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 10",
            "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 1"
          },
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# <= 20 &
                ↪ #$.Grid.CurrentState.PerBranchesDown# > 10",
            "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 2"
          },
          {
            "StatePredicate":"#$.Grid.CurrentState.PerBranchesDown# > 20",
```

```
126                 "ResultState":"#$.Grid.CurrentState.PerBranchesDown# =
                        ↪ AGGD(#$.Grid.CurrentState.PerBranchesDown#, 2, 2, 1) &
                        ↪ #$.Grid.CurrentState.CustomerOutageLevel# = 3"
127             }
128         ]
129     }
130   ]
131 }
```

Listing G.1: Electrical Grid SEAM Specification

Bibliography

[1] Ohm's Law. https://en.wikipedia.org/wiki/Ohm Accessed: 2023-08-28. 9.1

[2] James S. Albus. A Reference Model Architecture for Intelligent Systems Design. `http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=820486`. Accessed: 2019-10-18. 3.4

[3] Hanieh Alipour and Yan Liu. Model Driven Deployment of Auto-Scaling Services on Multiple Clouds. In *2018 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 93–96, 2018. doi: 10.1109/ICSA-C.2018.00033. 3.2

[4] Paolo Arcaini, Raffaela Mirandola, Elvinia Riccobene, and Patrizia Scandurra. A dsl for mape patterns representation in self-adapting systems. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture*, pages 3–19, Cham, 2018. Springer International Publishing. ISBN 978-3-030-00761-4. 3.2

[5] T. Athay, R. Podmore, and S. Virmani. A Practical Method for the Direct Analysis of Transient Stability. *IEEE Transactions on Power Apparatus and Systems*, PAS-98(2): 573–584, 1979. doi: 10.1109/TPAS.1979.319407. 1.2, 9.2

[6] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert Brayton. Verifying continuous time Markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, pages 269–276, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-68599-9. 7.2

[7] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate Symbolic Model Checking of Continuous-Time Markov Chains. In *Proceedings of the 10th International Conference on Concurrency Theory*, CONCUR '99, page 146–161, Berlin, Heidelberg, 1999. Springer-Verlag. ISBN 3540664254. 7.2

[8] Paolo Ballarini, Michael Fisher, and Michael Wooldridge. Uncertain Agent Verification through Probabilistic Model-Checking, 2006. 3.3

[9] Salvador Barberà, Peter Hammond, and Christian Seidl. *Handbook of Utility Theory. Volume 1 Principles*. 01 1998. ISBN 9780792381747. 4.1, 5, 6.1, 7.2

[10] Salvador Barberà, Peter Hammond, and Christian Seidl. *Handbook of Utility Theory. Volume 2 Extensions*. 01 2004. ISBN 9781441954176. 6.1

[11] Michael Bowling and Manuela Veloso. An Analysis of Stochastic Game Theory for Multiagent Reinforcement Learning. Technical Report CMU-CS-00-165, Carnegie Mellon University School of Computer Science, October 2000. URL `https://www.cs.cmu.`

edu/~mmv/papers/00TR-mike.pdf. 4

[12] J.V. Bradley. *Distribution-free Statistical Tests*. Prentice-Hall, 1968. ISBN 9780132162593. URL https://books.google.com/books?id=QKFqAAAAMAAJ. 11.1

[13] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL https://www.rfc-editor.org/info/rfc8259. 5

[14] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-02161-9. doi: 10.1007/978-3-642-02161-9_3. URL https://doi.org/10.1007/978-3-642-02161-9_3. 3.1

[15] Javier Cámara, David Garlan, Bradley Schmerl, and Ashutosh Pandey. Optimal Planning for Architecture-Based Self-Adaptation via Model Checking of Stochastic Games. In *Proceedings of the 10th DADS Track of the 30th ACM Symposium on Applied Computing*, Salamanca, Spain, 13-17 April 2015. 6.1, 6.4, 10.1.4

[16] Javier Cámara, Gabriel A. Moreno, David Garlan, and Bradley Schmerl. Analyzing Latency-Aware Self-Adaptation Using Stochastic Games and Simulations. *ACM Trans. Auton. Adapt. Syst.*, 10(4), jan 2016. ISSN 1556-4665. doi: 10.1145/2774222. URL https://doi.org/10.1145/2774222. 6.1, 6.4, 10.1.4

[17] Javier Cámara, David Garlan, Gabriel A. Moreno, and Bradley Schmerl. *Analyzing Self-Adaptation via Model Checking of Stochastic Games*. Number 9640. Springer, 2017. 6.4, 10.1.4

[18] Javier Camara, Wenxin Peng, David Garlan, and Bradley Schmerl. Reasoning about Sensing Uncertainty in Decision-Making for Self-Adaptation. In *Proceedings of the 15th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems (FOCLASA 2017)*, 2017. 3.3, 6.4

[19] Javier Cámara, Wenxin Peng, David Garlan, and Bradley Schmerl. Reasoning about Sensing Uncertainty and its Reduction in Decision-Making for Self-Adaptation. *Science of Computer Programming*, 167:51–69, 1 December 2018. 10.1.4

[20] T. Chen, M. Kwiatkowska, D. Parker, and A. Simaitis. Verifying Team Formation Protocols with Probabilistic Model Checking. In *Proc. 12th International Workshop on Computational Logic in Multi-Agent Systems (CLIMA XII 2011)*, volume 6814 of *LNCS*, pages 190–297. Springer, 2011. 3.3

[21] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. Automatic Verification of Competitive Stochastic Systems. *Formal Methods in System Design*, 43(1):61–92, 2013. 6.4

[22] Taolue Chen, Vojtěch Forejt, Marta Kwiatkowska, David Parker, and Aistis Simaitis. PRISM-games: A Model Checker for Stochastic Multi-Player Games. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 185–191, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN

978-3-642-36742-7. 8.2, 10.1.3, 11.1

[23] Betty H. C. Cheng, Kerstin I. Eder, Martin Gogolla, Lars Grunske, Marin Litoiu, Hausi A. Müller, Patrizio Pelliccione, Anna Perini, Nauman A. Qureshi, Bernhard Rumpe, Daniel Schneider, Frank Trollmann, and Norha M. Villegas. *Using Models at Runtime to Address Assurance for Self-Adaptive Systems*, pages 101–136. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08915-7. doi: 10.1007/978-3-319-08915-7_4. URL https://doi.org/10.1007/978-3-319-08915-7_4. 3.3

[24] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Institute for Software Research, School of Computer Science,Carnegie Mellon University, May 2008. URL http://reports-archive.adm.cs.cmu.edu/anon/isr2008/abstracts/08-113.html. Technical Report CMU-ISR-08-113. 3.1, 4, 5.6, 5.6, 10.1.4, 11.1

[25] Javier Cámara, Pedro Correia, Rogério de Lemos, David Garlan, Pedro Gomes, Bradley Schmerl, and Rafael Ventura. Incorporating architecture-based self-adaptation into an adaptive industrial software system. *Journal of Systems and Software*, 122:507–523, 2016. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2015.09.021. URL https://www.sciencedirect.com/science/article/pii/S0164121215002113. 1

[26] Samir El-Masri Darren Foster, Carolyn McGregor. A Survey of Agent-Based Intelligent Decision Support Systems to Support Clinical Management and Research. In M. Bernardo and J. Hillston, editors, *First International Workshop on Multi-Agent Systems for Medicine, Computational Biology, and Bioinformatics*, pages 16–34, 2005. 3.2

[27] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles*, 2007. URL https://www.amazon.science/publications/dynamo-amazons-highly-available-key-value-store. 7.1, 7.2

[28] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz. Market-Based Multirobot Coordination: A Survey and Analysis. *Proceedings of the IEEE*, 94(7):1257–1270, July 2006. ISSN 0018-9219. doi: 10.1109/JPROC.2006.876939. 3.2

[29] Roberto Rodrigues Filho, Elvin Alberts, Ilias Gerostathopoulos, Barry Porter, and Fábio M. Costa. Emergent web server: An exemplar to explore online learning in compositional self-adaptive systems. In *2022 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 36–42, 2022. doi: 10.1145/3524844.3528079. 3.2

[30] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Software Engineering Meets Control Theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing*

Systems, SEAMS '15, pages 71–82, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2821357.2821370. 3.4

[31] W. Findeisen, F.N. Bailey, M. Brdys, K. Malinowski, P. Tatjewski, and A. Wozniak. *Control and Coordination in Hierarchical Systems*. International Series on Applied Systems Analysis. John Wiley & Sons, 1980. URL http://pure.iiasa.ac.at/id/eprint/1227/. 3.4

[32] Luca Florio. Decentralized Self-Adaptation in Large-Scale Distributed Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 1022–1025, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. doi: 10.1145/2786805.2803192. URL https://doi.org/10.1145/2786805.2803192. 3.2

[33] Illinois Center for a Smarter Electric Grid. IEEE 39 Bus System. https://icseg.iti.illinois.edu/ieee-39-bus-system/. Accessed: 2023-9-18. 1.2, 9.2

[34] U.S.-Canada Power System Outage Task Force. Final Report on the August 14, 2003 Blackout in the United States and Canada: Causes and Recommendations. *https://www.energy.gov/oe/articles/blackout-2003-final-report-august-14-2003-blackout-united-states-and-canada-causes-and*, 2004. 9.1, 9.1, 9.1, 10.1.1, 10.1.3

[35] Vojtěch Forejt, Marta Kwiatkowska, Gethin Norman, and David Parker. *Automated Verification Techniques for Probabilistic Systems*, pages 53–113. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21455-4. doi: 10.1007/978-3-642-21455-4_3. URL https://doi.org/10.1007/978-3-642-21455-4_3. 6.2

[36] Wikipedia Foundation. List of probability distributions. https://en.wikipedia.org/wiki/List_of_probability_distributions. Accessed: 2023-06-25. 5.1

[37] Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002. 2

[38] Nicola Franco, Hoai My Van, Marc Dreiser, and Gereon Weiss. Towards a self-adaptive architecture for federated learning of industrial automation systems. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 210–216, 2021. doi: 10.1109/SEAMS51251.2021.00035. 3.2

[39] David Garlan, Bradley Schmerl, and Shang-Wen Cheng. *Software Architecture-Based Self-Adaptation*, pages 31–55. 04 2009. ISBN 978-0-387-89827-8. doi: 10.1007/978-0-387-89828-5_2. 1

[40] David Garlan, Nicolas D'Ippolito, and Kenji Tei. The 2nd Controlled Adaptation of Self-Adaptive Systems Workshop (CASaS2017). Technical Report NII-2017-10, National Institute of Informatics, 24-28 July 2017. 3.4

[41] P. Klazoglou K.Niwtaki Georgios Andreadis, K.-D. Bouzakis. Review of Agent-Based Systems in the Manufacturing Section, 03 2014. 3.2

[42] Simos Gerasimou, Radu Calinescu, Stepan Shevtsov, and Danny Weyns. UNDERSEA:

An Exemplar for Engineering Self-Adaptive Unmanned Underwater Vehicles (Artifact). *Dagstuhl Artifacts Series*, 3(1):3:1–3:2, 2017. ISSN 2509-8195. doi: 10.4230/DARTS.3.1.3. URL http://drops.dagstuhl.de/opus/volltexte/2017/7141. 11.1

[43] Ilias Gerostathopoulos and Evangelos Pournaras. TRAPPed in Traffic? A Self-Adaptive Framework for Decentralized Traffic Optimization. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 32–38, 2019. doi: 10.1109/SEAMS.2019.00014. 11.1

[44] Miriam Gil, Vicente Pelechano, Joan Fons, and Manoli Albert. Designing the Human in the Loop of Self-Adaptive Systems. In Carmelo R. García, Pino Caballero-Gil, Mike Burmester, and Alexis Quesada-Arencibia, editors, *Ubiquitous Computing and Ambient Intelligence*, pages 437–449, Cham, 2016. Springer International Publishing. ISBN 978-3-319-48746-5. 11.1

[45] Github. MDP Toolbox for Python. https://pymdptoolbox.readthedocs.io/en/latest/, . Accessed: 2023-10-13. 11.1

[46] Github. NashPy. https://nashpy.readthedocs.io/en/stable/discussion/other-python-game-theory-libraries.html, . Accessed: 2023-10-13. 11.1

[47] Github. PyNFG. https://pypi.org/project/PyNFG/0.1.2/, . Accessed: 2023-10-13. 11.1

[48] Github. Sagemath. https://doc.sagemath.org/html/en/reference/game_theory/index.html, . Accessed: 2023-10-13. 11.1

[49] S. Givant and P. Halmos. *Introduction to Boolean Algebras*. Undergraduate Texts in Mathematics. Springer New York, 2008. ISBN 9780387402932. URL https://books.google.com/books?id=ORILyf8sF2sC. 5.1

[50] Thomas J. Glazier, Bradley Schmerl, Javier Cámara, and David Garlan. Utility Theory for Self-Adaptive Systems. Technical Report CMU-ISR-17-119, Carnegie Mellon University Institute for Software Research, December 2017. URL http://acme.able.cs.cmu.edu/pubs/uploads/pdf/CMU-ISR-17-119.pdf. 4.1

[51] Google. Google Compute Engine. https://cloud.google.com/compute/, . Accessed: 2019-10-18. 8.1

[52] Google. Google Cloud Networking Incident #19009. https://status.cloud.google.com/incident/cloud-networking/19009, . Accessed: 2019-10-12. 1.2, 8.1, 8.1, 8.1, 10.1.1

[53] Google. Google Pub/Sub Documentation. https://cloud.google.com/pubsub/docs, . Accessed: 2019-10-18. 8.2

[54] NERC Steering Group. Technical Analysis of the August 14, 2003, Blackout: What Happened, Why, and What Did We Learn? *http://www.nerc.com/docs/docs/blackout/NERC_Final_Blackout_Report_07_13_04.pdf*, 2003. 1, 9.1, 9.1, 9.1, 10.1.1, 10.1.3

[55] S. Gössner and C. Bormann. JSONPath – XPath for JSON,

July 2020. URL https://www.ietf.org/archive/id/draft-goessner-dispatch-jsonpath-00.html. 5

[56] Hans Hansson and Bengt Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6, 02 1995. doi: 10.1007/BF01211866. 6.1, 6.2, 6.3

[57] IBM. An Architectural Blueprint for Autonomic Computing. https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf. Accessed: 2019-10-18. (document), 1, 3.1, 4.5, 4.1, 4.1, 4.1

[58] M. Usman Iftikhar, Gowri Sankar Ramachandran, Pablo Bollansee, Danny Weyns, and Danny Hughes. DeltaIoT: A Real World Exemplar for Self-Adaptive Internet of Things (Artifact). *Dagstuhl Artifacts Series*, 3(1):4:1–4:2, 2017. ISSN 2509-8195. doi: 10.4230/DARTS.3.1.4. URL http://drops.dagstuhl.de/opus/volltexte/2017/7142. 11.1

[59] Anne Immonen and Eila Niemelä. Survey of reliability and availability prediction methods from the viewpoint of software architectures. 7(65), Feburary 2008. ISSN 1619-1374. doi: 10.1007/s10270-006-0040-x. URL https://doi.org/10.1007/s10270-006-0040-x. 3.1

[60] The MathWorks Inc. MATLAB version: 9.14.0 (R2023a), 2023. URL https://www.mathworks.com. 1.2, 9.2, 9.2, 10.1.3

[61] David Garlan Javier Cámara, Wenxin Peng and Bradley Schmerl. "Reasoning about Sensing Uncertainty in Decision-Making for Self-Adaptation". In *Proceedings of the 15th International Workshop on Foundations of Coordination Languages and Self-Adaptive Systems (FOCLASA 2017)*, 2017. 3.3

[62] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, Cambridge, 2003. 5.1

[63] Nicholas R. Jennings. On Agent-based Software Engineering. *Artif. Intell.*, 117(2):277–296, March 2000. ISSN 0004-3702. doi: 10.1016/S0004-3702(99)00107-1. URL http://dx.doi.org/10.1016/S0004-3702(99)00107-1. 3.2

[64] Nicholas R. Jennings. An Agent-Based Approach for Building Complex Software Systems. *Communiations of the ACM Vol. 44, No. 4*, 2001. 3.2

[65] Capers Jones and Olivier Bonsignour. *The Economics of Software Quality*. Addison-Wesley Professional, 1st edition, 2011. ISBN 0132582201. 10.1.1

[66] Wenyun Ju. *Modeling, Simulation, and Analysis of Cascading Outages in Power Systems*. PhD thesis, University of Tennessee, Knoxville, 2018. URL https://trace.tennessee.edu/cgi/viewcontent.cgi?article=6862&context=utk_graddiss. 1.2, 9.2

[67] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer, vol 36, issue 1*, 2003. 3.1

[68] Jeffrey Kephart and D.M. Chess. The Vision Of Autonomic Computing. *Computer*, 36: 41 – 50, 02 2003. doi: 10.1109/MC.2003.1160055. 4.1

[69] Heiko Koziolek. Performance evaluation of component-based software systems: A sur-

vey. *Performance Evaluation*, 67(8):634–658, 2010. ISSN 0166-5316. doi: https://doi.org/10.1016/j.peva.2009.07.007. URL https://www.sciencedirect.com/science/article/pii/S016653160900100X. Special Issue on Software and Performance. 3.1

[70] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268, 2007. doi: 10.1109/FOSE.2007.19. 3.1

[71] Christian Krupitzer, Felix Maximilian Roth, Sebastian VanSyckel, Gregor Schiele, and Christian Becker. A survey on engineering approaches for self-adaptive systems. *Pervasive and Mobile Computing*, 17:184–206, 2015. ISSN 1574-1192. doi: https://doi.org/10.1016/j.pmcj.2014.09.009. URL https://www.sciencedirect.com/science/article/pii/S157411921400162X. 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian. 3.1, 4.1

[72] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic Model Checking. In M. Bernardo and J. Hillston, editors, *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, volume 4486 of *LNCS (Tutorial Volume)*, pages 220–270. Springer, 2007. 6.1

[73] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011. 7.2, 10.1.3, 11.1

[74] Marta Kwiatkowska, Gethin Norman, David Parker, and Gabriel Santos. Automatic Verification of Concurrent Stochastic Systems. *Formal Methods in System Design*, 58:188–250, 2021. 6.4

[75] Eric Lloyd, Shihong Huang, and Emmanuelle Tognoli. Improving Human-in-the-Loop Adaptive Systems Using Brain-Computer Interaction. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 163–174, 2017. doi: 10.1109/SEAMS.2017.1. 11.1

[76] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1):5–34, 2003. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(02)00378-8. URL https://www.sciencedirect.com/science/article/pii/S0004370202003788. Planning with Uncertainty and Incomplete Information. 6.3

[77] Martina Maggio, Alessandro Vittorio Papadopoulos, Antonio Filieri, and Henry Hoffmann. Self-Adaptive Video Encoder: Comparison of Multiple Adaptation Strategies Made Simple (Artifact). *Dagstuhl Artifacts Series*, 3(1):2:1–2:3, 2017. ISSN 2509-8195. doi: 10.4230/DARTS.3.1.2. URL http://drops.dagstuhl.de/opus/volltexte/2017/7140. 11.1

[78] Paulo Henrique Maia, Lucas Vieira, Matheus Chagas, Yijun Yu, Andrea Zisman, and Bashar Nuseibeh. Dragonfly: a Tool for Simulating Self-Adaptive Drone Behaviours.

In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 107–113, 2019. doi: 10.1109/SEAMS.2019.00022. 11.1

[79] Mohammad Hossein Manshaei, Quanyan Zhu, Tansu Alpcan, Tamer Bacşar, and Jean-Pierre Hubaux. Game Theory Meets Network Security and Privacy. *ACM Comput. Surv.*, 45(3), jul 2013. ISSN 0360-0300. doi: 10.1145/2480741.2480742. URL https://doi.org/10.1145/2480741.2480742. 4

[80] Gonçalo S. Martins, Hend Al Tair, Luís Santos, and Jorge Dias. POMDP: POMDP-based user-adaptive decision-making for social robots. *Pattern Recognition Letters*, 118:94–103, 2019. ISSN 0167-8655. doi: https://doi.org/10.1016/j.patrec.2018.03.011. URL https://www.sciencedirect.com/science/article/pii/S0167865518300825. Cooperative and Social Robots: Understanding Human Activities and Intentions. 6.3

[81] matpower.org. MatPower.org. https://matpower.org/. Accessed: 2023-9-18. 1.2, 9.2

[82] Gabriel A. Moreno. *Adaptation Timing in Self-Adaptive Systems*. PhD thesis, Institute for Software Research, School of Computer Science,Carnegie Mellon University, April 2017. URL http://reports-archive.adm.cs.cmu.edu/anon/isr2017/abstracts/17-103.html. Technical Report CMU-ISR-17-103. 4, 4.1, 7.1, 11.2

[83] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Flexible and Efficient Decision-Making for Proactive Latency-Aware Self-Adaptation. *ACM Transactions on Autonomous and Adaptive Systems*, 13(1), May 2018. URL https://doi.org/10.1145/3149180. 6.1, 10.1.4

[84] Saralees Nadarajah. A Generalized Normal Distribution. *Journal of Applied Statistics*, 32(7):685–694, 2005. doi: 10.1080/02664760500079464. URL https://doi.org/10.1080/02664760500079464. 5.1

[85] Asoke K. Nandi and Detlef Mämpel. An Extension of the Generalized Gaussian Distribution to include Asymmetry. *Journal of the Franklin Institute*, 332 (1):67–75, 1995. ISSN 0016-0032. doi: https://doi.org/10.1016/0016-0032(95)00029-W. URL https://www.sciencedirect.com/science/article/pii/001600329500029W. 5.1

[86] Matthias Noebels, Robin Preece, and Mathaios Panteli. AC Cascading Failure Model for Resilience Analysis in Power Networks. *IEEE Systems Journal*, 16(1):374–385, 2022. doi: 10.1109/JSYST.2020.3037400. 1.2, 9.2

[87] Preece R. Panteli M. Noebels, M. AC-CFM GitHub. https://github.com/mnoebels/AC-CFM. Accessed: 2023-9-18. 1.2, 9.2

[88] G. Norman, D. Parker, and X. Zou. Verification and Control of Partially Observable Probabilistic Systems. *Real-Time Systems*, 53(3):354–402, 2017. 6.3

[89] Ashutosh Pandey. *Hybrid Planning in Self-adaptive Systems*. PhD thesis, Institute for Software Research, School of Computer Science,Carnegie Mellon University, Feburary

2020. URL http://reports-archive.adm.cs.cmu.edu/anon/isr2020/abstracts/20-100.html. Technical Report CMU-ISR-20-100. 11.2

[90] Klaus Pohls Paul-Andrei Dragan, Andreas Metzger. Towards the Decentralized Coordination of Multiple Self-Adaptive Systems. *Dagstuhl Artifacts Series*, 2023. 3.2

[91] Google Cloud Platform. GoogleComputeEngine. https://cloud.google.com/compute. Accessed: 2023-10-19. 11.1

[92] Maxim Raya, Mohammad Hossein Manshaei, Mark Felegyhazi, and Jean-Pierre Hubaux. Revocation Games in Ephemeral Networks. 10 2008. doi: 10.1145/1455770.1455797. 4

[93] Orna Raz and Mary Shaw. An Approach to Preserving Sufficient Correctness in Open Resource Coalitions. In *Proceedings of the 10th International Workshop on Software Specification and Design*, IWSSD '00, page 159, USA, 2000. IEEE Computer Society. ISBN 0769508847. 1.2

[94] Mazeiar Salehie and Ladan Tahvildari. Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2), may 2009. ISSN 1556-4665. doi: 10.1145/1516533.1516538. URL https://doi.org/10.1145/1516533.1516538. 3.1, 4.1

[95] Sanny Schmid, Ilias Gerostathopoulos, Christian Prehofer, and Tomas Bures. Model Problem (CrowdNav) and Framework (RTX) for Self-Adaptation Based on Big Data Analytics (Artifact). *Dagstuhl Artifacts Series*, 3(1):5:1–5:3, 2017. ISSN 2509-8195. doi: 10.4230/DARTS.3.1.5. URL http://drops.dagstuhl.de/opus/volltexte/2017/7143. 11.1

[96] SEAMS. Artifact Repository. https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/. Accessed: 2023-10-19. 11.1

[97] AWS Web Services. Amazon API Gateway. https://aws.amazon.com/api-gateway/, . Accessed: 2023-9-18. 7.2

[98] AWS Web Services. AWS CloudWatch. https://aws.amazon.com/cloudwatch/, . Accessed: 2023-9-18. 7.2

[99] AWS Web Services. Amazon Cognito. https://aws.amazon.com/cognito/, . Accessed: 2023-9-18. 7.2

[100] AWS Web Services. DynamoDB Accelerator (DAX). https://aws.amazon.com/dynamodb/dax/, . Accessed: 2023-9-18. 1.2, 7.1, 7.2

[101] AWS Web Services. Amazon Dynamo. https://aws.amazon.com/dynamodb/, . Accessed: 2023-9-18. 7.2

[102] AWS Web Services. Amazon Elastic Compute Cloud (EC2). https://aws.amazon.com/ec2/, . Accessed: 2023-9-18. 7.2, 11.1

[103] AWS Web Services. Amazon Elastic BeanStalk. https://aws.amazon.com/elasticbeanstalk/details/, . Accessed: 2023-9-18. 1.2, 7.1, 7.2

[104] AWS Web Services. EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/, . Accessed: 2023-9-18. 7.2

[105] AWS Web Services. Amazon Lambda. https://aws.amazon.com/lambda/, . Accessed: 2023-9-18. 7.2

[106] AWS Web Services. Amazon SQS Queue. https://aws.amazon.com/sqs/, . Accessed: 2023-9-18. 7.2

[107] AWS Web Services. AWS Samples. https://github.com/aws-samples, . Accessed: 2023-9-18. 7.2

[108] AWS Web Services. AWS Samples Shopping Cart. https://github.com/aws-samples/aws-serverless-shopping-cart, . Accessed: 2023-9-18. 1.2, 7.2

[109] Azure Cloud Services. Azure Virtual Machines. https://azure.microsoft.com/en-us/products/virtual-machines, . Accessed: 2023-10-19. 11.1

[110] Lloyd S. Shapley. Stochastic Games. *Proceedings of the National Academy of Sciences*, 39:1095 – 1100, 1953. URL https://api.semanticscholar.org/CorpusID:1989943. 4.1

[111] Mary Shaw. 22nd International Conference on Software Engineering (ICSE 2000). In *Proceedings of the 4th International Software Architecture Workshop (ISAW-4)*, pages 46–50, 2000. 1.2

[112] Jonathan Shieber. Google cloud is down, affecting numerous applications and services. https://techcrunch.com/2019/06/02/google-cloud-is-down-affecting-numerous-applications-and-services/. Accessed: 2019-10-17. 8.1, 8.1

[113] Yong-Jun Shin, Lingjun Liu, Sangwon Hyun, and Doo-Hwan Bae. Platooning LEGOs: An Open Physical Exemplar for Engineering Self-Adaptive Cyber-Physical Systems-of-Systems. In *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 231–237, 2021. doi: 10.1109/SEAMS51251.2021.00038. 11.1

[114] Vitaly Shmatikov. Probabilistic Analysis of an Anonymity System. *J. Comput. Secur.*, 12(3,4):355–377, May 2004. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1297352.1297359. 3.3

[115] A. Simaitis. *Automatic Verification of Competitive Stochastic Systems*. PhD thesis, Department of Computer Science, University of Oxford, 2014. 6.4

[116] Christian Stier, Anne Koziolek, Henning Groenda, and Ralf Reussner. Model-Based Energy Efficiency Analysis of Software Architectures. In Danny Weyns, Raffaela Mirandola, and Ivica Crnkovic, editors, *Software Architecture*, pages 221–238, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23727-5. 3.1

[117] Manfred Stoll. *Introduction to Real Analysis*. 11 2001. ISBN 978-0321046253. 5.1

[118] Technopedia. Private Cloud. https://www.techopedia.com/definition/13677/private-cloud. Accessed: 2019-10-18. 8.1

[119] Thomas Vogel. MRUBiS: An Exemplar for Model-Based Architectural Self-Healing and Self-Optimization. In *Proceedings of the 13th International Conference on Software En-*

gineering for Adaptive and Self-Managing Systems, SEAMS '18, page 101–107, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357159. doi: 10.1145/3194133.3194161. URL https://doi.org/10.1145/3194133.3194161. 11.1

[120] Danny Weyns and Tanvir Ahmad. Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review. In Khalil Drira, editor, *Software Architecture*, pages 249–265, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39031-9. 1

[121] Danny Weyns, Nelly Bencomo, Radu Calinescu, Javier Cámara, Carlo Ghezzi, Vincenzo Grassi, Larse Grunske, Paola Inverardi, Jean-Marc Jezequel, Sam Malek, Raffaela Mirandola, Marco Mori, and Giordano Tambrrellii. *Perpetual Assurances for Self-Adaptive Systems*. Number 9640. Springer, 2017. 3.3

[122] Wikipedia. Control Plane. https://en.wikipedia.org/wiki/Control_plane, . Accessed: 2019-10-18. 10.1.1

[123] Wikipedia. Regular Expression. https://en.wikipedia.org/wiki/Regular_expression, . Accessed: 2023-9-18. 5.4

[124] Tong Wu, Qignshan Li, and Lu Wang. A Validation Method of Self-Adaptive Strategy Based on POMDP. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 373–375, 2019. doi: 10.1109/ICSME.2019.00061. 6.3

[125] IEEE Xplore. Citations: A Practical Method for the Direct Analysis of Transient Stability. https://ieeexplore.ieee.org/document/4113518/citations?tabFilter=papers#citations, . Accessed: 2023-9-18. 1.2

[126] IEEE Xplore. Citations: MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education. https://ieeexplore.ieee.org/document/5491276/citations?tabFilter=papers#citations, . Accessed: 2023-9-18. 1.2

[127] Lofti A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965. URL http://www-bisc.cs.berkeley.edu/Zadeh-1965.pdf. 5.1

[128] Ray Daniel Zimmerman, Carlos Edmundo Murillo-Sánchez, and Robert John Thomas. MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education. *IEEE Transactions on Power Systems*, 26(1):12–19, 2011. doi: 10.1109/TPWRS.2010.2051168. 1.2, 9.2