USING ASSL AS A METHOD FOR INTENT EXPRESSION TO ENACT AUTONOMIC NETWORKING

Solmaz Jaberi

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

March 2023

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Solmaz Jaberi**

Entitled: **Using ASSL as a Method for Intent Expression to Enact Autonomic Networking**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Sandra Cespedes

_____ Examiner
Dr. Sandra Cespedes

_____ Examiner
Dr. Aiman Hanna

_____ Co-supervisor
Dr. J. William Atwood

_____ Co-supervisor
Dr. Joey Paquet

Approved by _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Using ASSL as a Method for Intent Expression to Enact Autonomic Networking

Solmaz Jaberi

The term "Intent" is used in network management to designate the specification of goals or outcomes, without specifying how to achieve them. Ideally, intent should be specified in a natural language (e.g., English), but it must then be transformed into a representation that can be interpreted by the network devices.

The term "Autonomic Network" is used to describe a network that assumes many management functions "on its own". Such networks are well-suited to being "intent-driven".

This thesis provides a comprehensive definition of Intent, in the form of a set of Intent Objectives.

A set of Intent examples (expressed in English) is then presented, chosen to reflect intents from three distinct network environments (Carrier networks, Data Center networks, and Enterprise networks), and all of the Intent Objectives. Transformations of the examples into the Autonomic System Specification Language (ASSL) are given. ASSL was designed for the specification and verification of autonomic systems.

We show that, in spite of being designed for autonomic systems, ASSL is capable of expressing network intents. The expressiveness of ASSL is evaluated by demonstrating that it can cover almost all of the Intent Objectives, for the three network environments.

We conclude with discussion of the expressiveness of ASSL, with respect to each of the Intent Objectives, and the ways in which the current ASSL development environment could be improved.

Acknowledgments

I want to express my sincerest gratitude to my supervisors, Dr. Atwood and Dr. Paquet for their support and guidance throughout my master's program. Shaping this research became possible with their encouragement, expertise, and supervision. Not only my academic journey but my life has benefited from their guidance. I would like to express my deepest love and appreciation for my parents and brother, who always supported me and loved me unconditionally. They have been my constant source of inspiration in all aspects of my life. I want to dedicate this thesis to them with love and gratitude. Also, I want to thank my friends who made this adventure more memorable and I'm grateful for having them.

Contents

List of Figures

List of Tables

List of Acronyms

ACP	Autonomic Control Plane
AC	Autonomic Computing
AEIP	Autonomic Element Interaction Protocol
AE	Autonomic Element
ANIMA	Autonomic Networking Integrated Model and Approach
ANI	Autonomic Network Infrastructure
API	Application Programming Interface
ASA	Autonomic Service Agent
ASIP	Autonomic System Interaction Protocol
ASSL	Autonomic System Specification Language
AS	Autonomic System
BRSKI	Bootstrapping Remote Secure Key Infrastructure
CBOR	Concise Binary Object Representation
CLI	Command Line Interface
DC	Data Center
ECA	Event-Condition-Action
GRASP	GeneRic Autonomic Signaling Protocol
GUI	Graphical User Interface
IETF	Internet Engineering Task Force
IP	Interaction Protocol
IRTF	Internet Research Task Force

JVM	Java Virtual Machine
LSTM	Long Short-Term Memory
NEMO	NEtwork MOdeling
NMRG	Network Management Research Group
ONOS	Open Network Operating System
PBNM	Policy-Based Networking Management
QoS	Quality of Service
RFC	Request for Comments
SDN	Software-Defined Networking
SLA	Service Level Agreement
SLO	Service-level Objective
VM	Virtual Machine

Chapter 1

Introduction

In today's world, computer networks are essential in different aspects of our life, from residential internet to businesses. Managing networks more flexibly and productively is necessary to improve the quality of services and to reduce expenditures. The phrase *computer network management* describes some operations that govern a network and its resources. Computer networks should be adaptable to changes based on modifications to business goals. This is due to the dynamic nature of business requirements. Network administration is carried out by monitoring and controlling network components to ensure that it achieves its goals. The complexity of network management activities has been continuously increasing due to different factors, e.g.: (1) the development of new network technologies, (2) the development of new technologies that require telecommunication, (3) the increase in size of the networks, and (4) the increase in traffic flowing over the networks.

It is frequently apparent in many research studies that there needs to be a uniform understanding and definition of network goals due to their broad range. For example, for some technologies, speed is the priority in network communications, while in others, the quality of the service is the most critical goal. Another area for improvement regarding network management is the gap between stakeholders and network technicians. This gap exists because the stakeholders' perspective about network goals is much higher than network technicians' perspective. In order to reduce this gap, it is crucial to modernize network

management with tools to bridge this gap, to create a more coherent experience for all network users, and to clarify any ambiguity in goals to meet the users' expectations.

Early networks were straightforward due to the limited number of devices or services that needed to be maintained, making network management relatively simple. However, network management advanced along with the scale and complexity of networks. As a result, several strategies have been created to simplify network management. *Intent-based networking* is a recent innovation in network management that enables the network to automatically adapt to changing conditions and requirements by using declarative statements or expressions to indicate the desired behavior or state of the network. The high-level needs from a business view can be expressed using the concept of *intent*, which focuses on the intended result but not on the methods for achieving it. *Intent* as high-level network goals cover a flexible, vast context that can be used as a definitive objective to define network goals acceptable for both technical and non-technical network users. However, a method must be developed to express the purpose simply for non- technical users to understand. *Intent* can also be translated into the specific commands of unique devices. Thus, this study tries to define the concept of *intent* more adequately, based on its most straightforward meaning, which is the network goal. Network goals are composed of diverse objectives, from having more secure communications to maintaining high performance and increasing efficiency or any other desirable outcomes. Some studies worked on formalizing systems and the activities that must be taken, which theoretically can be mapped to the specific commands needed by various devices. In order to ascertain if they may be used to represent intent and for what types of *intent*, this thesis investigates such formal systems.

Notably, *autonomic systems* can support the idea of *intent* to facilitate its application to networks. *Autonomic systems* can manage and operate themselves with minimized or zero human intervention. These systems can adapt automatically to changing conditions and requirements since they are self-managing and self-regulating. Considering the human nervous system as an example, where it consists of a complex network of cells and pathways. The nervous system is responsible for diverse tasks to make the human body functional;

however, this system does not require a controller outside the body. Without an outside controller, the human body can cure itself while experiencing health issues or perform different responsibilities, such as breathing and digesting. This inspired autonomicity feature can be added to networks to facilitate their management, based on appropriately expressed intent. *Autonomic networks*, analogous to *autonomic systems*, can manage themselves in any situation to maintain the goals that are defined for them based on their requirements. Also, if an undesirable situation occurs, they can take appropriate action to solve the problem to normalize their situation. This leads network management to *intent-based networking* where a network is managed based on a unified declared goal through different technologies.

1.1 Objectives and Motivation

The combination of *intents* as the objectives to construct desirable outcomes for the network and *autonomic network* as the infrastructure to apply those objectives can modernize network management efficiently and productively. Attaining this goal requires to have the following parameters:

- A comprehensive definition of intent.
- A means to express intents using a high-level language that can be executed as an operational control system.

This research focuses on the following essential questions, which we found that need to be better covered in the literature: (1) What is the definition of *intent* objectives? (2) In what language should the intents expressed? (3) How can these intents relate to an autonomic system independently of the underlying networking infrastructure? Following an extensive literature study, we concluded that there is a shortcoming of an adequate, comprehensive definition for *intent* as the main idea of conducting the network's operation based on our observation of attempts toward goal-driven networking. To address this issue, our first challenge was to create such a definition based on different sources in the literature.

Ideally, the level of abstraction to define intents as the operational purpose of a network should be using a high-level language by focusing on *what* should be achieved rather than *how* to achieve those goals. However, it is an assiduous task due to the dynamic nature of business requirements and human language features to describe them. Thus, focusing on high-level languages, which are close to natural language and easy to understand by users with any level of networking knowledge, is more logical. Then, besides the expressiveness and understandability of higher-level languages, it should be able to cover a wide variety of intents.

1.2 Methodology

Following our in-depth analysis of the various definitions of intent-based networking, we shall first provide a detailed description of what constitutes intent-based networking and its required operational characteristics. In addition, we investigate specification languages and frameworks to find the most appropriate one that can express intents. This specification language should have a hierarchical basis. The hierarchical structure helps the autonomic network to define different levels of declaration to distinguish public and private behaviors of the system. Moreover, this framework should be able to achieve desired outcomes of the network by specifying autonomic behaviors under self-management regulations. Therefore, the intent specification shall combine with the characteristics of an autonomic system. This strategy can facilitate the management of the network by applying intents. After finding such a language/framework as the solution, we shall describe it as an autonomic network specification platform and then demonstrate conceptually and by examples how it can be used to express networking intents. Finally, we shall evaluate our solution by demonstrating that applying the solution framework to intent-based autonomic networking meets our definition, objectives, and characteristics of such systems.

1.3 Thesis Contributions

This study addresses deficiencies in intent-based network management approaches to make the following contributions.

- Provide an integrated definition of intent, and its objectives.
- Find a specification language as the solution to `express` intents under the context of autonomic networks through numerous and diversified concrete examples.
- Investigate advantages and deficiencies of the proposed solution as an intent expression language in autonomic networks.

1.4 Thesis Scope

Our main goal is to show the coverage and expressiveness of the possible solution to define intents and autonomic networks based on our defined intent objectives. Our focus is more on the *expression* of the intents rather than the operational details of their *deployment*.

1.5 Thesis Outline

After presenting an overview of our research in Chapter 1, Chapter 2 reviews the related work in the intent-based networking area to explore the shortages in this research area. Then we initiate our research solution by recommending our definition of intent objectives in Chapter 3, followed by a description of our suggested solution to the stated problem, along with experimental examples to validate it with different scenarios. In Chapter 4, we analyze the concrete examples, advantages, and shortcomings of the proposed intent specification language as the solution. Finally, in Chapter 5, we conclude the whole research and present possible future avenues of this research work.

1.6 Summary

There needs to be a more adequate and integrated intent definition that covers all aspects of network goals acceptable for different user types. Despite the progress in intent-based networking, the necessity of a unified intent expression specification language still needs to be resolved.

Chapter 2

Background

In this chapter, we introduce the concepts used in this research and pre-existing research solutions related to our study or upon which our proposed solution is built.

2.1 Autonomic and Intent-Based Networking

The notion of autonomic networking comprises concepts described in the following sections. It is essential to understand that different definitions and terminologies describe autonomic networking. We present only some of them that relate most to our point of view in this thesis.

2.1.1 Autonomic Computing and Autonomic Networking

Autonomic computing was first described in a white paper manifesto by IBM in 2001, later published as a paper [21]. Autonomic computing is conceptually based on the notion of the human autonomic nervous system in which any modification to the system's state is determined, analyzed, and managed by the system itself to achieve self-regulation, or homeostasis [3, 23, 42]. Regarding this purpose, an autonomic computing system typically includes the following properties [15, 21], which are often referred to as the *self-chop* or *self-** properties of autonomic computing:

- *Self-Awareness*: The system's capability to develop models that record information about itself and its surroundings and reason using these models to extrapolate new data that it can use to act to attain high-level goals, provided that the data, the model, and the goal can all change over time [8].

- *Self-Protection*: The system's ability to be aware of potential threats or security issues and how to handle them.

- *Self-Optimization*: The system's ability to improve its functionality and performance towards an optimum state.

- *Self-Healing*: The system's ability to restore itself in case of any problem that might be hampering its functionality or to predict and avoid any such potential problems.

- *Self-Configuration*: The system's ability to automatically and appropriately re-arrange itself after any modification or change happens that makes the current configuration inappropriate for the situation.

When applied to computer networking, these goals refer to self-management of a computer network aiming to minimize low-level human operator intervention. Each network has a purpose of achieving its business goals. The expression of autonomic behavior is to be defined at a high level of abstraction, possibly but not necessarily close to human language. It aims to express business goals that need to be more directly understood by the underlying networking infrastructure it seeks to control. Hence, there needs to be some language that describes the autonomic behavior, which is then translated into actions or configuration changes that are to be applied to the underlying networking infrastructure. However, designing such a language is difficult since the range of network business goals is vast and diverse. The formulation of low-level network policies, defined by predetermined static rules, is currently made possible by various methods. These kinds of solutions frequently concentrate on the static expression of a policy. However, they are unable to dynamically monitor more complicated policies that must be monitored due to their inherently dynamic nature or the fact that they must be enforced in a networking environment that is constantly changing.

Rather than relying on a formal language to allow the user to express the required autonomic behavior directly, some existing solutions for autonomic networking concentrate on a mechanism to extract the autonomic specifications from human language or other information. For example, some patterns can be found in the human language or additional information, which are then matched with specific predefined categories of behavior identified on the service provider side.

2.1.2 Policy-Based and Intent-Based Networking

Policy-Based Networking Management (PBNM) is a push-based approach for entering the network policies separating the instructions for ruling a system from its functionality [10]. Inherently, policies define conditions for an action to happen, and PBNM is a systematic injection of policies into a system, which aims to regulate the system by triggering certain actions when certain predefined conditions are met. Using PBNM, each rule contains a set of conditions or control loops that triggers a goal or action. The policy rules are provided by users or knowledge systems, and the control loops can be executed without outside intervention. Thus, policy-based approaches help the automation of management procedures.

On the other side, the concept of *intent* is a high-level abstract definition of business and operational goals to manage a network [45]. Inherently, the processing of intent implies a learning phase compared to the execution of policies, as they are defined with the highest level of abstraction. Policies are easier to be transformed into lower-level networking instructions since they are more precisely aimed. In fact, policies can be considered a subset of intents, a concept that aims to cover a more comprehensive range of network management goals.

In other words, policies are Event-Condition-Actions (ECA) entities, and intents are high-level, abstract, and declarative objectives unaware of hardware details [45]. Eventually, intents should be translated into low-level network actions and configurations applied to network devices. There are different approaches to this purpose [27]:

- The **machine learning approach** such as [17] to intent-based networking focuses

on creating a bridge between network management and natural language to analyze administrators' utterances as input through Natural Language Processing (NLP) to extract key-value pairs to create machine-understandable commands. The NLP pipeline starts from lexical analysis, parsing the syntax to find the semantics based on the domain, and finally, to the mapping of a particular operational meaning in the form of networking tasks that constitute the operational version of the intent. This approach requires enormous data sets of vocabulary from different languages and networking goals, which is a negative aspect due to the need for such resources. Furthermore, another drawback of this method is vagueness, since users can express their intents to the system in many different ways. To alleviate this problem, the definition of a unified well-structured specification language can be beneficial.

- The **mathematical approach** views network management as a bottom-up procedure with networking conditions and goals as finite states [5, 29]. By converting intents and network states to logical predicates, this approach tries to guide the network to a proper reaction. However, mathematical or graph-based strategies are limited to some specific types of intent. Also, they are categorized as low-level approaches that require several other technologies to complete their deployment.

- The **intent refinement approach** can complete the two previous approaches by developing refinement steps to complement other approaches and methods [18]. This approach allows users to modify reactions or intents in case of dissatisfaction with the network management. Another refinement method is achievable by adding a feedback system to the previous approaches.

Some example technologies using these three approaches are discussed in Section 2.2.

2.2 Advanced Network Management Technologies

2.2.1 SDN

In traditional low-level networking, the routing devices are responsible for both routing decisions and the actual process of packet forwarding. Dividing the network management architecture into different planes and adding programmability features to traditional network management added more flexibility to traditional management methods. Software-Defined Networking (SDN) architecture is one of the most paramount evolutions in this category [12].

SDN divides the networks into three abstraction layers: the *control plane*, the *data plane*, and the *application plane*. It centralizes routing decisions in the control plane layer and adds programmability features for operators to conduct the network based on network applications. This feature enhances the flexibility, implementation, and performance of networks on the one hand and diminishes the possibility of error occurrence on the other hand. The data plane handles forwarding the packets based on the control plane's decision-making, and the application layer helps developers to connect to lower-level devices to program network goals as instructions based on business requirements. The connection between the application layer and the control plane is handled by northbound interfaces such as Application Programming Interfaces (API) and Command Line Interfaces (CLI). Southbound interfaces such as OpenFlow handle the connection between the control and data planes. Despite the advantages of approaches similar to SDN, the gap between SDN APIs and network management as they are implemented at a lower level resulted in some challenges for its combination with intent-based networking. Northbound APIs can only partially support autonomic network management's complexity since business requirements are more complex than simple policies to be converted to machine language. Therefore, some pragmatic solutions are required to develop SDN-based architecture to meet different aspects of autonomic network management. Then some other technologies like ONOS were developed to provide an open-source control plane for SDN networks to alleviate issues regarding scalability and reliability [13].

2.2.2 ONOS

The Open Network Operating System (ONOS) [24, 4, 13] provides an infrastructure to implement Software-Defined Networks (SDN) that can deploy networking intents. It facilitates communication between network components and end hosts and adjacent networks and their management by providing a control plane through a network operating system. ONOS allows defined high-level goal-oriented networking specifications using an *intent framework*. Intent specifications are expressed as objects that are then compiled and translated to network low-level configuration called *installable intents* that can be dynamically installed and executed.

In ONOS, the Intent is an immutable object expressing an application's request to alter the network's behavior to provide autonomic networking capabilities. Intents can be described in terms of network resources, constraints, criteria, and instructions [24]. Also, the intent framework working with ONOS evaluates some metrics from the network under specific scopes to create intent objects to modify network behavior [24]. These intent objects are unchangeable, and the system can process the installation by converting them to flow rules model objects. Although intents are immutable, ONOS has a built-in mechanism to avoid conflicts between intents by prioritizing one intent over others in case of incompatibility between different intents.

ONOS is an open-source platform that prepares modular and distributed control planes for SDN-based networks. This platform has a layered architecture and can collaborate with other technologies like Merlin and Nile (see below). ONOS can be used as an infrastructure for an autonomic system in association with other technologies, but it cannot specify intents from business requirements. Therefore, specification modeling is required to take responsibility for the intent specification. For this reason, an intent framework subsystem was created in ONOS, which assesses intent objects derived from policies to compile and install them on ONOS.

The fact that ONOS might not offer as much flexibility or adaptability as an intent-based networking platform should be one possible restriction. For instance, it can be challenging

to modify or extend ONOS to satisfy particular needs or to support new technologies or protocols since this platform's intents are considered immutable objects. Notably, ONOS does not have a feature or language to express intents. Another potential restriction of ONOS is its need for adaptability with other intent-based networking platforms or technologies. Not all networking infrastructures can handle the ONOS dependencies to install and deploy intents. Thus, although ONOS is a robust and scalable platform, there might be better options for some use cases.

2.2.3 Merlin

Merlin is a language to implement network policies through mathematical logic and regular expressions [5, 29]. The input to the Merlin compiler, including network policy, physical topology, and mappings, are processed to provide static configuration for low-level networking devices such as switches. It was designed to mitigate the shortcomings of northbound APIs by classifying the packets, controlling forwarding paths, and specifying packet transformation and bandwidth limitations. Although this framework lacks the dynamism to modify defined policies, it can reduce this shortcoming by using negotiators as middleboxes in the network. Negotiators are run-time components designed to transform policies in run-time by communicating workload parameters among themselves. However, this can be an issue for the system's integrity in its implementation since some extra components are required to complete the processes that an autonomic network should manage. This issue can be resolved through a controller. Merlin uses a mixed-integer programming approach that concentrates mainly on integer-valued conditions. Thus, from the point of view of using Merlin as a general autonomic networking system, more elaborated quality models such as quality-based metrics or quality of service might not be feasible in Merlin.

2.2.4 Lumi

Lumi is a modular system for intent-based networking that helps intent users demonstrate their intent through natural language and refine it through its feedback system. It completes the procedures in four different steps [17]. In the first step, the information is extracted and tagged to be prepared for the next stage, where Nile intents are composed of the previous input. Nile and Merlin contribute to making an integer autonomic system with more dynamic features. Nile handles the intent expression by adding a layer of abstraction between natural language intents and network commands. Its integration with Merlin [5, 29] and Nile [5] can result in the highest level of abstraction. Lumi is of the most recent technologies in intent-based networking, which uses machine learning approaches and intent refinement to create a complete package. Lumi has a feedback system using DialogFlow (chatbot), which helps users enter instructions based on human language. Then this feedback is processed through Long Short-Term Memory (LSTM) to modify any intents based on user requirements. LSTM refers to sequential data modeling and other natural language processing activities to process human language input as intent based on keywords. For example, a keyword such as "Location" is helpful to process intents during directing network traffic to specific places. Lumi provides a chatbot for human operators to modify commands when required to enable more dynamism through intent refinement. Nevertheless, the most crucial shortcoming of these approaches is the need for a public and comprehensive database of commands, network intents, and human language to train the modeling based on that. Also, it might be challenging to adapt the combination of these systems, including Lumi with its chatbot interface for intent refinement, Nile to process intent through NLP, and Merlin to handle lower-level configuration policies as a package to all current technologies.

2.2.5 NEMO

The NEMO (NEtwork MOdeling) language is a domain-specific language to specify networking intents with a minimum number of commands based on intent-based primitives [30]. Intent-based primitives refer to high-level abstract concepts extracted from

policies, topologies, and services to define intents for networks. The structure of this language consists of three parts, including (1) objects to express network elements and resources, (2) operations for acting upon those objects, and (3) results for checking if the conditions of the networking environment are met or not. These objects are divided into nodes, connections, and flow.

This language was inspired by SQL logic in that its fundamental components are derived from Event-Condition-Action (ECA) rules, which means that following the occurrence of an *Event*, an appropriate *Action* is executed within a particular *Constraint*. It declares desired network behaviors or states based on declarative expressions to be adaptable by modifications in the system. To simplify intent expression, various groups of instructions are supplied by NEMO to define models and behaviors, prepare accessibility to resources, and connection regulation.

This language requires other technologies, such as data modeling languages, to form and execute its actions on the network. Moreover, it has built-in data types, keywords, predefined network models, and base methods of functionality abstraction like *Create*, *Import*, or *Update* to form the actions that can be taken into account by its defined actors, which are the nodes concerning the intent operators. These requirements can be a drawback since the integration of the whole network system might be affected negatively due to adding heterogeneity by different technologies on the one hand, and compatibility of the current network with all these NEMO requirements is another challenge. Other potential shortcomings of NEMO would be:

- Less granularity than traditional management to apply certain changes to the network.
- Dependency on the centralized management system and databases, or modeling languages such as YANG [44] to model and interpret intents. Yang models networking data similar to a tree-based construct which can be used to shape intents. Networking data includes network states and configurations.
- Limited to specific scenarios and intent types, such as traffic rerouting.

It is noticeable that this analysis of NEMO is based on the limited documentation of this framework that was accessible. This is a new approach, and its documentation needs to be improved; therefore, it was impossible to find more documents to investigate the detailed implementation of this technology.

2.3 IRTF/IETF Approaches

The Internet Engineering Task Force (IETF) [3] is the standards development organization for the Internet. According to the IETF itself, "The overall goal of the IETF is to make the Internet better." [1]. The Internet Research Task Force (IRTF) [16] is a sister organization, which"focuses on longer term research issues related to the Internet" [16]. Documents from the IETF and the IRTF are published as a *Request for Comments* (RFC).

The IETF has many "Working Groups", of which the *Autonomic Networking Integrated Model and Approach* (ANIMA) Working Group is pertinent to this thesis. The IRTF has several "Research Groups", of which the *Network Management Research Group* (NMRG) is pertinent to this thesis. NMRG has produced RFCs that address research issues related to Autonomic Networks [3, 19], and research issues related to the concept of Intent [10, 26]. ANIMA has produced RFCs that define the reference model, operation, and protocols for an Autonomic Network adhering to the concepts specified by NMRG [2, 6, 9, 11, 20, 28]. Further discussion of some of these RFCs will be given in subsequent sections.

2.3.1 RFC 7575: Autonomic Networking Design Goals

RFC 7575 [3] from NMRG determines autonomic network design goals and common terminologies and concepts of the subject to form an abstract modeling reference for other intent-based autonomic networking-related works. RFC 7575 outlines the design goals of autonomic networking:

- *Self-* properties*: Self-management policies, including self-chop policies, should be handled by autonomic functions and with minimal human or zero human intervention.

Autonomic functions are autonomic activities to drive a network without any requirement for configuration.

- *Coexistence with Previous Network Management Approaches*: Modernized autonomic management should be compatible with previous management methods. This goal can be achieved by setting management priorities for node management technologies and autonomic management, respectively.

- *Default Security*: Secure infrastructure through domain certifications and cryptography.

- *Decentralization and Distribution*: Both central and decentralized management methods should be available.

- *User-friendly Autonomic Northbound Interfaces*: Any interface designed for human interaction with network infrastructure should be simple to use.

- *Autonomic Reporting*: Information about the network performance and information should be gathered and reported to the users.

- *Abstraction*: The interactions between humans and the network should be designed at the highest possible level, with the lowest amount of detail about configuration or machine language.

- *Autonomic Control Plane and Common Autonomic Networking Infrastructure*: Creation of a common autonomic infrastructure with standard autonomic functionalities for the management of networks.

- *Decoupling of Function from Layer*: Autonomic functions are separated from the layers of networks which results in the possibility of creating them by all the devices from different layers, e.g., switches from layer two or routers from layer three.

- *Supporting Full Life-cycle of Devices*: This knowledge is required for the autonomic networks to reduce any requirement for inputs from outside the system. Thus, the network operates itself based on the defined goals and devices' life cycles.

In the third chapter, these design goals help us to form our solution regarding an integrated definition of network goals.

2.3.2 RFC 9315: Intent-Based Networking Concepts

RFC 9315 [10] from NMRG defines the term "intent" and gives a general overview of the functionality that goes along with it. The objective is to help create a consistent description of concepts and terms to serve as the basis for further defining related research and engineering problems and their solutions. Intent defines objectives and results in an utterly descriptive approach, outlining what must be achieved but not how. Thus, intent employs data abstraction to reduce the need for low-level device configuration and functional abstraction to encapsulate the logic behind the management of devices to accomplish a specific objective [10]. Abstraction means that the autonomic network can accomplish the desired objectives (i.e., a desired state or behavior) without requiring the user to define the specific technical methods for doing so. The autonomic network can recognize and ingest the intent of an operator or user and configure and modify itself in accordance with the user intent. An intent-based network can determine how to accomplish the goal independently. Also, users may need a single conceptual point of engagement with the network instead of modifying the entire network goal. Procedures that initiate remedial measures are required when network activity deviates from desired intent to prevent any risk of conflicting intents. Mechanisms such as restoring network compliance, notifying, and reporting to assist the network in articulating any changes in the original intent can reduce potential conflicts between the changes. Users should be able to verify and track whether the network is following and complying with intent using the services and interfaces provided by intent assurance. This is required to evaluate the success of actions made in the fulfillment process, providing crucial input that enables those functions to be taught or tweaked over time to

get the best results. Thus, since there might be multiple intents in one system or some particular intents to affect a specific part of the network without any effects on other parts, some mechanism to avoid the risk of conflicting intents or goals should be considered, such as the prioritizing mechanism in [24]. Southbound interfaces provides the feature of applying the same network configuration to diverse networking technologies. ONOS and southbound interfaces provide an infrastructure to model configurations with languages such as YANG. This method of implementing networks can result in a portability feature for intent-based networking. SDN enables scalable intents for intent-based networking and enables central management and control of network devices through an SDN controller. This scalability is because SDN separates the control and data planes. Manual configuration of every device is omitted, and thus the controller can utilize high-level "intents" to define the desired state of the network. This enables localization, on the one hand, and quicker adaptation to shifting network circumstances and requirements, as well as more effective and scalable network administration, on the other hand. Therefore, adding more devices or resources to the network is more manageable as the controller should adapt to the changes rather than the entire system components individually. Also, intent-based networks require principles, guidelines, or standards regarding security considerations. This security can include the security of the intent-based network architecture, reducing the effects of false and malicious intents, and creating statements about security policies. To summarize, RFC 9315 provides some characteristics of intent-based networking with general examples of what can or cannot be categorized as intent. Other current technologies, such as ONOS and SDN, can also complete the scheme of an intent-based network.

2.3.3 RFC 9316: Networking Intent Examples and Classification

Intents are high-level policies to conduct an autonomic network and to abstract an intent; three principles should be considered since an intent model is composed of these concepts, namely, "context", "capabilities", and "constraints" [23], which are the fundamentals for intent classification. Context justifies the applicability of an intent model to an autonomic

network, and capabilities refer to the operational aspect of an intent model, which relies on how declarative the intent and its programming language are, where the capabilities can be regulated through conditions by the constraints. Depending on the context, tasks and magnitude of the intents can be deduced, which results in a more explicit class determination for intent solutions like carrier, or DC network, intent users, and scope. The system can then determine some properties of the intent by recognizing its resources and capabilities. These systems' capabilities then aid in identifying some properties of the intent, such as intent types, users, scope, and network scope. The system may identify rules for various intended users or solutions due to constraints that specify the intent's restrictions. RFC 9316 [23] focuses on the intent classification to categorize this concept based on an intent classification methodology. This categorization helps narrow the wide range of network goals into groups more expressible by network technologies. It also gives a clearer view of what can be categorized as an intent *Intent Types*, and who can use these intents as *Intent Users*. In general, using this methodology, the actions listed below can be used to classify an intent:

- *Determine the solution's intent type* (e.g., carrier, enterprise, and data center). Carrier network solutions are for high-volume, wide-area communication services run by telecommunications firms, also known as service providers, that offer voice, data, and video communication services to end consumers. Data center network solutions are used for managing and processing enormous amounts of data. They frequently employ technologies like Software-Defined Networking (SDN) and virtualization to facilitate communication between servers, storage devices, and other data center hardware. Enterprise network solutions are used for corporate operations and organizational communication. These networks securely and dependably connect workers and systems within the organizations across several locations, such as various offices or distant sites.

- *Determine the intent user types* (e.g., customer, network operators, service operators). The network's aims and objectives are defined and used by the intent user. A human network administrator, a network user, an automated system, or a hybrid of the two may be involved. These users can be professionals aware of networking concepts or

non-professionals without networking knowledge.

- *Determine the intended concept of various intents based on the networking context* (e.g., network intent, customer service intent). This refers to the different categories or types of intents that can be defined and managed within a network. Examples of this categorization are as follows. Customer service intents refer to the network intents constructed based on service-level agreements to satisfy customer needs. Network and underlay network service intents refer to the goals and objectives for managing and maintaining the underlying network infrastructure services, configuration, optimization, and other tasks in an intent-oriented network provided by underlay network administrators and service operators. Network and underlay network intent types refer to objectives for network configuration and other tasks related to network resources such as switches and routers and their life-cycle management. Cloud management intent types are the intents created to manage resources and their configuration in DC networks. Cloud resource management intent types can be formed to handle the life cycle of these resources. The goal of the security, quality of service (QoS), and traffic engineering as strategy intents is to ensure that users can receive a secure, high level of service from the network. The goal of the configuration and monitoring policies and auto-recovery is to ensure that the network is functioning correctly and that any problems are found immediately as another strategy intent type. In order to make the prospect of an immediate recovery, this would include putting up monitoring systems to track the performance of the network and its services and automatically producing alarms if any faults are discovered. Ensuring that the network is developed and designed to provide the greatest possible service is the strategy intent for design models and rules for networks and network service design. By taking into account variables like scalability, performance, security, and reliability, as well as the types of services that will be provided on the network, the design models and policies for networks and network service design aim to make sure that the network is built in a way that will provide the best service to users. In order to ensure that

tasks like network maintenance, troubleshooting, and upgrades are carried out quickly and with the least amount of user disruption possible, the strategy intent for designing workflows, models, and policies for operational task intents would be to ensure that the network is operated effectively and efficiently. Operational task intents are defined as the particular goals and objectives for actions carried out by network administrators or operators to maintain and administer a network. For instance, operations should be performed with the least disruption possible for users during network migration to a new platform or location, device replacements, and software upgrades.

- *Determine the intent scope based on the intent solution* (such as connectivity and application). Intents are applied within a specific scope dependent on the granularity level to regulate the intended networks' behavior. If the intent impacts a connection, it refers to connectivity scope. It refers to the application scope if it specifies the apps affected by the intent request. It is security/privacy scope if the intent specifies the security characteristics of the network, consumers, or end users. When the intent specifies the network's QoS characteristics, it is the QoS scope.

- *Determine the network scope based on the intent scope* (e.g., campus, radio access). Regardless of the intent user type, the network, or network components, that represent the intent targets are impacted by the user's intent request. The network scope refers to the target that the declarative policy affects in the network. Some examples are physical network elements, campus networks, cloud edges, and cores.

- *Determine the abstraction concepts* (e.g., technical, non-technical). The level of technical detail in intent and whether or not technical network information must be disclosed to the user after the intent is implemented is referred to as intent abstraction.

- *List the criteria for the life cycle* (e.g., persistent, transient). Persistent intents have a longer life cycle compared to transient intents that are policies temporarily effective to the network.

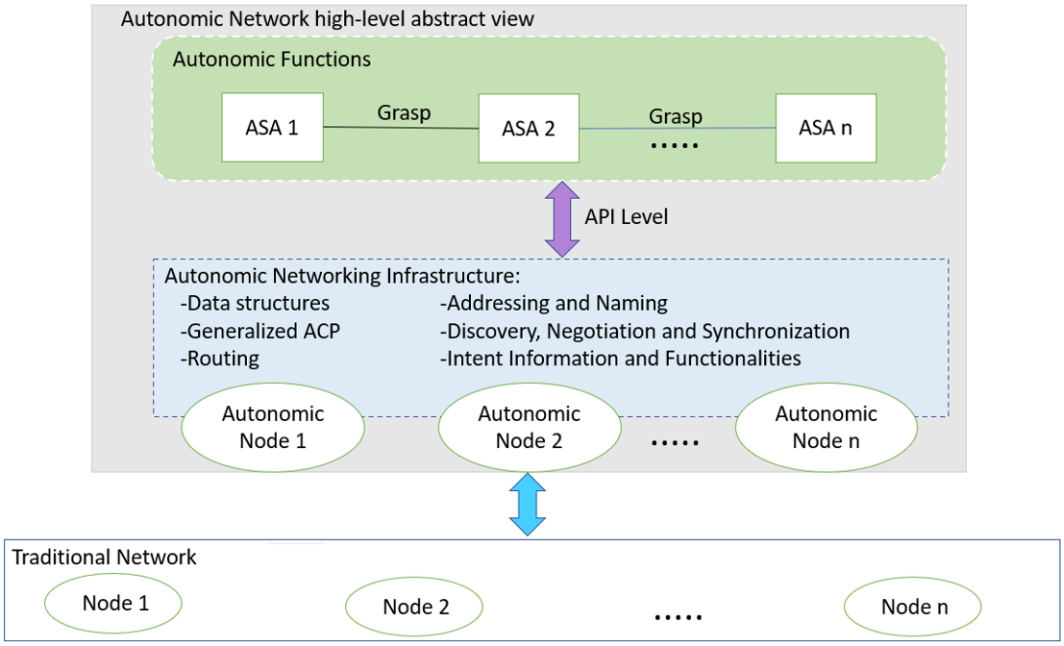

Figure 1: ANIMA abstract representation of autonomic network components

2.3.4 ANIMA Documents and Goals

As discussed in Section 2.3, NMRG has given a set of design goals for autonomic networks [3]. The ANIMA Working Group of the IETF has subsequently developed a set of five documents [2, 6, 9, 11, 28] that define how to achieve these design goals in a coherent way. The various nodes in an autonomic network (switches, routers, computers), called Autonomic Network Elements, are members of a *domain* [2].

Each node in that domain contains a number of Autonomic Service Agents (ASA), which, either alone or in collaboration with peer ASAs in other nodes, express the actions required to achieve a particular autonomic function. The ASAs obtain services from an underlying Autonomic Network Infrastructure (ANI), as depicted in Figure 1.

The ANI depends in turn on the underlying services provided by the local operating system of the node, including access to the physical networking devices. One component of the ANI is the Autonomic Control Plane (ACP), which provides for inter-node communication [11]. This communication is carried by a purpose-built protocol called GeneRic Autonomic Signaling Protocol (GRASP) [6]. The GRASP Application

Programming Interface (API) [9] provides access to the ACP, offering functionalities that are especially useful in Autonomic Networks, specifically: *Discovery, Synchronisation, and Negotiation*. An ASA will use these functionalities to achieve the goal(s) of the autonomic function(s) that it implements.

GRASP introduces the concept of *GRASP Objectives* to support communication of the necessary data among the autonomic nodes. A GRASP objective can carry any arbitrary data structure, and includes a set of flags to determine the purpose of the data, i.e., for negotiation or synchronization. The data in a GRASP exchange are encoded using the Concise Binary Object Representation (CBOR) [7]. This minimizes the "on-the-wire" cost, which will be especially important in low-power environments such as the Internet of Things.

As one of the design goals of RFC7575 is *Default Security*, enrolment of a node into a domain is performed by a zero-touch bootstrapping protocol called Bootstrapping Remote Secure Key Infrastructure (BRSKI) [28]. This ensures that all members of a domain have the ability to trust each other, and to verify that that trust remains valid.

While the ANIMA design has as its target the support of "Intent-Based Networks", the ANIMA documents do not provide any specification of how to represent "Intent". However, the flexibility built into GRASP objectives and CBOR ensure that GRASP objectives will be able to transport any such representation that may be developed in the future.

2.4 ASSL

The Autonomic System Specification Language (ASSL) is a declarative specification language used to represent the structure, behavior, and communication of a group of elements aiming to achieve a common task by expressing system-level and element-level goals [14, 31, 35, 42].

ASSL creates the formal design of autonomic systems based on the self-management characteristics of the system. In the ASSL paradigm, autonomic elements are compared to software agents that can control their actions and interactions with other autonomic elements to produce or utilize computing services. This framework simulates the autonomic behavioral models through its unique formal notation, which results in a Java code skeleton as its output.

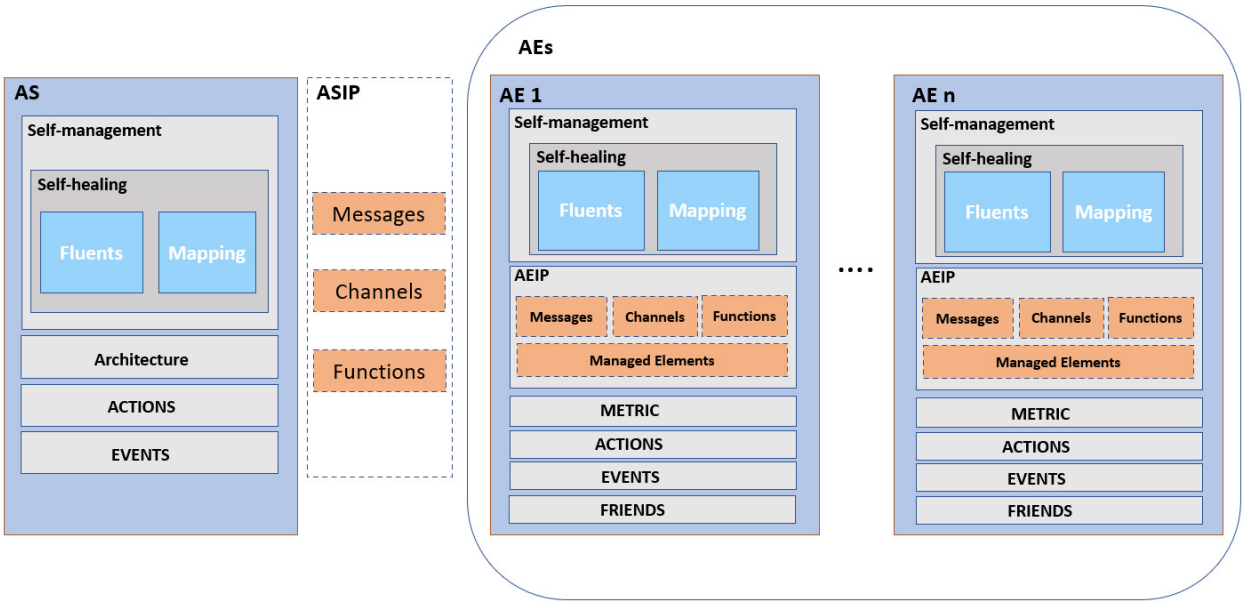

Figure 2: ASSL components abstract view

Moreover, this framework contains a validation procedure as a part of its toolkit to check the consistency of the specified code against the semantics of the framework grammar. Generally, ASSL views autonomous systems (ASs) as being made up of autonomic elements (AEs) that communicate via interaction protocols. ASSL is defined through formalizing multi-tier levels with a hierarchical approach to specify these building blocks. The ASSL tiers and their associated sub-tiers are abstractions of various elements of the intended autonomic system, as depicted in Figure 2.

Operationally, upon compilation, an ASSL specification is translated into an event-driven reactive system specification language that enables the expression of the structure of a collectivity of elements that participate in standard behavior that is achieved by orchestrating each of their local behavior. ASSL has been used to represent the autonomic behavior for NASA multi-agent-based exploratory space probes collaborative missions [32, 33, 37, 38], group-based space probes telecommunication behavior [34, 36], the specification of real-time reactive systems [25, 39, 41], self-scheduling robotics [26], autonomic pattern-recognition systems [40].

The most fundamental parts of an autonomic system expressed using ASSL are managed

elements or managed resources and the autonomic manager as its controller. The controller manages the autonomic system based on the high-level goals for the system to reach and the metrics, events, and actions related to attaining these goals, as expressed by the specification of the role of each managed element in the system.

In order to express autonomicity, the system as a whole is defined with its higher-level goals and behavior, which enables it to dynamically keep meeting its goals through the event-driven monitoring of metrics and triggering actions when the metrics go out of acceptable boundaries. In the same way, each autonomic element involved in the system is defined with its own goals, metrics, and actions.

Suppose a system or element action requires another element to be notified or prompted into action. In that case, a message is sent to this element, triggering an event, which then triggers an action to be taken by this element. The specification of the actions aims at changing the system's state or an element's state so that the metrics and goals of the system go back to the desired state as measured by the expressed metrics. This mechanism, in turn, requires the specification of interaction protocols between the elements and an event-handling mechanism that enables the elements to react to messages, metrics going out of bounds, or goals ceasing to be met. Each of the different tiers of the specification is expressed in more detail in the sections below.

The *Autonomic System* (AS) tier provides the infrastructure for the entire system to handle self-chop mechanisms at the highest system level. The autonomic system's general rules, architecture, and policies are defined at this level of abstraction. To construct the autonomic system, three main components are required: (1) the general rules or plans of the system as a whole, (2) the building blocks as autonomic elements, and (3) an interaction protocol as the means of communication between these components to enable collective behavior. Three different approaches are available to achieve these components. First, using a top-down approach, events and metrics can be determined from a global perspective of the system view. In addition, with the bottom-up approach, details from low-level configuration can be extracted to shape condition measurements. The final approach consists of the

contribution of the previous two, which leads us to build an autonomic system by considering both abstract and detailed-level properties.

In a manner analogous to multi-agent system specifications, autonomic system specifications are composed of individual components called *Autonomic Elements* (AE), which are capable of managing themselves by running local autonomic behavior of their own. In order to achieve collective behavior, the autonomic system and the autonomic elements need a means of communication between them which can be expressed as *Interaction Protocols* (IP). There are two levels of interaction protocols, one for the AS tier for general system-level communications called the *Autonomic System Interaction Protocol* (ASIP) and the other protocol for the AE tier to handle communication between the autonomic elements, called the *Autonomic Element Interaction Protocol* (AEIP).

ASSL focuses on two primary architectural styles, namely, decentralized and centralized. For the first one, aptly referred to as *Grid*, autonomic elements are directly connected in a linear/grid format. For the second style, autonomic elements form a centralized node through grouping, where an AE proxy will represent the collectivity of autonomic elements in the group council. The list of all autonomic elements (AEs) belonging to the same autonomic system and groups with their members and their direct and transitive dependencies are to be specified to express the architecture. Autonomic elements can interact with each other directly when declared as *Friends* by forming a private negotiation group since they are connected as neighbors or in the same group. Declaring autonomic elements as friends gives each other access to their internally declared tiers.

Suppose we have a network domain including a router that can be named as registrar to enroll other devices into this domain through certificates. The network topology can be depicted through a concept called *Group* in the ASSL specification, where devices are considered autonomic elements or the group members of the topology. These devices depend on each other from different aspects of their collective behavior. These dependencies can be direct or transitive, and the groups and a device can be mentioned as the council of the autonomic domain. In this scenario, the registrar router should be assumed as the council

since it has all the information gathered from other devices and is the domain's central management.

Another central aspect of ASSL specifications is *Metric*. Metrics are measurable quantities or qualities from an autonomic system that can be monitored that can be beneficial for controlling the system [42]. Related to the notion of metrics, the *Threshold Classes* specify ranges of metric values of particular interest or meaning for the system specification. Based on these classes of values, a metric is, in turn, determined as valid or invalid. For example, once a metric is deemed as invalid by its value belonging to one of the threshold classes, an event can be triggered, which is then trapped by a related *Fluent*, upon which *Mappings* are defined that map them to an *Action* that will, in turn, hopefully, rectify the situation to make the system state to go back to a desirable state. Some examples of the metrics can be message size, specific time measurements, or information extracted from the quality of service purposes. In order to extract the value of the metrics, the metric *Source* can be specified, which is expected to be available from a managed element to provide specific information for an autonomic element's state. The metric controller module is responsible for handling metric operations for both metrics and metric sources.

As an ASSL specification essentially declares a system of somehow independent agents to collaborate on a collective problem to solve, communication between these elements is necessary. *Messages* are defined at both tiers of interaction protocols as a means for negotiation between autonomic nodes since information can be transmitted between autonomic elements through messages. Negotiation in ASSL consists of predefined messages for general purposes, such as initiating and terminating an information exchanging session, and particular empty messages to exchange data to achieve specific goals. For instance, if an autonomic element requires a metric, it sends an empty message to the targeted autonomic element, which fills the message skeleton with the required Metric it has and returns it to the previous autonomic element.

Channels are a communication interface between autonomic nodes to provide a link simulation for transferring messages. Each message can be exchanged through a *Channel*.

Channels are often unidirectional; however, they can be grouped as channel buses and transfer messages concurrently. Also, a controller module provides different functionalities to create, manage or adjust channels and their settings. Channels can express communication between autonomic elements in two private and a public mode where the former transfer the public messages and the latter transmit the private messages among autonomic elements, which are members of a trusted group specified as friends, to improve privacy and reduce the risk of malicious attacks. To complete the communication procedure, *Functions* facilitate the channel utilization by conducting the stream of defined messages to the correct channel or channels.

Behavioral specifications specified as one or more conditional boolean expressions that can monitor and measure system performance are service-level objectives. A set of executable activities or responses of the system to any service level objectives, policies, or behavior models defined by the autonomic system are called *Actions*. These actions can contribute to different tiers and sub-tiers of ASSL, such as the AE tier, after meeting a specific condition regulated through a particular "action contract". States (situations or cycles of incidents) of the autonomic system based on a specific duration of time are *Fluents*, which are at the essence of the concept to determine self-management policies. Fluents can be initiated or terminated by *Events*. They represent a specified timed sequence that tracks when a defined condition is fulfilled, then allows an attached *Mapping* to specify an *Action* to be triggered in order for the fluent to be deactivated through a fluent-deactivating event. Each event is evoked by a specific duration related to timing or processes in the system. *Mapping* clauses map the conditions presented in the fluents to the desired actions that, in turn, will eventually change the system's state.

Autonomic elements rule one or more *Managed Elements* through managed resource interface as its communication model. These interface functions can specify four clauses such as *Parameters* to introduce attributes or functionality to the interface. In essence, the managed elements are the non-autonomic elements used and controlled by the autonomic system specification. The manageability interface is the means for communication with

other autonomic elements and entities in the autonomic system. An autonomic element can be responsible for one or more managed elements. This management is handled through defining a Java-like interface, or a named set of functions without implementation, for each managed resource where the function's return type and its parameter types are predefined in the ASSL framework.

One of the characteristics of an autonomic system that can be expressed in ASSL is *Self-Healing*, which helps the system to correct itself in case any undesirable state is reached and to prevent any possible problems or system errors related to this undesirable state. In this case, errors are internal and related to system functionality or failure; therefore, different possible undesired states can be considered in this specification.

Self-management capability divided into self-chop objectives is the primary purpose of autonomic computing, and its policies can be determined through fluent. Self-management policies can be expressed for both the AS tier in general, as well as each of the AE tiers in particular. For the general tier, the self-chop policies are specified based on the general and public rules. Regarding the AE tier, these characteristics are specified in more detail from the perspective of a specific autonomic element. At the same time, each of the autonomic elements is represented and managed as a minor autonomic system with its policies and conditions, such as fluents that should not conflict with the whole system. ASSL provides a collection of tools as ASSL toolset to form an appropriate environment for developing, maintaining, and executing autonomic behaviors. This toolset includes:

- *ASSL Editor and ASSL Grammar Modules* including the syntax of ASSL that equip the autonomic system to express autonomic behavioral models. Also, Graphical User Interface (GUI) is another tool that supplies the modeling system with a means for interaction between the user and autonomic system models,through which the user can write, edit, and compile the specifications.

- *ASSL Compiler Module* allows the compilation and parsing of the specified autonomic behaviors into an intermediate code to analyze the logic based on the semantics.

- *ASSL Java Generator Module* supplies a means for verifying the autonomic behaviors through consistency checking and then converting them to executable code.

The ASSL toolset generally creates integrity for the whole autonomic system, while specification, maintenance, and verification are handled in a unified system.

2.5 Problem Statement

Intent is a high-level construct, which will be expressed in some sort of problem-specific (high-level) language. In RFC9316 [23], several intent examples are given, each of which is written in English. To achieve the desired effect of the intents, it is necessary to transform the declarative form of these intents into commands that can be understood by the existing software on individual network devices (switches, routers, hosts, etc.). This requires a set of procedures, which will take declarative information from the point of interaction with the (human) user, and distribute specific commands to the affected devices. Many of the actions required to give effect to an intent can be carried out independently by network devices, which suggests that organizing an *Intent-Based Network* as an *Autonomic Network* will be an effective approach. Clearly, however, other actions will require coordination among a set of devices, so a desirable property of a proposed solution will be the provision of controlled information exchange within a group of related devices.

Intent has different meanings in different application areas. Various approaches to expressing intent have been investigated, along with their advantages and shortcomings. Once a definition of intent is accepted for a particular domain, it is necessary to determine how to effectively express this intent, so that the desired outcomes can be achieved within the intent-based network. Our exploration and analysis of the existing intent-based management technologies have led us to two major problems, which we identify as our problem statement for this thesis:

- There is a variety of definitions of intent, as different research groups and network users have their own specific descriptions of the network goal. For the purposes of

this thesis, we will adopt the definition in RFC 9315 [10], and then formulate a more comprehensive set of *Intent Objectives*, which will cover the core characteristics of a networking goal from different perspectives.

- The set of examples in RFC 9316 [23] is written in English. The concepts of autonomic systems in ASSL are expressed in a high-level language. To demonstrate that autonomic systems (as represented by ASSL) are, in fact, a good basis for constructing an *Intent-Based Network*, we will show that the mapping from English to ASSL is feasible, for a wide range of intents, which are chosen to ensure that all of the Intent Objectives can be met.

Chapter 3

Solution

A definition of *intent* is given in RFC 9315 [10]:

> A set of operational goals (that a network should meet) and outcomes (that a network is supposed to deliver) defined in a declarative manner without specifying how to achieve or implement them.

In order to evaluate the expressiveness of a chosen representation for intent, we require a more detailed description of the essential features of intent. Development of this set of *Intent Objectives* is the first goal of this chapter.

A definition of an *Intent-Based Network* (IBN) is also given in RFC 9315 [10]:

> A network that can be managed using intent.

Note that this definition does *not* imply that the network is also autonomic.

Finally, a definition of an *Intent-Based System* (IBS) is given in RFC 9315 [10]:

> A system that supports management functions that can be guided using intent.

In such a system, there will be:

1. a point of interaction with the user or administrator, thorough which the intent will be specified;

2. a transformation of the form in which the intent is given into an intermediate form that can be interpreted by the IBN;

3. a distribution of the transformed intent to the device(s) that have to alter their state in order to produce the desired effect.

Given that we could find no detailed documentation for NEMO, we demonstrate that ASSL is a suitable intermediate form, and is able to express most of the intent objectives. This is the second goal of this chapter.

3.1 Intent Objectives

An autonomic system (AS) can use the objectives, such as intents in autonomic networks, and features as generic characteristics. AS objectives are considered system needs, while AS properties could be considered standards outlining fundamental implementation processes. As we discussed in Chapter 1, the vagueness of the definition of "intent" in intent-based network management has led us to delineate our definition. This characterization can be viewed from three main perspectives: language aspects, execution, and deployment criteria. The characterization is inspired by the positive aspects of current intent-based networking technologies, and by documents focusing on intent-oriented concepts, as summarized in Section 2.3.2.

Since different crucial characteristics were mentioned for intent-based networking and intent, in diverse documents, and existing studies, we created our list by gathering the crucial aspects of intent-based networking from different resources as an integrated list. The intent objectives are mandatory requirements for a solution that enables the expression and deployment of networking intents, which includes the fact that the underlying intent-based network should be able to respect these properties.

To summarize, the characteristics of intent-driven networking expression and implementation are the following as discussed in Section 2.3.2:

- *Abstract Formulation* [3, 10]: The intent expression language should enable the expression of intents in a language somehow close to human language and enable the user to express high-level intents without relying on low-level technical networking terminology.

- *Declarative Outcome Formulation* [10]: The intent expression language should enable the declaration of a goal for a network instead of a procedure to be followed to achieve this goal. The compilation/interpretation of the intent implemented in the intent-driven networking system should automatically translate the desired goals/outcomes into an operational solution that enables the network to achieve the stated goals.

- *Portability*: The operation of the intent-driven networking system should have a minimal dependency concerning lower-level configurations and be minimally bound to platform-specific considerations. It should minimally rely on specific tools to be installed outside of its operation. If it does depend on any of these, the intent-driven networking system should automatically adapt its configuration when the intents are deployed. An intent's formulation should not need to be changed when it is deployed in a different context.

- *Distributed and Local Behavior Management* [3]: The operation of the intent-driven networking system should orchestrate the synergy of distributed networking elements that participate in the execution of intent. The operation of the intent-driven networking system should also enable each networking element to express and execute its own local goals independently from the system-level goals in which they participate.

- *Composability* [10]: An intent can be expressed as a module that clearly defines its interactions with the exterior, i.e., its interface, in a way that (1) allows other intents to interact with it in order to provide composite intents that can express and deliver more than if they would not be combined and (2) allows the intent to be seamlessly replaced by another intent that exposes the same interface.

- *Efficiency* [42]: The intent expression language should allow the succinct expression of intents, whose translated meaning corresponds to the execution of complex operations inferred from the original intent expression. The implementation of these operations should aim for efficiency of execution.

- *Scalability* [22]: The operation of the intent-driven networking system should consider scalability when deploying an intent in a context where the scale of the controlled network is changing. Scaling up the complexity of the controlled network, in either number of member nodes or number of intents, should not require the execution of the intent to increase resource consumption or reaction time to unacceptable levels.

- *Monitoring Capabilities* [42]: An intent-based network should monitor its behavior to verify if the network is working based on the defined autonomic behavior. The operation of the intent-driven networking system should enable deployment of intents that not only constitute static commands to be executed once upon deployment but should also enable the deployment of dynamically deployed intents that require constant monitoring of a situation and ensure that a goal is constantly met over time.

- *Security* [3]: The operation of the intent-driven networking system should rely on secure interactions in the deployment of intents. Any member of an autonomic domain must be authorized by identification, such as a certificate granted by a domain certification authority. Nodes in the autonomic system use these certificates to identify their surrounding nodes, define the domain's edges, and secure cryptographic communication in their domain. Also, different domains can securely verify and communicate with each other by relying on a mutually agreed trust anchor.

- *Autonomic Reporting* [3]: The operation of the intent-driven networking system should offer network visibility through reporting that ought to take place across the entire network. These reports should be narrowed to the events important to the user, not the unnecessary lower-level details. The network itself should gather and aggregate network-related data, including metrics, and deliver it to the administrator or user in

a consolidated manner, which can then be used to measure the effectiveness of the deployed intents.

3.2 Using ASSL to Express Intents

The ASSL [42] framework can cover the intent specification in a structure where the characteristic of an autonomic system is considered. This results in a unified view of the autonomic network, which eases the procedures for modification or managing the network based on the intent. The other positive aspect of using the ASSL framework for an intent-based autonomic network is to reduce the use of data-based driven technologies causing human intervention, such as NEMO, which requires queries as a part of its procedures. ASSL is designed to provide autonomic computing for autonomic systems, and it has not been used for expressing networking problems. In this chapter, we will demonstrate that we can use this framework for the expression of networking problems. Since ASSL is a simulation of an autonomic system with its communication channels and building blocks, the concept of the Autonomic Control Plane (ACP) as the logical control plane of the autonomic network can be included in the intent specification. As well as simulating all the previously described ANIMA notions as stated in Section 2.3 and Section 2.3.4, ASSL also provides us with the means of expressing our intent objectives as explained in Section 3.1.

To resolve the lack of an intent specification language, we can use the ASSL framework since it provides the means of expressing intents in a formalized simulation-only manner for the autonomic networks. Analogous to an autonomic system, an autonomic network can be developed from behavioral models and a control loop to manage the network in an autonomic manner. The hierarchy of ASSL allows us to construct the autonomic network in the following steps. Firstly, the `AS` tier provides the means to create a global `AS` perspective for the autonomic network by presenting the general network rules and metrics necessitated to define the public characteristics of the intent, which is accessible by all the network nodes. To construct a communication tool under this layer, the AS-level interaction protocol (`ASIP`) can be specified. The network topology can be shaped by the AS's individual

components stated as AEs with their autonomic behavior handling the general network rules from `AS` perspective. A set of rules particular to an `AE` tier which can include SLO and self-management policies, an `AEIP` as the private communication protocol, friends as a circle of trusted autonomic nodes, recovery protocols, behavioral models, network states, and intent metrics for particular autonomic nodes. Moreover, it is possible to define interaction interfaces presented as managed elements `AEIP` to form particular methods to manage the devices connected to the autonomic network.

The ASSL perspective on autonomic systems is based on multi-agent systems, in which the autonomic elements are designed for managing resources and providing services as individual components under the `AS` level. Considering an autonomic network similar to an autonomic system, its architecture topology can comprise the relationship between separate or groupings of autonomic nodes. This architecture topology contains two models; First, the grid architecture as shown in Figure 3 where the autonomic nodes connect in a non-hierarchical and decentralized manner, and the second style as shown in Figure 4, autonomic nodes can be managed by a centralized node called `AE` Council composed of conceptual `AE` proxies as representative of AEs and help us to build a more flexible architecture. AEs can be the representatives of autonomic nodes as they abstract their autonomic behaviors. Also, it is notable that, in some scenarios of networking the devices, or in other words AEs are connected to each other directly, however, they may need centralized management to take their orders from.

The fundamental basis for coordinating the autonomic networks is the intent of high-level abstract policies, including conditions and specific measurements to evaluate the network's performance. These intents should be driven in an autonomic network environment capable of providing AC self-management policies (also known as the self-chop properties): self-configuring, self-healing, self-optimizing, and self-protecting. ASSL formal model supplies these capabilities in its hierarchical architecture for both node and system levels.

Self-management policies can be expressed with fluents driven by events and mapped to actions via mappings. To explicate more, fluents are the means to define diverse states

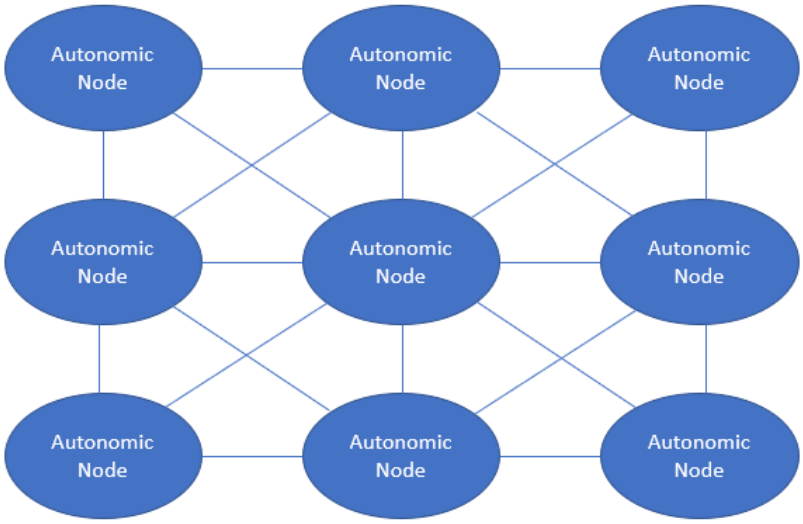

Figure 3: ASSL grid architecture

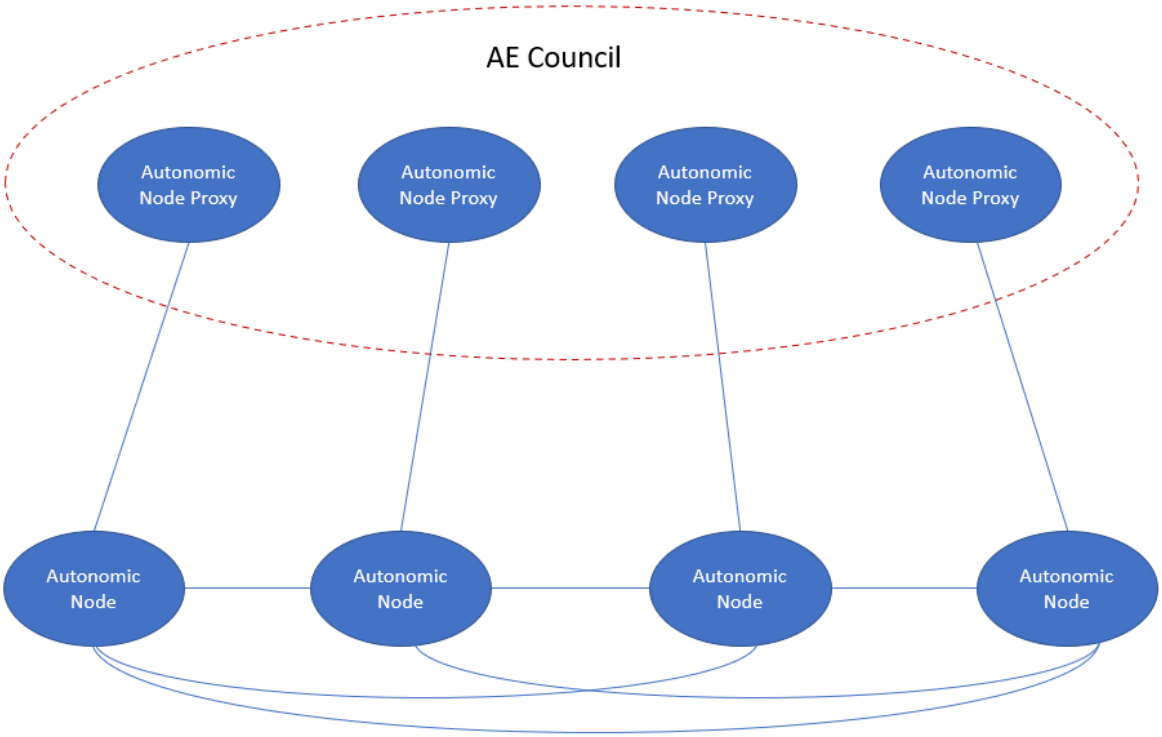

Figure 4: ASSL centralized architecture

or conditions that can occur in the autonomic network during a node or intent life cycle where these states can be initiated and terminated by events which are the means to specify time duration for any context in ASSL. Besides Self-management policies, there are service-level objectives (SLO) to measure the performance of the autonomic network based on particular metrics. The conditions stated by self-management policies can cause different operations under the intent consideration, which actions can express over `AS`, `AE`, and `ASIP` tiers based on the privacy level. In other words, actions can be demonstrated as responses to behavioral models of the autonomic network based on the intent and expressed as SLOs or self-management policies. Also, the coordination between the mentioned conditions or states and corresponding actions are handled through Mapping specifications.

Reactive and proactive self-healing are the two modalities under autonomic computing consideration. When operating in reactive mode for autonomic networks, an `AS` should detect network faults, recover from them, and, if applicable, fix them. In proactive mode, an `AS` keeps an eye on vital signs to identify and prevent health issues before they arise or reach unfavorable levels.

Autonomic elements serve as the foundation of autonomic systems. Therefore, they can simulate the abstract version of autonomic networking nodes as a basis for autonomic networks with the capability of simulating autonomic functionalities. The `AE` architecture is depicted in Figure 5. An `AE` typically extends programming constructs such as objects and services to form a software unit to encapsulate a node-level self-management mechanism with its rules and conditions, clear conceptual dependencies, and stated interfaces.

Public interaction protocol (`ASIP`) and private interaction protocol (`AEIP`) are indicated by the ASSL framework to provide abstract connectivity between autonomic nodes in two levels. For `AEIP`, when two AEs have an "agreement" on it, they can communicate across their `AEIP` channel and send private messages to each other.

Channels can simulate a conceptual communication link between the autonomic nodes to provide an abstract negotiation protocol, which is one of the fundamental contexts defined under autonomic networking. Negotiation is essential in autonomic network infrastructure

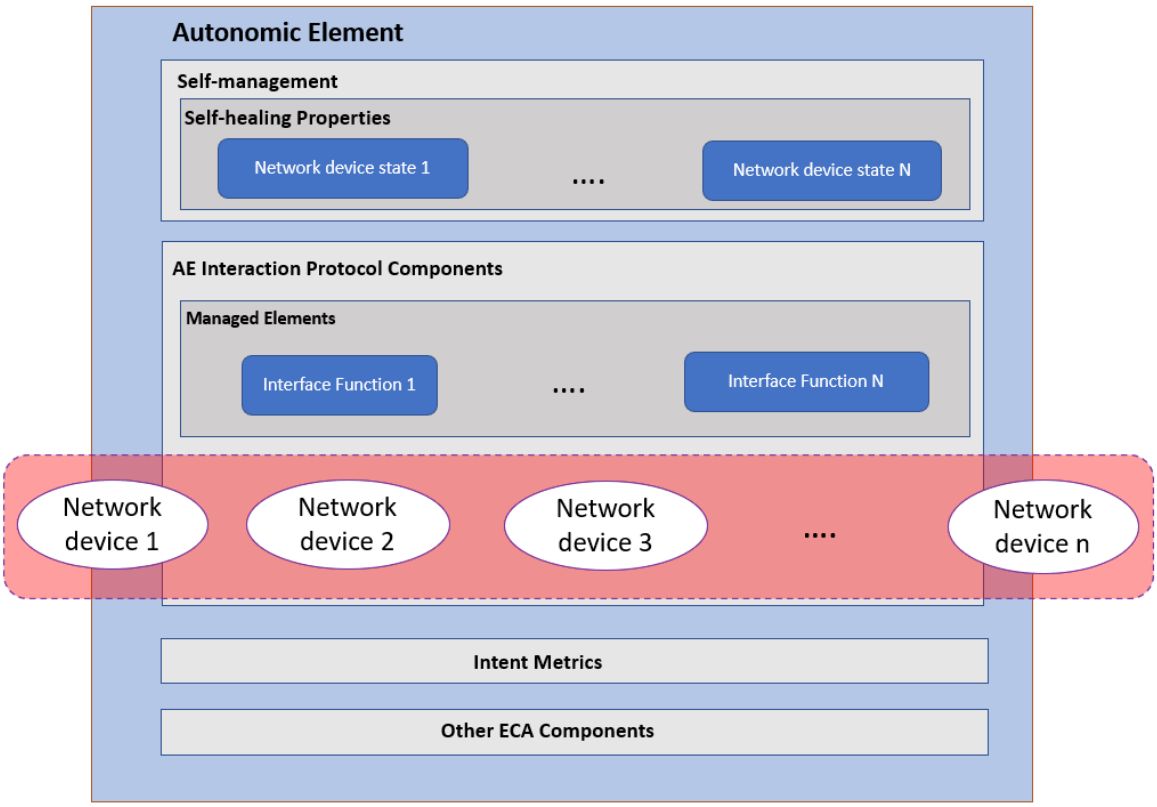

Figure 5: Autonomic element architecture

to allow networking nodes to exchange information about different subjects, like their states or intent metrics. In ASSL specification, messages, as the fundamental component for its negotiation mechanism between autonomic elements, are responsible for transmitting data of different types over private or public channels. A message format is specifically designed for negotiation concepts in ASSL that can be formed by beginning and end messages between the expression of autonomic nodes. To complete the negotiation process among autonomic nodes, we need to connect messages to channels through the *functions* of ASSL.

Intents in autonomic networks can be composed of different conditions at different levels, and to satisfy this characteristic of intent, there are more condition statements available in the ASSL framework. Guards are another possible conditional statement that can be expressed to limit the execution of particular actions or events. To explain more, this clause can enhance the adequacy of other clauses by adding more details to restrict that prompt

for certain conditions or metrics derived from intent objectives.

Any measurable content or parameter in autonomic networks extracted from monitoring or other resources can be equivalent to metrics from the ASSL framework. The notion of metric in autonomic networks can be amalgamated with intent definition since the intent is declarative operational policies including different measurements, which can be categorized based on intent classification criteria [23]. The metrics extracted from this classification can be quantified, qualified, or a composition of both, and ASSL provides different types of metrics to cover quantitative measurements, such as variables extracted from managed resources, and qualified metrics, such as performance or timing of the responses, which can be specified in ASSL based on Quality of Service or other intent scope properties. In ASSL, threshold classes specify the acceptable ranges for metrics to be assessed by the ASSL compiler. Therefore, this capability is beneficial to evaluate if the autonomic network is performing based on the specified intent, which results in network assurance.

The managed element is the resource working with ASSL such as network, server, or other software entities, which can be managed through a manageability interface made up of sensors and effectors, where the former gathers information about the managed element's current state and state transitions, and the latter alters the AE's state. The interface functions for network-managed resources can be used by ASSL actions to control managed elements or by ASSL metrics to acquire data on managed elements' characteristics, such as network performance. These empty generated skeletons add more flexibility to the design of the autonomic network, particularly if the entire network is not autonomic, the non-autonomic devices can be controlled based on the network intent as these managed resources without any interference with the autonomic control loop.

One or more managed elements in collaboration with an autonomic manager controller make up a primary control loop, which is the fundamental aspect of the logic behind autonomic network management. Based on observed measurements and events defined by the intent conditions and objectives in the autonomic network management criteria, the autonomic manager decides how to manage its related resources. The monitor, analyzer,

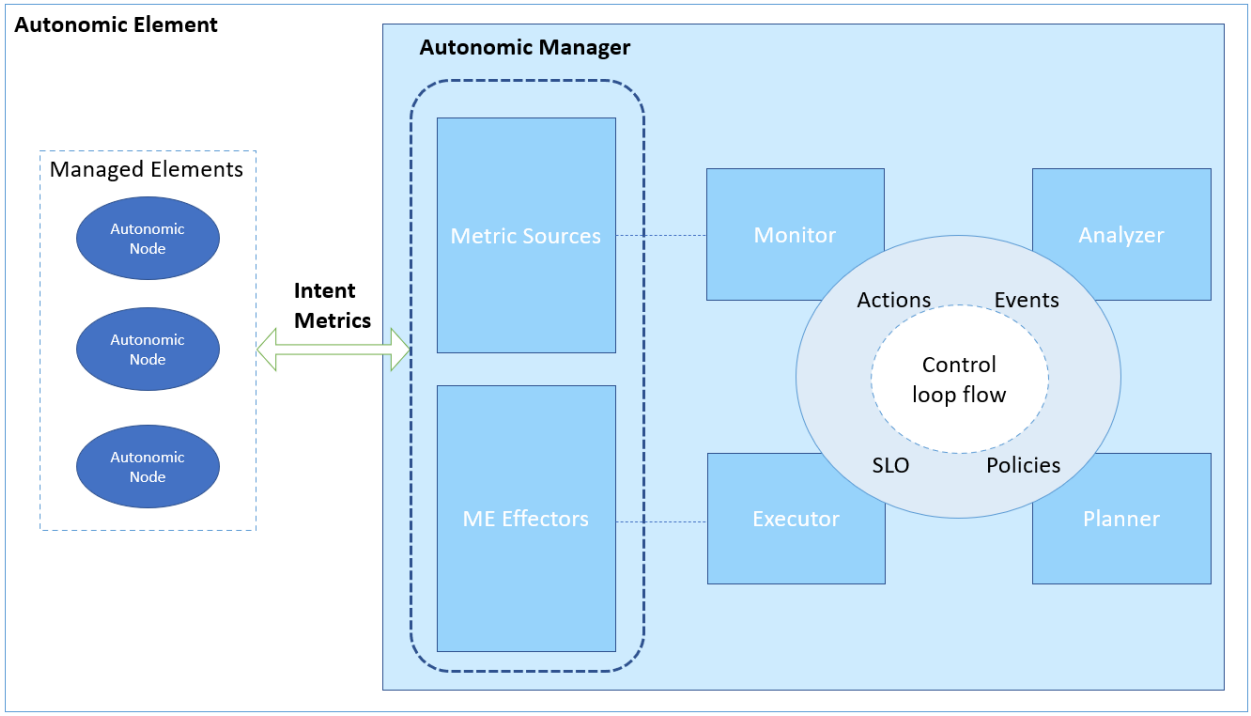

Figure 6: AE generic architecture and control loop

planner, and executor are the stages to form the functionality of the control loop. In our case, the monitor collects data from managed elements or networking nodes. The Planner provides the learning phase for the autonomic network through the arrangement of actions required for achieving the intent objectives. While the Analyzer determines whether the more complicated system policies and their impact on the system's environment are in place, the Executor ensures that the actions from the Planner stage are carried out per the models of the Planner. The basis for all the mentioned units is the knowledge gathered from the managed resources to extract the appropriate metrics to construct the intent to be measured for the performance of the autonomic network. The `AE` generic architecture for its control loop is illustrated in Figure 6. In the current version of ASSL, the decision-making logic for the control loop is constructed based on the control flow of different sequential events, from gathering information to performing the appropriate actions.

The main benefit of modeling with ASSL is that it is designed explicitly for modeling autonomic systems, allowing us to express the autonomic system's attributes and behaviors

at an abstract level. The abstract formulation is a well-known method to reduce complexity. To further explain, abstraction refers to hiding unnecessary features and encapsulating as many possible details to achieve a particular purpose, which can be completed with generalization and aggregation. To formulate intent abstractly, ASSL provides formal expressions representing the desirable network goals. `AS` tiers, `ASARCHITECTURE`, and fluents are sub-components that shape formal expressions. These intent specifications can be verified through the ASSL consistency check engine as they are formal methods. The concept of having tier-based architecture is ASSL's primary feature that facilitates the classification of components of an autonomic network in an abstract manner. For instance, some hosts are connected through links in an autonomic network. Each of these nodes and links between them as means of communication has its implementation details, which are diverse due to manufacturer-based attributes and optional to determine for different scenarios. An `AE` tier can encapsulate these node attributes and emphasize a specific metric, or `GUARD`, based on the intent requirements to define the conditions. Similarities between these tiers and sub-tiers depend on each other to form the integrated autonomic network. This abstract hierarchical design also allows the users to reuse a block of specification since each block can be considered a black box with its encapsulated intent-based characteristics. A group of related AEs organized as a unified abstraction is called generalization in ASSL `AECLASS`. The `AE` classes or abstract AEs can be utilized to generalize any similarities among the AEs in an AS. As a result, `AE` classes specify attributes, such as `AE` policies and behavior models, inherited by actual AEs under its autonomic system. This feature of ASSL brings the possibility of defining a template autonomic behavior specification for the autonomic nodes with similar characteristics, such as a group of hosts in a university laboratory, to inherit their behavioral models from one central autonomic node specification resulting in the simplification of an intent expression. Even though an `AS` is considered a unit in the intent expression procedure, this `AS` abstraction comprises one component ASIP, multiple component AEs, and an aggregate of both, including many sub-component objects.

The segregation of the AC (autonomic computing) features, such as self-management

properties, into manageable sub-domains, referred to as ASSL tiers and sub-tiers, forms the hierarchical abstraction and, therefore, less crucial details of the autonomic network, such as those regarding vendor-specific implementation details, can be concealed and focus the specification based on the high-level abstract intent. Thus, in general, the ASSL specification provides two levels of abstraction. The high-level abstractions form tiers in the ASSL hierarchy, while contextualized subdomains (sub-tiers), which comprise low-level system modules, are called low-level abstractions. Also, a managed resource is a distinct software system like networks, servers, or databases that provides services. At the same time, their functionality and architecture have not been emphasized by ASSL, unlike the autonomic computing properties. However, they are accessible with specified interfaces through an abstract version of the managed resource resulting in a low level of intrusion since the AC features are wrapped for previous configurations or systems on the network devices.

To conclude, the concrete architecture of the components in the autonomic network architecture is not addressed by ASSL. Instead, ASSL specification models an autonomic wrapper in the form of an autonomic element through a set of managed resource interface functions that can be designed to encapsulate the target network as a managed resource and grants it autonomic capabilities and intent specification objectives. In addition, regarding implementation, the framework creates the AEs as building blocks of the autonomic network simulation as logical collections of Java classes known as Java packages, which causes a direct relationship between the specified models and the actual system architecture. The comparisons between the implementation and the high-level ASSL standards result in abstraction at the implementation level. High-level, human-readable ASSL abstract constructs can also be translatable for machine language to be used in networking devices. The ASSL compiler can semantically verify all abstract autonomic behaviors specified by ASSL to ensure the reliability of the specifications for the autonomic network. This experimental phase should be completed by connecting ASSL specifications to software agents.

Like abstract formulation, ASSL has the means to define the declarative outcomes of

the intents. Unlike abstract formulation, the declarative outcome of intents should be determined with more details. Since this framework can demonstrate conceptual definitions of the specified intents that impact network state and these declarations can contain more detail through metrics, variables, and custom types in its semantics, ASSL is declarative. These metrics combine to create a complex variable construct that serves as the foundation for an ASSL tier instance that can alter model contexts after being received by `AS` operational model. As a result, ASSL investigates its tier clauses as micro semantic environments to build a declarative specification tree out of their internal rule definitions, which are descriptions of concepts rather than actual actions. The `ASSLTierToken` class in the Compiler module completes this tree of declarative tiers. Then, its result is used to estimate the model's consistency by assessing the relationships between any identifier and its declaration. Also, this assessment checks the accurate assignments of `AE` properties to prepare it for parsing by the ASSLParser class to create an intermediate code required to generate the skeleton of Java output without mentioning all the steps to achieve the specified intent in the ASSL model specification in the outcome. An example of an intent declarative outcome can be maintaining a particular bandwidth limitation. This example can be defined through ASSL constructs such as fluents that guide the autonomic network to actions, including numeric variables or metrics to express the bandwidth condition and sub-actions to complete the desired autonomic behavior. The self-management policies as a part of the ASSL specification model follow the state-transition machine paradigm, and fluents play the role of states or situations that an autonomic network node can accept. Specified `EVENTS` cause the transitions between these network states.

Portability can be assessed from two angles: the portability of ASSL specification language in case of execution of the language and the portability of intent specifications in terms of modeling and connecting those intents to the networking environment. ASSL is designed platform independent and needs a system that can run Java. Moreover, the ASSL toolset's environment for developing intent specifications contains a grammar compiler, consistency checker, and specification editor, which omits any dependency between the ASSL

and the underlying platform. Thus, in the case of execution, this framework is portable. Abstract ASSL constructs enable intent users to describe the desired behavior of a network, such as service level agreements (SLAs) and traffic engineering policies, independent of the underlying network infrastructure and technology. As a result, the intent can be applied to various vendors and platforms without modification. This portability is in terms of intent definition or modeling. To explain more, by considering the primary conditions of the intent at an abstract level, these conditions are the same for diverse vendors. For instance, if based on an intent, there is a bandwidth limitation of 50% for all video streaming traffic, the intent declaration of ASSL for all vendors is the same to keep this amount as the metric to measure for video streaming without considering the configuration of this policy. Thus, the modeling of the intent at the highest level of abstraction can be portable. On the other hand, communicating with devices and software agents is required to complete the intent application procedure. Managed resources such as networking devices are separated from ASSL in terms of the configuration since it is a simulation model of the autonomic problem. To connect the managed resources, such as network nodes that contain the low-level configuration, to ASSL specification, we can define the Java-like interface functions skeleton, which can return ASSL predefined types. This decoupling between the low-level devices and ASSL specifications outlines this language's potential portability and platform-agnostic characteristic. However, the portability of intent specification can be challenging since to interpret the ASSL specifications to different vendor-based configurations, there might be different interpretation requirements that might affect the intents. This potential challenge might be reduced though having a template set of interpretations to be used for different vendors. This means that the features of ASSL resulting in abstraction for intent specification minimize the modification of the core intent specification. Because instead of changing the core intent specification, we can adjust the software agents connecting to ASSL interface functions when there is a need to apply intents to devices from different vendors.

The hierarchical design of ASSL, consisting of different tiers, allows us to define the distributed and local autonomic network behaviors. Any autonomic behavior under the

`AS` tier is accessible for the entire autonomic network, and any desirable network goal defined under the `AE` tiers is localized for the AE. The distribution concept is covered by ASSL's specification, miming the network as an integrated autonomic system. However, after producing the Java code, all the packages and classes are included in a single file, making it inconvenient for dispersed execution in the current version of ASSL. Another aspect to discuss this objective is the network's central and distributed management. A centralized management system that is in charge of regulating the behavior of the system or network is referred to as distributed behavior management. The centralized management system is in charge of deciphering the ASSL specification and creating the unified configuration commands or actions connected to the system or network's components in a distributed behavior management system. Local control systems or agents that regulate the behavior of specific components or systems inside a system or network are referred to as local behavior managers. Each element or subsystem in a local behavior management system has its local control system or agent, which is in charge of deciphering the pertinent section of the ASSL specification and producing the necessary configuration commands or actions.

Starting at the low-abstract levels (supplied by the sub-tiers), where we provide functionally linked `AS` characteristics, is one method for developing autonomic networks through ASSL. As a result, ASSL permits us to initiate the structure of autonomic elements such as autonomic networking nodes by lower-level details, including metrics for the intent specification, and then add other functionalities or adjust the previous one to shape the precise intent specification. On the other hand, different self-management policies such as self-configuring, self-healing, and self-optimizing can be specified separately and then combined with working together with minimal modification in the core of the intent specification. Thus, ASSL features provide the potential of having composability for the intent specification at a functional and internal level. Nevertheless, it should be considered that there is a risk of composability failure if we have different ASSL specifications that aim to rule the same autonomic network. This can not be estimated in this thesis scope since the current version of ASSL, as the suggested solution, focuses on the simulation-only nature of

ASSL. However, it is undeniable that formal verification of ASSL constructs can ensure the autonomic networks that different policies can work together under the same domain if the specification is consistent.

ASSL is efficient due to different reasons. Since the framework is a formal specification language, it verifies the consistency of the intent expressions by its toolset. Therefore, the autonomic network can be ensured that the network can adapt itself based on the conditions specified in ASSL. The autonomic desired behavior expressed with ASSL is understandable in terms of language. Also, since this language encapsulates the complexities, the generated Java output is greater than the specified intents in the case of the number of lines of the code. Since the ASSL design provides modular specification code blocks that are reusable in different scenarios with minimal modifications, and due to its abstract level of specification, it is easier to maintain specification by this framework, which increases efficiency.

The scalable essence of the ASSL design hastens the `AS` development process and, consequently, the autonomic network for which ASSL specifies its intent. This is achievable because the ASSL architecture scales effectively, allowing for sub-system modularity and the capacity to grow the developing autonomic network by consistently including AEs (autonomous elements). Moreover, this scalability is possible under the internal blocks to add more metrics, as was described in the description of the *Composability* intent objective. To explicate more, it is possible to specify an autonomic node as an `AE` in ASSL with core concepts and then add other autonomic nodes to the system to increase the number of AEs with minimal disruption in the whole autonomic behavior structure. Also, a reverse proof would be decomposition as a strategy for abstraction when the autonomic network expression initiates by the high-level representation of the system and then develop low-level abstractions of the specified intent details, such as metrics to show how features and operations are synchronized. Nevertheless, the current ASSL architecture is static, and by adding components to the system, the code should be regenerated.

ASSL implements self-monitoring as a feature of the ASSL control loop algorithm for various metrics and SLOs, or other measurements to determine whether or not they adhere to

their threshold and limitations. For instance, it uses the `ASSLMONITOR` class as the foundation for all `AS` monitors. Another illustration is `AE ASSLMONITOR`, which keeps track of the classes in the `AE` control loop. The ASSL internal control loop begins with monitoring by determining whether each service-level objective (SLO) has been accomplished and whether each metric is accurate. Thus, any measurable construct or policy defined by ASSL can be monitored to estimate and verify if it works based on the specified conditions. Also, proactive external monitoring can be performed by having each autonomic node submit a notification message, including a brief demonstration of the node state regularly. Another external monitoring is applicable if the monitoring tool or software is considered a managed element working under the control of the autonomic network specification with ASSL. However, software agents should be developed for this purpose to complete the interaction of ASSL with the actual networking devices.

Autonomic networks should protect themselves against any external attacks, and security is a critical component of both software and hardware. Security in ANIMA's perspective is defined based on the domain of authority for network nodes authorized by a similar certificate [28]. The devices under these domains can contribute to each other as adjacent whose connections are secure through certificates and cryptographic methods. In general, ANIMA discusses security under two main categories, including creating a secure infrastructure through cryptography and making domain-based architecture to establish authorized domains and adjacency for a group of nodes [28]. The former is not covered by an intent specification language similar to ASSL since this security is mainly concerned with low-level devices or managed elements. However, regarding the latter, the abstract simulation of the security concepts to form an adjacency is achievable in ASSL. On the one hand, due to the grouping capability of ASSL centralized architecture, autonomic elements can become group members to simulate a domain structure. For this purpose, autonomic elements can form a group as an `AE` council. This can be referred to as a simulation of a central node that can include `AE` proxies generated from groups and councils. The `AE` proxy concept plays the role of a representative for each autonomic element from the group for

any required negotiation between the council and that autonomic element. They can be treated as normal AEs. The advantages of the proxy concept to the architecture should be summarized as the flexibility of the abstract design of the domain. However, coupled with a recovery protocol, they can enhance some security aspects since they can facilitate the gathering and analyzing of information related to malfunctioning of autonomic networks either as a result of network failure or malicious attacks.

Interaction Protocols (IP) can specify messages and communication functions under the `ASIP` or `AEIP` to connect these elements. ASSL interaction protocol features shape this abstract communication based on the required level of accessibility, which can be public for the whole `AS` or private for an AE. Also, reliable autonomic nodes can be listed under a static concept called `FRIENDS` in ASSL to form groups of adjacent nodes with the capability of private communication through private `CHANNELS` that are only accessible by the autonomic nodes under `FRIENDS` member lists. Therefore, the abstract domain-specific security between autonomic nodes can be simulated through ASSL group architecture and completed with private interaction protocols through private `CHANNELS`.

3.3 Concrete Intent Examples Expressed Using ASSL

To assess the expressiveness of the ASSL framework as a specification language for autonomic networks with the capability of representing intents abstractly, in this section, we demonstrate some intents based on the intent classification presented in RFC9316 [23].

The main key to using ASSL as a solution to intent-based autonomic networks is dividing the network architecture and the intent into ASSL components. This process facilitates extracting an intent model subsuming context, capabilities, and conditions. There are different components to model intent-driven autonomic networks with ASSL. The following four categories can be considered to formulate autonomic behavior in ASSL based on intent:

- *Goals and objectives*: The network's goals and objectives are used to describe the network's general purpose and desired behavior, such as minimizing the downtime of

a network.

- *Policies and rules*: Policies and rules are used to specify the specific activities the network must take in order to accomplish its goals and objectives. Resource allocation, security, and traffic routing policies are a few examples of policies.

- *Constraints and requirements*: Requirements and constraints are used to specify the requirements that must be met for the network to operate correctly. Limitations may include things like the maximum permitted delay, the minimum bandwidth needs, or the maximum permitted energy consumption.

- *Management functions*: The steps that the network should do to govern its operation are defined by management functions. Monitoring the state of the network, making configuration changes, or creating warnings are some examples of management tasks.

All the four mentioned categories can be shaped through a chain of fluents and events, with metrics as their core constraint, to form desirable self-management policies for the autonomic network. ASSL builds the intents as logical predicates that can change over time to implement intents. The logical predicates can form through fluents representing the status of a networking device or a particular element, such as the bandwidth utilization of a link. To maintain the autonomic network in the desired state, actions can perform the intent objectives. Applying configuration changes to network components and creating alarms when a problem is found are examples of these actions. An autonomic management framework or a software agent must connect these actions to the network to interpret them into commands. Messaging in ASSL can be used to request and transmit configuration commands or operations, as well as to transmit and request information about the status and behavior of the network. In the following sections, we provide concrete examples of networking intents, and we provide an implementation in ASSL to demonstrate the appropriateness of ASSL as a solution to express intents. The existing ASSL compiler does not provide any mechanism to access specific metrics on an individual network device. Thus, coding of the concept of accessing this metric on a managed element is specified as function calls that return a specific

value or values. This implementation strategy allows the representation of "access a metric" in a way that can be compiled using the existing ASSL compiler, which does not invalidate any of the examples presented.

3.3.1 INTENT 1

INTENT 1	
Intent :	*Always maintain a high quality of service and high bandwidth for gold-level subscribers.*
Solutions :	Carrier Networks
Intent User :	Customer/Subscriber
Intent Type :	Customer Service Intent
Source :	[23]

Table 1: Specification/classification for INTENT 1

The first network management intent we chose to represent is related to the quality of service (QoS) as one of the most important criteria for assessing the network's performance. The specifications/classification of this intent is presented in Table 1. From a conceptual perspective, there are five steps to achieve QoS implementation in the network: (1) *Traffic Classification*, (2) *QoS Labeling*, (3) *Policing*, (4) *Marking*, and (5) *Evaluating*. Network operators configure these steps of QoS with various vendor-based details, from categorizing access lists to grouping similar packets to mapping policies to each of these categories based on technical measurements in queuing interfaces. However, based on RFC 9316 [23], customers or subscribers who might not be aware of these technical details can be named as users of this intent. Thus, defining QoS goals at an abstract level can solve this issue. The other important note for choosing QoS as a strategy intent type to specify is its impact on boosting the network performance. It also can be beneficial in optimization and Service Level agreements (SLA) between network providers and their customers since the main goal for all these concepts is to improve the network performance. Network traffic is shaped based on IP packets that transmit various kinds of information, including video, voice, and data, over the network. Each of these traffic types can have a priority based on their

characteristics or, in particular cases, the network management requirements defined by network owners. In general, packets containing video should be prioritized since missing one packet can negatively affect the whole frame of video received at the destination device. With similar logic, voice over IP and data packets are the following priorities in the packet transmission procedure. By classifying traffic types, each group is labeled to be in the queue of the traffic transmission process based on their priority. Therefore, packet loss or other network quality issues will be reduced. Also, these classifications can be transferred based on policies created by network operators who determine the appropriate policy regarding the bandwidth or other relevant network performance measurements. Labeled traffic is marked after estimating if it behaves according to the defined policy, and those packets that do not respect the policies will be dropped. Finally, the whole process is evaluated to check the quality of service performance.

There should be an autonomic *controller* node to manage the autonomic computing of QoS requirements and a *customer* autonomic node to serve as the gold-level service subscriber. The QoS steps can be defined as self-configuring policies in the Controller node, where any other customer node can use the services based on their needs.

ASSL provides the means to simulate the QoS implementation steps under the context of autonomic networks. Therefore, without considering the exact details of the technical implementation, we can concentrate on intents outcomes to express them through formal declarative statements. The high-level abstraction of ASSL encapsulates the details of the procedures to achieve the intent. Regarding the general logic of this specification, `AS` and `AE` perspectives construct that the hierarchy of this specification consists of general and private rules. The former includes the self-configuring policy to configure the whole autonomic network using an `IMPL` action called `ConfigureAutonomicNetwork` as depicted in Figure 7. The latter includes network stages to simulate QoS specification under the `Controller` autonomic element that is only accessible for the autonomic elements, which are grouped with the `Controller` defined in the `ASARCHITECTURE`. This hierarchical view provides an abstract security in case of accessibility of policies for autonomic elements. Also, this design helps

```
AS AutonomicNetwork {
  TYPES{VidACL, VoiceACL, DataACL}
  VARS{string VidTag; string VoiceTag; string dataTag}
   ASSELF_MANAGEMENT {
      SELF_CONFIGURING {
         FLUENT inAutonomicNetworkConfiguration {
             INITIATED_BY { EVENTS.TimeToConfiguration }
             TERMINATED_BY {  EVENTS.ConfigurationDone } }
         MAPPING {
             CONDITIONS { inAutonomicNetworkConfiguration }
             DO_ACTIONS { ACTIONS.ConfigureAutonomicNetwork } }
               }//SELF_CONFIGURING
           } // ASSELF_MANAGEMENT

  ASARCHITECTURE {
    AELIST {AES.GoldLevelSubscriber, AES.Controller}
     DIRECT_DEPENDENCIES { DEPENDENCY AES.GoldLevelSubscriber { AES.Controller }}
     TRANSITIVE_DEPENDENCIES { DEPENDENCY AES.Controller {AES.GoldLevelSubscriber }}
   GROUPS {
     GROUP GoldLevel {
     MEMBERS { AES.GoldLevelSubscriber, AES.Controller }
      COUNCIL { AES.Controller }
     }
   }
 }//ASARCHITECTURE
  ACTIONS {
     ACTION IMPL Configuration {
        TRIGGERS { EVENTS.ConfigurationDone }
     }
     ACTION ConfigureAutonomicNetwork {
        GUARDS { ASSELF_MANAGEMENT.SELF_CONFIGURING.inAutonomicNetworkConfiguration }
        DOES { call IMPL ACTIONS.Configuration }
        ONERR_TRIGGERS { EVENTS.ConfigurationDenied }
     }
   } // ACTIONS

  EVENTS {
     EVENT TimeToConfiguration {ACTIVATION { PERIOD { 1 MIN } }DURATION { 500 MSEC}  }
     EVENT ConfigurationDone {  DURATION { 500 MSEC } }
     EVENT ConfigurationDenied {  DURATION { 500 MSEC } }
  }//EVENTS

}//AS
```

Figure 7: Controller `AS` in Intent 1.

the system to share distributed autonomic behavior under the `AS` level while allowing the customers to have their local autonomic behaviors under the `AE` level to meet the *Distributed and Local Behavior Management* intent objective. The ASSL architecture design of the intent is shown in Figure 7.

Two abstract links provide interaction between the autonomic elements and complete the logic flow: `public_link` for public messages shared among the whole autonomic network and

```
FRIENDS {   AELIST { AES.GoldLevelSubscriber }   }

AEIP{
     MESSAGES{
       MESSAGE msgQoSGold{
         SENDER { AES.Controller }
         RECEIVER { AES.GoldLevelSubscriber }
         MSG_TYPE { TASK }
       }
     }//MESSAGES

     CHANNELS{
       CHANNEL GoldLink{
          ACCEPTS{AEIP.MESSAGES.msgQoSGold,AES.GoldLevelSubscriber.AEIP.MESSAGES.msgBandWidthIncrease }
          ACCESS{DIRECT}  DIRECTION{INOUT}}
     }//CHANNELS

     FUNCTIONS{
       FUNCTION sendMsgQoSGold{
         DOES{ AES.Controller.AEIP.MESSAGES.msgQoSGold >> AES.Controller.AEIP.CHANNELS.GoldLink }
       }
       FUNCTION receiveBandWidthMsg{
         DOES{ AES.GoldLevelSubscriber.AEIP.MESSAGES.msgBandWidthIncrease << AES.Controller.AEIP.CHANNELS.GoldLink }
       }
     }//FUNCTIONS

     MANAGED_ELEMENTS {
       MANAGED_ELEMENT monitoringTool {
         // check header for the protocol number
         INTERFACE_FUNCTION checkHeader { RETURNS { DECIMAL } }
         INTERFACE_FUNCTION getCoS{
             PARAMETERS{INTEGER Tag }
             RETURNS{INTEGER}}
         INTERFACE_FUNCTION checkSource{RETURNS {DECIMAL}}
         INTERFACE_FUNCTION checkDestination{RETURNS{DECIMAL}}
         INTERFACE_FUNCTION DeviceAddress{RETURNS{DECIMAL}}
         INTERFACE_FUNCTION checkBandwidth{RETURNS{DECIMAL}}
         INTERFACE_FUNCTION monitorQoS{}
         INTERFACE_FUNCTION reportPerformanceRate{RETURNS{DECIMAL}}
         }
       }
     } // AEIP
```

Figure 8: Controller `AEIP` in Intent 1.

`Gold_link` for the private interaction through messages between the Controller and Gold-level subscriber. This private channel makes the interaction between the autonomic elements more secure since the private interactions are not accessible unless autonomic elements are defined as members of each other's `FRIEND` list group. The `AEIP`, including the messaging, channels, and managed elements, is illustrated in Figure 8.

To express Intent 1 with ASSL, the network's goals are divided into two main sub-goals, including maintaining a high quality of service and high bandwidth. The intent is specified into two steps. The first step starts with the autonomic network configuring itself according

to the `inAutonomicConfiguration` fluent specified under `AS` block. Then required phases to achieve QoS as a self-configuration policy are specified under the autonomic element called Controller, as the abstraction for the central management autonomic node. The specification of fluents guiding the autonomic element to required actions to attain QoS for the network traffic is shown in Figure 9. The self-configuring policy models traffic type identification through the `checkHeader` interfaces function of the `monitoringTool` that is recognized as the managed element as shown in Figure 8 working with the autonomic network and then classified as `VideoTag`, `VoiceTag`, and `DataTag` as depicted in Figure 21. These taggings are the conditions to create access lists for grouping packets to prepare them for QoS labeling.

The Intent 1 is specified in two steps to show the potential of ASSL to provide scalability at an abstract level. In the first step, only the `Controller` and the `AS` tier are defined and executed. Then the expression of the gold-level subscriber is added as another `AE` and executed. In both cases, the execution trace of autonomic behavior shows that all the specified parts of the autonomic behavior work; thus, adding the second `AE` does not disturb the functionality of the whole structure of the autonomic network. The first lines of the output trace are shown in Figure 11 where there is a gold-level subscriber and in Figure 10 where besides gold-level subscriber trace of specifications such as fluents is started from line 78 besides the previous trace of the Controller.

Figure 12, and Figure 13 illustrate the generated packages of the specified intent for the `AS` tier before, and after addition of the subscriber's AE. The modification of the `AS` package for this addition is the creation of `ASARCHITECTURE` to form a grouping between the Controller and Gold-level subscriber AEs to have access to the messaging interaction through the AEIP.

Before abstractly scaling the autonomic network, AEs tier generates the packaging for the Controller only as seen in Figure 14, and after this abstract scaling, the packages for the Controller and Gold-level subscriber are generated as shown in Figure 13, Figure 15, and Figure 16. The modification that are caused by adding the specification for a new `AE` are marked with red shapes.

```
AE Controller{
  AESELF_MANAGEMENT {
    SELF_CONFIGURING {
      FLUENT inIdentifyTrafficType{
            INITIATED_BY { EVENTS.IPPacketReceived }
            TERMINATED_BY { EVENTS.TrafficIdentified } }
      FLUENT InClassification{
            INITIATED_BY { EVENTS.TrafficReceived }
           TERMINATED_BY { EVENTS.ClassificationDone} }
      FLUENT InLabelingQoS {
            INITIATED_BY { EVENTS.TimeToLabel }
            TERMINATED_BY { EVENTS.QoSLabeled }  }
      FLUENT InPolicing{
            INITIATED_BY { EVENTS.TimeToPolicing }
            TERMINATED_BY { EVENTS.PolicingDone } }
      FLUENT InMarking{
           INITIATED_BY { EVENTS.TimeToMarking }
            TERMINATED_BY { EVENTS.MarkingDone }  }
      FLUENT InEvaluation{
            INITIATED_BY { EVENTS.TimeToEvaluatingQoS }
            TERMINATED_BY { EVENTS.EvaluationDone }   }
      FLUENT inQoSGoldSetting{
           INITIATED_BY{EVENTS.EvaluationDone}
            TERMINATED_BY{EVENTS.QoSGoldSet}}
      MAPPING {
            CONDITIONS {inIdentifyTrafficType }
            DO_ACTIONS { ACTIONS.IdentifyTraffic}   }
      MAPPING {
           CONDITIONS { InClassification}
           DO_ACTIONS { ACTIONS.classifyPackets}  }
      MAPPING {
            CONDITIONS {InLabelingQoS }
             DO_ACTIONS { ACTIONS.LabelTraffic}   }
      MAPPING {
            CONDITIONS {InPolicing}
             DO_ACTIONS { ACTIONS.policing}  }
      MAPPING {
            CONDITIONS {InMarking }
            DO_ACTIONS { ACTIONS.MarkTraffic}   }
      MAPPING {
           CONDITIONS {InEvaluation }
          DO_ACTIONS { ACTIONS.EvaluateQoS} }
      MAPPING {
         CONDITIONS {inQoSGoldSetting}
         DO_ACTIONS{ACTIONS.setQoSGold}}
          }//SELF_CONFIGURING
```

Figure 9: Self-configuring in Intent 1.

```
50 ************************************************************
51 ***************** AS STARTED SUCCESSFULLY ****************
52 ************************************************************
53 EVENT 'generatedbyassl.as.autonomicnetwork.events.TIMETOCONFIGURATION': has
   occurred
54 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INAUTONOMICNETWORKCONFIGURATION': has been initiated
55 ACTION 'generatedbyassl.as.autonomicnetwork.actions.CONFIGURATION': has been
   performed
56 EVENT 'generatedbyassl.as.autonomicnetwork.events.CONFIGURATIONDONE': has occurred
57 ACTION 'generatedbyassl.as.autonomicnetwork.actions.CONFIGUREAUTONOMICNETWORK': has
    been performed
58 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INAUTONOMICNETWORKCONFIGURATION': has been terminated
59 EVENT 'generatedbyassl.as.aes.controller.events.IPPACKETRECEIVED': has occurred
60 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_configuring.
   INIDENTIFYTRAFFICTYPE': has been initiated
61 ACTION 'generatedbyassl.as.aes.controller.actions.IDENTIFYTRAFFIC': has been
   performed
62 EVENT 'generatedbyassl.as.aes.controller.events.TRAFFICIDENTIFIED': has occurred
63 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_configuring.
   INIDENTIFYTRAFFICTYPE': has been terminated
64 EVENT 'generatedbyassl.as.aes.controller.events.TRAFFICRECEIVED': has occurred
65 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_configuring.
   INCLASSIFICATION': has been initiated
66 ACTION 'generatedbyassl.as.aes.controller.actions.CLASSIFYPACKETS': has been
   performed
```

Figure 10: Controller output trace for Intent 1.

```
67 ************************************************************
68 ***************** AS STARTED SUCCESSFULLY ****************
69 ************************************************************
70 EVENT 'generatedbyassl.as.autonomicnetwork.events.TIMETOCONFIGURATION': has
   occurred
71 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INAUTONOMICNETWORKCONFIGURATION': has been initiated
72 ACTION 'generatedbyassl.as.autonomicnetwork.actions.CONFIGURATION': has been
   performed
73 EVENT 'generatedbyassl.as.autonomicnetwork.events.CONFIGURATIONDONE': has occurred
74 ACTION 'generatedbyassl.as.autonomicnetwork.actions.CONFIGUREAUTONOMICNETWORK': has
    been performed
75 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INAUTONOMICNETWORKCONFIGURATION': has been terminated
76 EVENT 'generatedbyassl.as.aes.controller.events.IPPACKETRECEIVED': has occurred
77 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_configuring.
   INIDENTIFYTRAFFICTYPE': has been initiated
78 EVENT 'generatedbyassl.as.aes.goldlevelsubscriber.events.TIMETOINITIATEQOS': has
   occurred
79 FLUENT 'generatedbyassl.as.aes.goldlevelsubscriber.aeself_management.qosgold.
   INQOSGOLD': has been initiated
80 ACTION 'generatedbyassl.as.aes.goldlevelsubscriber.actions.QOSGOLD': has been
   performed
81 EVENT 'generatedbyassl.as.aes.goldlevelsubscriber.events.QOSINITIATED': has
   occurred
82 ACTION 'generatedbyassl.as.aes.controller.actions.IDENTIFYTRAFFIC': has been
   performed
```

Figure 11: Controller and gold-level subscriber output traces for Intent 1.

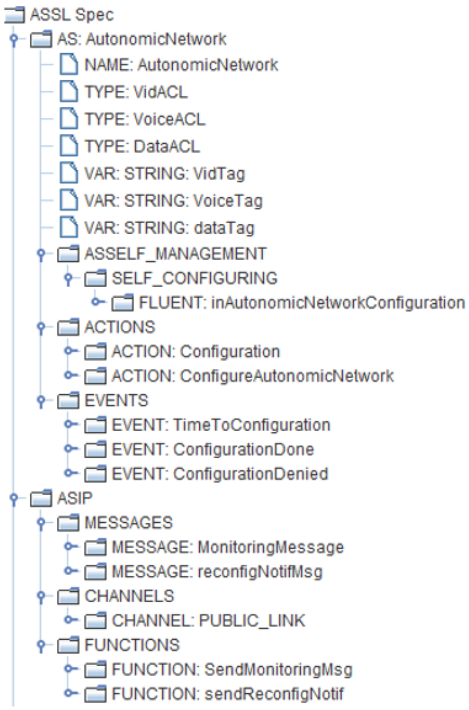

Figure 12: `AS` generated package before adding the gold-level subscriber in Intent 1.

QoS steps are specified, besides considering high bandwidth measurement as the metric value to be set in the Controller AE. During the `InPolicing` fluent, the autonomic network does the `policing` action as depicted in Figure 17 where the bandwidth metric value is set at an abstraction level, and expressing source, and destination IP address checkers as interface functions such as `checkSource` are called to differentiate the ingress traffic from the egress traffic. The `bandwidthPolicer` metric defined in the previous state is used in `MarkTraffic` caused by `inMarking` fluent for the simulation of packet marking. If packets respect the metric, the `PassPacket` IMPL action is called; otherwise, the `DropPacket` IMPL action provides the abstraction to drop the erroneous packets. `MarkTraffic`, and other related actions are depicted in Figure 18.

After this state, `InEvaluation` fluent occurs to help the system self-monitor its functionality at an abstract level, as shown in Figure 19. The performance rate can be reported through the `reportPerformanceRate` interface function that connects the system

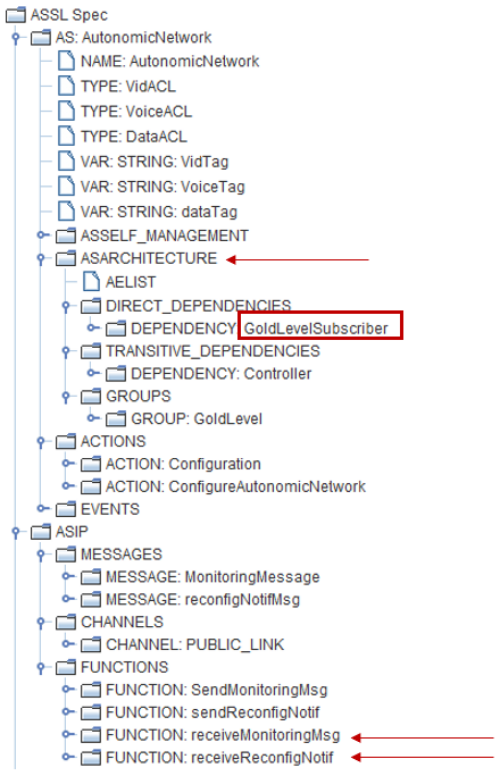

Figure 13: `AS` generated package after adding the gold-level subscriber in Intent 1.

to the monitoring tool managed element. This is designed to show the expressiveness of ASSL in meeting the *Monitoring* intent objective. However, the simulation-only nature of the current version of ASSL does not allow direct reading of the performance rate from the monitoring tool, which results in entering the performance rate manually. Nevertheless, since this modeling of monitoring is consistent after formalized checking of the specification, the metrics and the monitoring capability is valid.

The self-healing policy depicted in the `Controller` in Figure 20 helps the autonomic network to reconfigure itself as a solution if the quality of service for the Gold-level does not have a high performance. While two self-management policies, such as self-configuring and self-healing, can work individually but under one autonomic element, the potential capability of ASSL to support the *Composability* objective is outlined.

The abstract formulation of Intent 1 is met with network states, including no details about metrics, precise details, or sub-steps to perform an action. An example is shown

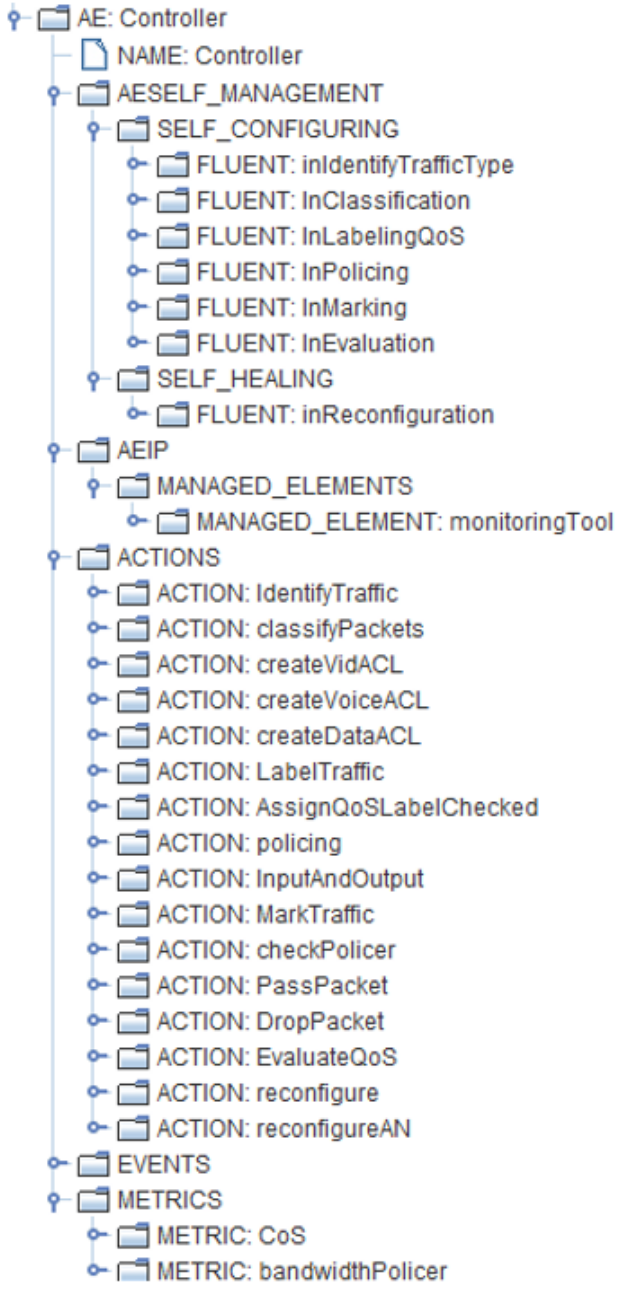

Figure 14: `AE` generated package before adding the gold-level subscriber in Intent 1.

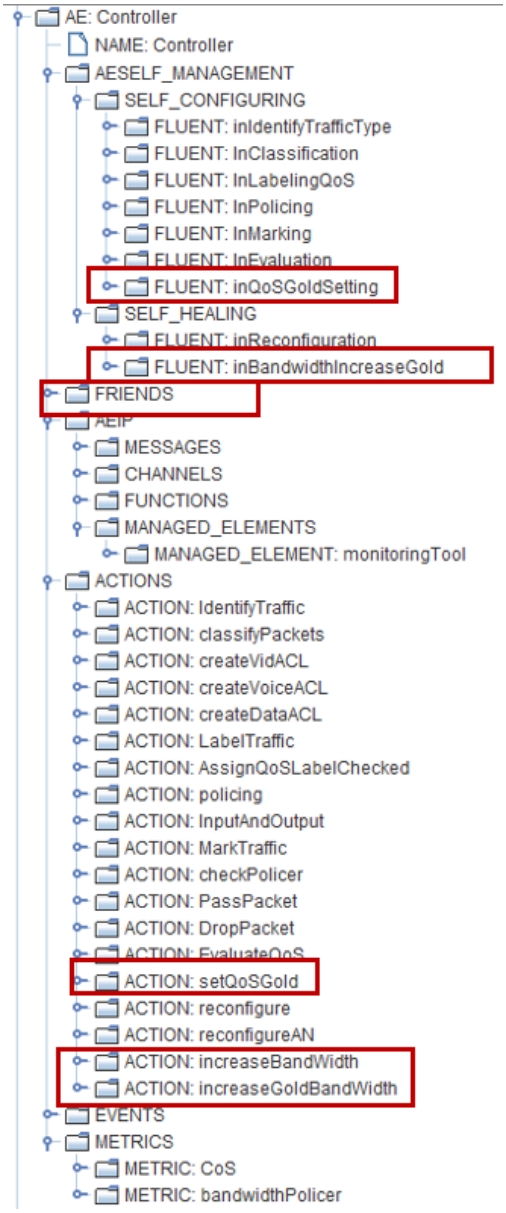

Figure 15: `AE` gold-level subscriber added to the controller package in Intent 1.

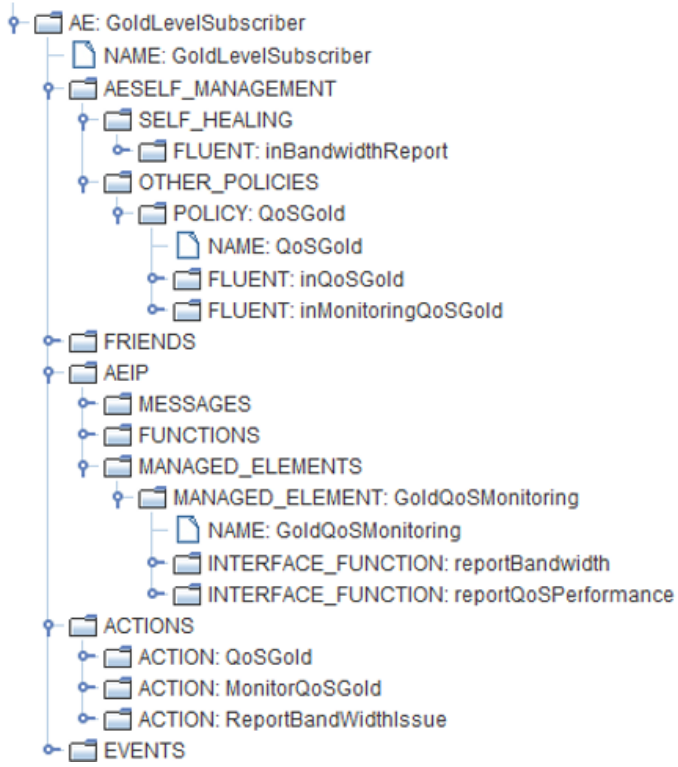

Figure 16: `AE` gold-level subscriber package added in Intent 1.

```
ACTION policing{
  GUARDS{AESELF_MANAGEMENT.SELF_CONFIGURING.InPolicing}
  DOES{
    SET METRICS.bandwidthPolicer.value=5;
    CALL AEIP.MANAGED_ELEMENTS.monitoringTool.checkSource;
    CALL AEIP.MANAGED_ELEMENTS.monitoringTool.checkDestination;
    CALL AEIP.MANAGED_ELEMENTS.monitoringTool.DeviceAddress;
    CALL IMPL ACTIONS.InputAndOutput
  }
  TRIGGERS{EVENTS.PolicingDone}
}

ACTION IMPL InputAndOutput{
  GUARDS{AESELF_MANAGEMENT.SELF_CONFIGURING.InPolicing}
  TRIGGERS{EVENTS.InputAndOutputIdentified}
}
```

Figure 17: Policing process specification in Intent 1.

```
ACTION MarkTraffic{
  GUARDS{AESELF_MANAGEMENT.SELF_CONFIGURING.InMarking}
  VARS{BOOLEAN policyRespected}
  DOES{
    CALL AEIP.MANAGED_ELEMENTS.monitoringTool.checkBandwidth;
    CALL IMPL ACTIONS.checkPolicer;

    IF policyRespected THEN CALL IMPL ACTIONS.PassPacket END
    ELSE CALL IMPL ACTIONS.DropPacket
    END
    }
  TRIGGERS{EVENTS.MarkingDone}
}

ACTION IMPL checkPolicer{
  GUARDS{METRICS.bandwidthPolicer}

  ENSURES{EVENTS.policyChecked}
}

ACTION IMPL PassPacket{
  ENSURES{EVENTS.passed}
}

ACTION IMPL DropPacket{
  GUARDS{NOT METRICS.bandwidthPolicer}
  ENSURES{EVENTS.dropped}
}
```

Figure 18: Marking process specification in Intent 1.

```
ACTION EvaluateQoS{
  GUARDS{AESELF_MANAGEMENT.SELF_CONFIGURING.InEvaluation}
  VARS{DECIMAL performanceRate}
  DOES{
    CALL AEIP.MANAGED_ELEMENTS.monitoringTool.monitorQoS;
    CALL ASIP.FUNCTIONS.SendMonitoringMsg;
    performanceRate=CALL AEIP.MANAGED_ELEMENTS.monitoringTool.reportPerformanceRate;
    CALL ASIP.FUNCTIONS.receiveMonitoringMsg

  }
  TRIGGERS{EVENTS.EvaluationDone}
}
```

Figure 19: Evaluation process specification in Intent 1.

```
SELF_HEALING{
  FLUENT inReconfiguration{
      INITIATED_BY{EVENTS.timeToReconfigure}
      TERMINATED_BY{EVENTS.reconfigurationDone} }
  FLUENT inBandwidthIncreaseGold{
      INITIATED_BY{EVENTS.TimeToIncreaseBandwidth}
      TERMINATED_BY{EVENTS.BandwidthIncreasedGold}}

  MAPPING{
      CONDITIONS{inReconfiguration}
      DO_ACTIONS{ACTIONS.reconfigure} }
  MAPPING{
      CONDITIONS{inBandwidthIncreaseGold}
      DO_ACTIONS{ACTIONS.increaseBandWidth}}
       }//SELF_HEALING
```

Figure 20: Self-healing policy in Intent 1.

in Figure 17 where the input and output traffic to the autonomic node are distinguished; however, the details of the procedure are not mentioned since this is an `IMPL` action, where details of it are computed out of the scope of the ASSL specification level.

The *Declarative Outcome Expression* intent objective is met in several parts of the autonomic behavior, as QoS intensively depends on various metrics. Some examples can be seen in Figure 21, which depicts the actions where the packets are identified and classified. Each of these actions includes parameters to categorize protocol numbers or set metric values called `CoS` referring to the class of service metric in QoS to classify groups of packets based on their tags. Due to the simulation-only essence of ASSL, the metrics were set manually. Metrics that are useful to maintain the high quality and high bandwidth to demonstrate the expressiveness of ASSL for the *Declarative Outcome Expression* intent objective are depicted in Figure 22. The bandwidth threshold is defined between 5 and 10. Suppose the metric's value is less than this threshold, like what is declared as an example in Figure 22 as 3. In that case, it results in bandwidth metric violation guiding the autonomic network to perform `increaseBandWidth` action to set the metric value to a number within the threshold range to maintain the high bandwidth as depicted in Figure 23.

```
ACTION IdentifyTraffic{
  PARAMETERS { string Tag; DECIMAL ProtocolNum }
  GUARDS { AESELF_MANAGEMENT.SELF_CONFIGURING.inIdentifyTrafficType }
  DOES { CALL AEIP.MANAGED_ELEMENTS.monitoringTool.checkHeader;
         // this number is for TCP protocol number which is used by video traffic
         IF ProtocolNum=6 THEN Tag="vid"; AS.VidTag="VidTag"
         END
         ELSE
         // this number is for UDP protocol number which is used by voice traffic
         IF ProtocolNum=17 THEN Tag="voice"; AS.VoiceTag="VoiceTag"
         END
         ELSE
         // this number is for RDP protocol number which is used by data traffic
         IF ProtocolNum=27 THEN Tag="data"; AS.dataTag="dataTag"
         END
         END
         END
          }
  TRIGGERS { EVENTS.TrafficIdentified}}

ACTION classifyPackets {
  GUARDS {AESELF_MANAGEMENT.SELF_CONFIGURING.InClassification }
  DOES {
         IF AS.VidTag= "VidTag" THEN CALL IMPL ACTIONS.createVidACL;
         SET METRICS.CoS.VALUE=4
         END
         ELSE
         IF AS.VoiceTag= "VoiceTag" THEN CALL IMPL ACTIONS.createVoiceACL;
         SET METRICS.CoS.VALUE=5
         END
         ELSE
         IF AS.dataTag= "dataTag" THEN CALL IMPL ACTIONS.createDataACL;
         SET METRICS.CoS.VALUE=1
         END
         END
         END }
  TRIGGERS {EVENTS.ClassificationDone}
  ONERR_TRIGGERS{EVENTS.defaultOption} //in case it is not possible to classify the traffic, default mode is activated
}
```

Figure 21: Identification and classification procedures in Intent 1.

3.3.2 INTENT 2

INTENT 2	
Intent :	*Request automatic rapid detection of device failures and pre-alarm correlation*
Solutions :	Carrier Networks, DC Networks
Intent User :	Underlay Network Administrator
Intent Type :	Operational Task Intent
Source :	[23]

Table 2: Specification/classification for INTENT 2

In networking, device failure and quick detection are crucial for various factors, including

```
METRICS {
      METRIC CoS {//class of service metric
            METRIC_TYPE { RESOURCE }
            METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.monitoringTool.getCoS}
            DESCRIPTION { "class of service field" }
            MEASURE_UNIT { "" }
            VALUE { }
            THRESHOLD_CLASS { INTEGER [0~7] }
                }

      METRIC bandwidthPolicer {
            METRIC_TYPE { RESOURCE }
            METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.monitoringTool.checkBandwidth}
            DESCRIPTION { "bandwidth" }
            MEASURE_UNIT { "Mbps" }
            VALUE { 3 }
            THRESHOLD_CLASS { DECIMAL [5~10] }
          }

            }//METRICS
```

Figure 22: Network metrics in Intent 1.

```
ACTION increaseBandWidth{
  GUARDS{AESELF_MANAGEMENT.SELF_HEALING.inBandwidthIncreaseGold}
  DOES{CALL IMPL ACTIONS.increaseGoldBandWidth;
       SET METRICS.bandwidthPolicer.VALUE=10}
}

ACTION IMPL increaseGoldBandWidth{
  PARAMETERS{string SubscriberName}
  TRIGGERS{IF SubscriberName="GoldLevelSubscriberAE" THEN EVENTS.BandwidthIncreasedGold END}
}

}//ACTIONS
```

Figure 23: Action to increase bandwidth in Intent 1.

cost, performance, and reliability. We have selected an intent falling into this category, whose specification/classification is depicted in Table 2. Network reliability is crucial for users and applications to access the resources and services they require. Rapid device failure detection and alert generation allow the network to respond quickly to problems and resume functioning. Device malfunctions can harm network performance because they obstruct traffic flow and result in delays or errors. This detrimental effect can be reduced, and users and programs can experience failures as little as feasible. It is possible to reduce the price of equipment repair and replacement by immediately identifying problems and sounding alerts. Creating autonomous alarms for device failures can enhance network performance in several ways, including quicker detection, reaction, proactive maintenance, and increased reliability. The increase in speed in procedures is due to faster failure pattern identification.

Some actors and roles could be included in a system for automatic rapid detection of device failures and pre-alarm correlation, including network devices, alarm generation processes, and monitoring tools. The first one is equivalent to autonomic elements, the second one can be formed through states and functionalities of the network, and the third one refers to managed resources of the autonomic network.

The ASSL structure to express this scenario contains a self-healing policy to simulate the procedures, a `Controller` autonomic element as the coordinator, and a host as a user device. Self-healing policy is about overcoming network failures, such as any crash caused by an external source. Each device regularly communicates with the coordinating device, and notifications are sent to the coordinator. The latter can use these notifications to ascertain when a device can no longer function owing to a crash or equipment issue. Thus, a device encounters an issue if the coordinating device does not receive the regular notification message `GeneralNotifMsg` and receives the broken device message `ToolProblemMsg`. Both messages are depicted in Figure 24. Transmission of these messages form a notification system for the autonomic network.

An ASSL specification provides the self-healing behavior of devices at the global and individual levels. For instance, for the dependent devices, we can consider if the device is

```
MESSAGES {
   FINAL MESSAGE ToolProblemMsg  {
      SENDER  { AES.host  }
      RECEIVER  { AES.Controller  }
      MSG_TYPE  { TEXT }
      BODY { "Tool Problem" }
   }
   FINAL MESSAGE GeneralNotifMsg  {
      SENDER  { AES.host }
      RECEIVER  { AES.Controller }
      MSG_TYPE  { TEXT }
      BODY  { "device is working" }
   }
}
CHANNELS {
   CHANNEL AEs_Link  {
      ACCEPTS  { AEIP.MESSAGES.ToolProblemMsg, AEIP.MESSAGES.GeneralNotifMsg }
      ACCESS  { SEQUENTIAL  }
      DIRECTION  { INOUT }
   }
```

Figure 24: Messages from host `AE` for Intent 2.

lost through a crash or shut down to be omitted from the network. Therefore, we can express the failure network states through fluents initiated by the `CrashHappen` event, terminated by the `ToolChecked` event to map it to `checkDevice` an `IMPL` action to assess the functionality of the device. The `crashHappen` event is initiated in case of threshold violation and modification in this metric `NetworkMetric` by showing less amount of bandwidth accepted in the node specified as in `GUARDS` in this event. As seen in Figure 25, the link bandwidth metric can have a threshold from 5 to an infinite number, and if the value of the `NetworkMetric` is chosen as 5. The network metric is violated if the value is less than 5, which is selected as the minimum bandwidth rate. The bandwidth violation shows that a crash occurred in the network. This logic also is an example of *Declarative Outcome Expression* intent objective.

The procedure starts with a failure in the autonomic node and conducts the autonomic element to check if the device's software is functional. This fluent, shown in Figure 26, checks whether the coordinating device is crashed. Figure 27, and Figure 28 show mappings and actions for `InCrashed` fluent, respectively. This expression also needs an interface function to connect the metric source with the dependent managed elements.

The other fluent is `inGeneralNotif` activated by the `timeTosendGeneralNotif` event,

```
METRICS {
  METRIC NetworkMetric  {
    METRIC_TYPE  {  RESOURCE  }
    METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.Host.GetNetworkMetric }
    DESCRIPTION  { "measure the network metric to ensure for device activity" }
    MEASURE_UNIT  { "Mbps" }//This should be chosen based on the measured metric
    VALUE { 5 }      //This should be chosen based on the measured metric
    THRESHOLD_CLASS  {  DECIMAL  [5 ~ )  }   //This should be chosen based on the measured metric
  }
}
```

Figure 25: Network metric referring to bandwidth measurement for Intent 2.

```
FLUENT InCrashed {
  INITIATED_BY  {  EVENTS.CrashHappen  }
  TERMINATED_BY  {  EVENTS.ToolChecked  }
}
```

Figure 26: Fluent that monitors a network device failure for Intent 2.

```
MAPPING { // if a device is crashed then check if the Tool on it is still operational
  CONDITIONS  {  InCrashed  }
  DO_ACTIONS  {  ACTIONS.CheckANTool  }
}
```

Figure 27: Mapping a fluent to the proper action when a device fails in Intent 2.

```
ACTION IMPL CheckTool {
  RETURNS {  BOOLEAN  }
  TRIGGERS {  EVENTS.ToolChecked }
}
ACTION CheckANTool {
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InCrashed }
  VARS { BOOLEAN canOperate }
  DOES { canOperate = false; CALL ACTIONS.CheckTool }
  TRIGGERS { IF (not canOperate) THEN EVENTS.ToolProblem END }
}
```

Figure 28: Actions of the autonomic network when a device failure occurs in Intent 2.

```
FLUENT InGeneralNotif  {
   INITIATED_BY  {  EVENTS.TimeToSendGeneralNotif }
   TERMINATED_BY  {  EVENTS.IsMsgGeneralNotifSent }
}
```

Figure 29: Fluent that manages the crash notification in Intent 2.

```
ACTION NotifyForGeneralNotif {
   GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InGeneralNotif }
   DOES { CALL AEIP.FUNCTIONS.SendGeneralNotifMsg }
}
```

Figure 30: Action to send a notification as the alarm in Intent 2.

which is a time-periodic loop and eliminated after the message is received by a coordinating device by the event called `isMsgGeneralNotifSent`. The fluent should be mapped to `NotifyForGeneralNotif` action that uses `sendGeneralNotifMsg` as the `AEIP` Function to transmit `generalNotifMsg` through the `AEs_link` channel as shown in Figure 24, which is an abstract link between the autonomic elements. Figure 28 and Figure 29 show the fluent and action related to the general notification to the system to check if autonomic elements are functional.

The fluents in the implementation illustrate how network states are encapsulated based on the abstract formulation to hide any intricate complexity of machine language, such as network device configuration commands. The fluents are the state machines to conduct the autonomic network to react correspondingly. For instance, since the intent focuses on alarm generation, if any device failure happens because of reasons such as metric violation, there are actions like `notifyManagement` defined as the response of the autonomic network to the failure state which illustrated under self-healing policy in Figure 31. Self-monitoring capability of the metric in ASSL helps the intent become more adequate by rapidly detecting any faults.

ASSL can monitor the metrics, although it is rare to read the metric value from actual managed elements directly because of the simulation-only essence of it. However, the conditions under which a metric should be measured or collected and the actions conducted in response to the metric's value can be specified and monitored using ASSL. This

```
AS AutonomicNetwork {
   ASSELF_MANAGEMENT {
      SELF_HEALING {
         FLUENT inCrashingDevice {
            INITIATED_BY { EVENTS.deviceCrashed }
            TERMINATED_BY { EVENTS.managementNotified }
         }
         MAPPING {
            CONDITIONS { inCrashingDevice }
            DO_ACTIONS { ACTIONS.notifyManagement }
         }
      }
   } // ASSELF_MANAGEMENT
```

Figure 31: `AS` self-healing policy in Intent 2.

internal monitoring capability of ASSL to set metric values similar to the implementation of `NetworkMetric` allows the system to estimate if the autonomic network works based on the set values for the metric to take proper actions in case of violation of the values. As an illustration, an ASSL specification might provide that a specific metric's value should be periodically measured for a specific time duration and that if the measurement goes above a specified limit, a warning should be created, or remedial action should be taken. Alternatively, it is necessary to employ some monitoring or management tool to gather and process the metric data to read the value of a metric from controlled items. This might be a monitoring tool that operates independently or a part of an autonomic management framework that controls how the managed elements function. This refers to software agents, management policy servers, or consoles that can be developed to interpret ASSL to appropriate commands. The intent objective that allows the autonomic network to have distributed local and global autonomic behavior occurs when there is a hierarchy to implement the general perspective of the autonomic network like `inCrashingDevice` for the `AS` tier to make it accessible for the entire autonomic network. Also, the `ASIP` tier is considered for public messaging through an abstract link simulation called `Public_Link`, as shown in Figure 32.

On the other hand, the states of crashes in each device are evaluated individually under the `AE` tiers. For example, if a host software encounters a problem, the `InToolProblem` fluent

```
ASIP {
   MESSAGES {
      MESSAGE msgdeviceCrashed {
         SENDER { ANY }
         RECEIVER { ANY }
         PRIORITY { 1 }
         MSG_TYPE {TEXT }
         BODY { "Crashed Device" }
      }
   }
   CHANNELS {
      CHANNEL Public_Link {
         ACCEPTS { ASIP.MESSAGES.msgdeviceCrashed }
         ACCESS { SEQUENTIAL }
         DIRECTION { INOUT } }
   }
   FUNCTIONS {
      FUNCTION senddeviceCrashedMsg {
         DOES { ASIP.MESSAGES.msgdeviceCrashed >> ASIP.CHANNELS.Public_Link }
      }
   }
}
```

Figure 32: `ASIP` tier for Intent 2.

```
FLUENT InToolProblem  {
   INITIATED_BY  {  EVENTS.ToolProblem  }
   TERMINATED_BY  {  EVENTS.IsMsgToolProblemSent  }
}
```

Figure 33: Failure state for a tool or software in an `AE` in Intent 2.

is initiated shown in Figure 33.

Also, the messages related to the software problem of networking autonomic elements like `ToolProblemMsg` are not accessible for all the autonomic elements unless they are defined in the `FRIEND` list of the Host. Figure 34 shows the local messaging accessible for the host's `FRIEND` list with access to `AEs_link`. If the `FRIEND` list is not declared, the specification compiling encounters an error for accessibility as depicted in Figure 35.

This separation of the local and global autonomic behaviors and limitations in the accessibility of different autonomic elements results in a secure interaction at an abstract level. The execution of this specification is portable since it can be executed on any device regardless of the system on that device capable of running Java.

```
FRIENDS {
   AELIST { AES.Controller }
}

AEIP {
   MESSAGES {
      FINAL MESSAGE ToolProblemMsg  {
         SENDER  { AES.host  }
         RECEIVER  { AES.Controller  }
         MSG_TYPE  { TEXT }
         BODY { "Tool Problem" }
      }
      FINAL MESSAGE GeneralNotifMsg  {
         SENDER  { AES.host }
         RECEIVER  { AES.Controller }
         MSG_TYPE  { TEXT }
         BODY  { "device is working" }
      }
   }
   CHANNELS {
      CHANNEL AEs_Link  {
         ACCEPTS  { AEIP.MESSAGES.ToolProblemMsg, AEIP.MESSAGES.GeneralNotifMsg }
         ACCESS  { SEQUENTIAL  }
         DIRECTION  { INOUT }
      }
   }
   FUNCTIONS {
      FUNCTION SendToolProblemMsg  {
         DOES {  AEIP.MESSAGES.ToolProblemMsg  >>  AEIP.CHANNELS.AEs_Link  }
      }
      FUNCTION SendGeneralNotifMsg  {
         DOES {  AEIP.MESSAGES.GeneralNotifMsg  >>  AEIP.CHANNELS.AEs_Link }
      }
   }
   MANAGED_ELEMENTS {
      MANAGED_ELEMENT Host {
         INTERFACE_FUNCTION GetNetworkMetric {  RETURNS { DECIMAL }  }
      } //this metric is measured based on the monitoring Tool which are assigned to the hosts
   }
} // AEIP
```

Figure 34: `AEIP` in Intent 2.

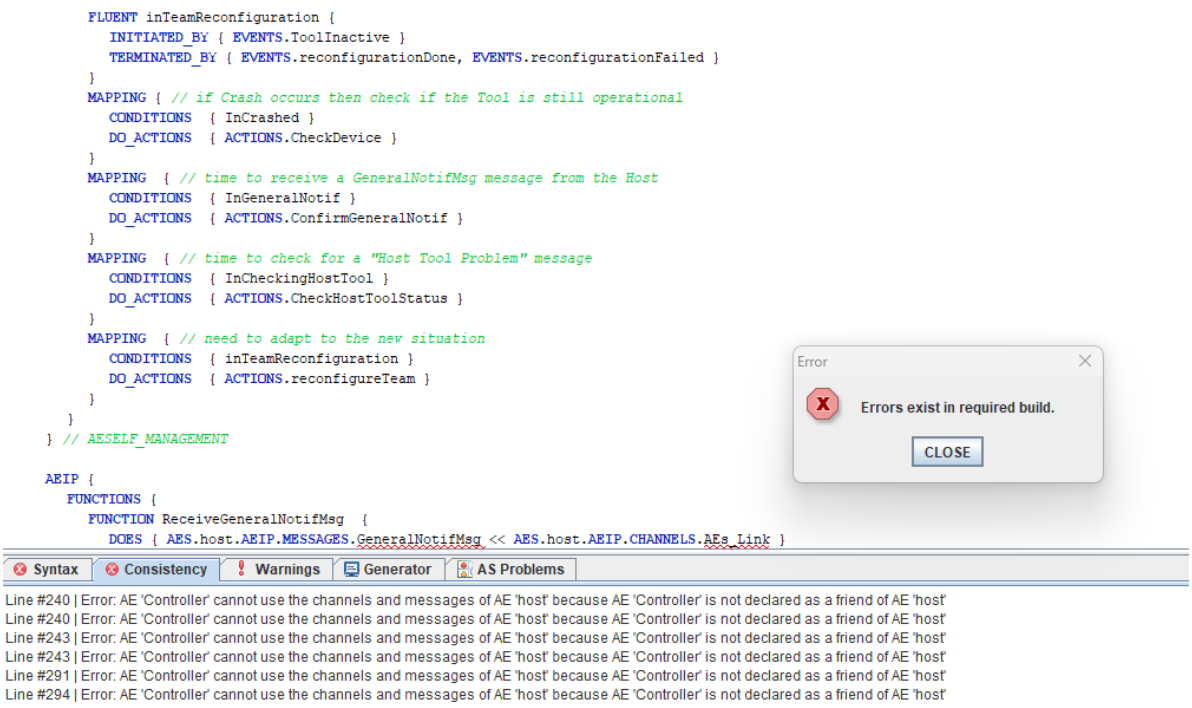

Figure 35: `FRIEND` accessibility error Intent 2.

3.3.3 INTENT 3

INTENT 3	
Intent :	*Add a new host!*
Solutions :	Carrier Networks
Intent User :	Network Operators
Intent Type :	Network and underlay network intent
Source :	[23]

Table 3: Specification/classification for INTENT 3

The most fundamental intent of a network is tackling modifications without any disturbance to the network operations. We have selected such a simple yet fundamental intent, whose specification/classification is depicted in Intent 3. Any adjustments, such as adding a new device, can result in a reconfiguration process for the whole system. If the network handles modifications through autonomic behaviors, time management will be optimized, and the risk of any human error will be reduced. The autonomic reconfiguration

of the network is a vital intent since applying all the adjustments separately in all the devices and tracking the dependent setting modification can take time, particularly in large-scale and complex networks. Adding a new host pragmatically is an operational task intent type, and reconfiguration of the system based on this intent can benefit all different intent users and solutions based on RFC 9316 [23], because the reconfiguration aspect is an autonomic behavior useful after any adjustment in the autonomic network. This intent is a fundamental goal for the autonomic network; however, it primarily refers to the two categories of network and underlay network service intent and network and underlay network intent types. Furthermore, technical or non-technical users of intent can use this autonomic behavior to enhance their network operations. Moreover, the messaging section that shows a new host is detected and transferred to all the autonomic network act as an alarm generation for network infrastructure under the network intent category. Due to the accessibility level of an autonomic reconfiguration behavior, it should be specified on a general basis, so all devices in the autonomic network can achieve this state of the network. It can be specified in a Controller autonomic node or the top-level hierarchy of the autonomic system as a general rule. Also, the autonomic network can have a metric that, if it changes, refers to detecting a new host. A managed element, like a monitoring tool, can track metric changes. Although this intent is simulated, we can omit the managed element section for the current implementation.

Self-reconfiguration properties of the network are an autonomic behavior for the entire system, and as a result, the `AS` tier should express it. Self-configuring under the self-management characteristics of an autonomic system can help the network to add a new host with ASSL specification language. The ASSL hierarchy for this specification also has an `ASIP` tier to help the autonomic network for the transmission of modification messages. In the self-configuring policy, the `newHostDetected` activates the `inReconfigurationForNewHost` fluent to lead the autonomic network to an action called `reconfigureNetwork`. The IMPL action named as `reconfigurationForNewHost` is implemented to extend the action's functionality for operators who can manually implement the generated code for any further

```
ASSELF_MANAGEMENT {
   SELF_CONFIGURING {
      FLUENT inReconfigurationForNewHost {
         INITIATED_BY { EVENTS.newHostDetected }
         TERMINATED_BY {EVENTS.reconfigurationForNewHostDone}
      }
      MAPPING {
         CONDITIONS { inReconfigurationForNewHost }
         DO_ACTIONS { ACTIONS.reconfigureNetwork }
      }
   }
} // ASSELF_MANAGEMENT

ACTIONS {
   ACTION IMPL reconfigurationForNewHost {
    RETURNS{BOOLEAN}
      TRIGGERS { EVENTS.reconfigurationForNewHostDone}
   }
   ACTION reconfigureNetwork {
      GUARDS { ASSELF_MANAGEMENT.SELF_CONFIGURING.inReconfigurationForNewHost }
     ENSURES { EVENTS.reconfigurationForNewHostDone}
      DOES {  call ASIP.FUNCTIONS.SendmsgNewDeviceFound;
            call ASIP.FUNCTIONS.ReceivemsgNewDeviceFound;
            call IMPL ACTIONS.reconfigurationForNewHost }
      ONERR_TRIGGERS { EVENTS.reconfigurationForNewHostDenied }
   }
} // ACTIONS
```

Figure 36: Elements of self-configuring properties in the autonomic network topology sample, as used to implement Intent 3.

complexity based on the system requirements. Also, if the reconfiguration fails for any reason, the `reconfigurationForNewHostDenied` event will show up. Figure 36 shows the procedures for moving the autonomic network to the state of reconfiguration.

The action calls functions of the `ASIP` tier, which are defined to transfer `msgNewDeviceFound` as a general notification message to the system to inform it about the new device. Figure 37 illustrates the `ASIP` tier. The metric `numberOfHost` for this intent counts the number of hosts that have been found.

3.3.4 INTENT 4

It is crucial to check networks in terms of device collapse since this issue is one of the most fundamental reasons for network failure. Device failures can occur for different reasons, such as errors in configuration, hardware incompatibilities, and network connection and link

```
ASIP {
   MESSAGES {
      MESSAGE msgNewDeviceFound {
         SENDER { ANY }
         RECEIVER { ANY }
         PRIORITY { 1 }
         MSG_TYPE {TEXT }
         BODY { "New Device" }
      }
   }
   CHANNELS {
      CHANNEL Public_Link {
         ACCEPTS { ASIP.MESSAGES.msgNewDeviceFound }
         ACCESS { SEQUENTIAL }
         DIRECTION { INOUT } }
   }
   FUNCTIONS {
      FUNCTION SendmsgNewDeviceFound {
         DOES { ASIP.MESSAGES.msgNewDeviceFound >> ASIP.CHANNELS.Public_Link }
      }
     FUNCTION ReceivemsgNewDeviceFound {
         DOES { ASIP.MESSAGES.msgNewDeviceFound << ASIP.CHANNELS.Public_Link }
      }
   }
}
```

Figure 37: `ASIP` tier specification used for Intent 3.

INTENT 4	
Intent :	*Remove non-functional device X!*
Solutions :	Carrier Networks
Intent User :	Network Operators
Intent Type :	Network and Underlay Network Intent
Source :	[23]

Table 4: Specification/classification for INTENT 4

damage. Both software and hardware issues can involve device crashing. From a low-level troubleshooting perspective, monitoring tools and packet tracing can estimate which device might be problematic. Low-level troubleshooting to recognize the exact device failure causes can take time, mainly if the network topology is complex or large-scale. A solution is to remove the nonfunctional device from the network topology and reconfigure it to adapt to the new topology. This convolution is due to the dependence of different devices on each other, and in case of removing the failed device from the system, whole devices should be reconfigured individually. Autonomic management of the network can facilitate the process

by considering Intent 4 as the solution to device failure. Intent 4 covers critical autonomic behaviors for network management. Removing a nonfunctional device from a network includes the following procedures. First, the recognition of the inoperable device; second, handling required notifications to inform the whole network about the failure; and third, removing the problematic device. Removing a device causes a modification in the network configuration, and thus a reconfiguration procedure is also necessary. Although Intent 4 is mainly categorized under the *Network and Underlay Network Intent* types rubric, it is notable that parts of the approach contain configuration that can be used as a basis in both *Networks and Underlay Network Service Intent* types. Also, since this intent results in the automation of tasks which can be removing a device, it can cover operational task intent types too. The section referring to notifying other nodes in the network is relevant to strategy intents since the goal is analogous to creating an alarm during a failure occurrence. Regarding intent users, the autonomic control of the procedures in Intent 4 speeds up and simplifies the responsibilities of network operators and administrators.

Two primary autonomic nodes can make up the infrastructure to achieve Intent 4. One Controller node for the central management of the procedures takes responsibility for notifications and removing the crashed device. Another autonomic node is the Host, which plays the role of the crashed device. The general rules shared among the nodes are related to the Host failure state, and this failure should be announced to the Controller. Furthermore, the autonomic policy for Intent 4 is self-healing since there is a network failure, and omitting the problematic device is the solution to solve the issue.

ASSL specification can formulate the states of the autonomic network for a self-healing policy where the crashed device is first recognized. Then two main reactions of the system to this state can occur. Firstly, The whole autonomic network, particularly the management node, is informed about the situation. In addition, the software's functionality on the device is checked to identify if the software is problematic. Finally, the removal occurs by shutting down the device. Intent 4 contains a self-healing policy because the removal scenario is a healing method for the system to remove the crashed device from the

network with minimum disruption or interference to the network's general performance. This definition of Intent 4 guides the scenario to have both *Abstract Formulation* and *Declarative Outcome Expression* intent objectives. ASSL can abstractly formulate this objective without considering machine-level instructions. This abstract formulation is shaped by designing fluents based on the logic of the scenario in a high-level representation and mapping the fluents to the appropriate actions to construct a desired autonomic behavior. The hierarchy of ASSL specification for Intent 4 consists of `AS`, `ASIP`, and `AE` tiers. In the `AS` tier, if a device crashes, it will be flagged by the `inCrashingDevice` fluent, the management is notified through the transmission of `msgdeviceCrashed` designed for this issue in `ASIP` tier by `senddeviceCrashedMsg`, and `receivedeviceCrashedMsg` functions through `Public_Link` channel. The `AS` fluents and actions and `ASIP` tiers are shown in Figure 38, and Figure 39. Since these specifications do not include specific details, such as a network metric, they can refer to the abstract formulation.

There are some fluents and actions in this specification with more detail, such as network metric or a call to a managed element. These parts of the specification demonstrate the *Declarative Outcome Expression* intent objective, such as in the removal procedure that starts with `InCheckingHostTool` fluent mapping the autonomic network to `CheckHostToolStatus` action as shown in Figure 40 where the system calls another action `shutDown` as depicted in Figure 41 to connect the autonomic network to the managed element called Host to use the `shutDownDevice` interface function. Figure 42 illustrates the managed element for shutting down the device. The simulation-only nature of the ASSL considers the device shutdown as the remedy to to remove the non-functional device from the autonomic network.

Furthermore, the Controller `AE` contains both a self-configuring policy for reconfiguration and a self-healing policy for adjusting the network operations to deal with the problem in a device by removing it. This illustrates the capability of the ASSL to handle composability by combining different policies to work together.

```
ASSELF_MANAGEMENT {
   SELF_HEALING {
      FLUENT inCrashingDevice {
         INITIATED_BY { EVENTS.deviceCrashed }
         TERMINATED_BY { EVENTS.managementNotified }
      }
      MAPPING {
         CONDITIONS { inCrashingDevice }
         DO_ACTIONS { ACTIONS.notifyManagement }
      }
   }
} // ASSELF_MANAGEMENT

ASARCHITECTURE {
   AELIST {AES.Host, AES.Controller}
   DIRECT_DEPENDENCIES { DEPENDENCY AES.Host { AES.Controller }}
   TRANSITIVE_DEPENDENCIES { DEPENDENCY AES.Controller  {AES.Host }}
   GROUPS {
      GROUP discovery {
         MEMBERS { AES.Host, AES.Controller }
         COUNCIL { AES.Controller   }
      }
   }
}

ACTIONS {
   ACTION notifyManagement { //notify management for the crashed device
      GUARDS { ASSELF_MANAGEMENT.SELF_HEALING.inCrashingDevice }
      DOES { CALL ASIP.FUNCTIONS.senddeviceCrashedMsg;
             ///
             CALL ASIP.FUNCTIONS.receivedeviceCrashedMsg
             }
   }
}
```

Figure 38: `AS` fluent, and action in Intent 4.

```
ASIP {
  MESSAGES {
    MESSAGE msgdeviceCrashed {
      SENDER { ANY }
      RECEIVER { ANY }
      PRIORITY { 1 }
      MSG_TYPE {TEXT }
      BODY { "Crashed Device" }
    }
  }
  CHANNELS {
    CHANNEL Public_Link {
      ACCEPTS { ASIP.MESSAGES.msgdeviceCrashed }
      ACCESS { SEQUENTIAL }
      DIRECTION { INOUT } }
  }
  FUNCTIONS {
    FUNCTION senddeviceCrashedMsg {
      DOES { ASIP.MESSAGES.msgdeviceCrashed >> ASIP.CHANNELS.Public_Link }
    }

    FUNCTION receivedeviceCrashedMsg{
      DOES { ASIP.MESSAGES.msgdeviceCrashed << ASIP.CHANNELS.Public_Link }
   }
  }
}
```

Figure 39: Fluent that manages the crash notification in Intent 4.

```
FLUENT InCheckingHostTool {
  INITIATED_BY { EVENTS.MsgGeneralNotifReceived }
  TERMINATED_BY {EVENTS.ToolInactive, EVENTS.ToolOk }
}

MAPPING  { // time to check for a "Host Tool Problem" message
  CONDITIONS  { InCheckingHostTool }
  DO_ACTIONS  { ACTIONS.CheckHostToolStatus }
}
```

Figure 40: Fluent for checking the status of the host in Intent 4.

```
ACTION CheckHostToolStatus {
   GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InCheckingHostTool }
   DOES {  CALL AEIP.FUNCTIONS.receiveToolInactiveMsg;
           CALL ACTIONS.shutDown}
   TRIGGERS {
      IF EVENTS.MsgToolInactiveReceived AND EVENTS.DeviceRemoved THEN
         EVENTS.ToolInactive
      END ELSE
         EVENTS.ToolOk
      END
   }
}

//REMOVING
 ACTION shutDown{
    DOES{CALL AEIP.MANAGED_ELEMENTS.Host.shutDownDevice }
    TRIGGERS{EVENTS.DeviceRemoved}
}
```

Figure 41: Action to shut down a failed device in Intent 4.

```
MANAGED_ELEMENT Host{
   INTERFACE_FUNCTION shutDownDevice{}
}
```

Figure 42: Shut down interface function in Intent 4.

INTENT 5	
Intent :	*Provide service S with guaranteed bandwidth for customer A.*
Solutions :	Carrier Networks
Intent User :	Service Operators
Intent Type :	Customer Service Intent
Source :	[23]

Table 5: Specification/classification for INTENT 5

3.3.5 INTENT 5

Service provisioning is a *Customer Service Intent*, and service operators are one of its users. Intents can benefit the networks by providing services to network customers. Network metrics such as bandwidth can directly impact the performance of this intent, and the context of autonomicity enables the network to respond quickly and adapt to the demands and expectations of its users. Besides ensuring the users receive the services, this intent is essential

to stakeholders allowing them to increase their financial benefits by providing various services to their users.

A central autonomic node should control service provisioning as the management or controller node. This central management node is helpful to the system in case of composability and potential scalability. This is because it is possible to add new autonomic elements to the system to work with the central management when required. Also, there should be a customer node to handle the procedures related to customer A as mentioned in intent Intent 5, and the network metric in this intent is link bandwidth which should be guaranteed.

It is possible to guarantee that the network can deliver the services reliably and efficiently, even when the network is subjected to changing conditions or requirements, by describing the desired behavior and needs of the services using ASSL specification. To specify the intent Intent 5, there is `AS` and `ASIP` tier as shown in Figure 43 to express a general logical predicate or network state for the autonomic network. The `AS` level consists of a self-healing policy for when the network is in trouble, and the management should be notified through public messages defined in the `ASIP` tier. In the `AE` tier, the Controller is defined as the autonomic node to manage a self-healing policy to guide the autonomic network to the appropriate action of checking the device if the autonomic network is in a state where the bandwidth requirement is not met.

In intent Intent 5, most of the specifications happen on the customer side since this is the customer who should be satisfied with the service. Therefore, states occur under the self-healing policy of the `CustomerA AE`. Firstly, the service is run under the `InServiceS` fluent; then, there are three states to describe an undesirable condition for the autonomic element with their corresponding actions. The fluents as shown in Figure 44 for the unpleasant state for intent Intent 5 are fluents: `InUnsatisfiedBandwidth`, `InInterfaceInActive`, `InUnsatisfiedBandwidthSimulation`.

With the fluents, different aspects of the autonomic behavior based on the intent are handled by mapping the autonomic network to the corresponding reaction to each state. To

```
AS AutonomicNetwork {
  ASSELF_MANAGEMENT {
    SELF_HEALING {
      FLUENT inProblem {INITIATED_BY { EVENTS.DeviceCrashed }TERMINATED_BY { EVENTS.managementNotified }}
      MAPPING { CONDITIONS { inProblem }  DO_ACTIONS { ACTIONS.notifyManagement }}}//SELF_HEALING
  }//ASSELF_MANAGEMENT
  ASARCHITECTURE {
    AELIST {AES.CustomerA, AES.Controller}
    DIRECT_DEPENDENCIES { DEPENDENCY AES.CustomerA { AES.Controller }}
    TRANSITIVE_DEPENDENCIES { DEPENDENCY AES.Controller  {AES.CustomerA }}
    GROUPS {
      GROUP discovery {
   MEMBERS { AES.CustomerA, AES.Controller }
   COUNCIL { AES.Controller }
      }//GROUP discovery
    }//GROUPS
  }//ASARCHITECTURE
  ACTIONS {
    ACTION notifyManagement {
      GUARDS { ASSELF_MANAGEMENT.SELF_HEALING.inProblem }
      DOES { CALL ASIP.FUNCTIONS.sendDeviceCrashedMsg;
             CALL ASIP.FUNCTIONS.receiveDeviceCrashedMsg}
    }//ACTION notifyManagement
  }//ACTIONS
  EVENTS {
    EVENT DeviceCrashed { }
    EVENT managementNotified {
      ACTIVATION  { SENT { ASIP.MESSAGES.msgDeviceCrashed } }
    }//EVENT managementNotified
  } //EVENTS
} //AS AutonomicNetwork
//==================== AS interaction protocol ============
ASIP {
  MESSAGES {
    MESSAGE msgDeviceCrashed { SENDER { ANY } RECEIVER { ANY }PRIORITY { 1 } MSG_TYPE { TEXT } BODY { "Crashed Device" }}//MESSAGE msgDeviceCrashed
  }//MESSAGES
  CHANNELS {
    CHANNEL Public_Link { ACCEPTS { ASIP.MESSAGES.msgDeviceCrashed } ACCESS { SEQUENTIAL } DIRECTION { INOUT }}}//CHANNELS
  FUNCTIONS {
    FUNCTION sendDeviceCrashedMsg { DOES { ASIP.MESSAGES.msgDeviceCrashed >> ASIP.CHANNELS.Public_Link }}
    FUNCTION receiveDeviceCrashedMsg { DOES { ASIP.MESSAGES.msgDeviceCrashed << ASIP.CHANNELS.Public_Link }} }//FUNCTIONS
}//ASIP
```

Figure 43: `AS` and `ASIP` tiers for Intent 5

```
AE CustomerA {
  AESELF_MANAGEMENT {
    SELF_HEALING {
      FLUENT InServiceS{
          INITIATED_BY{EVENTS.TimeForServiceS}
          TERMINATED_BY{EVENTS.ServiceSTimOut}}//FLUENT InServiceS
      FLUENT InUnsatisfiedBandwidth {
         INITIATED_BY { EVENTS.UnsatisfiedBandwidth }
         TERMINATED_BY { EVENTS.InterfaceChecked, EVENTS.InterfaceNotChecked }}
      FLUENT InInterfaceInActive {
         INITIATED_BY { EVENTS.InterfaceInActive }
         TERMINATED_BY { EVENTS.IsMsgInterfaceInActiveSent}}
      FLUENT InUnsatisfiedBandwidthSimulation {
         INITIATED_BY { EVENTS.TimeToSimulateUnsatisfiedBandwidth }
         TERMINATED_BY { EVENTS.UnsatisfiedBandwidth, EVENTS.CustomerAInUnsatisfiedBandwidth }}

      MAPPING { CONDITIONS {InServiceS} DO_ACTIONS{ACTIONS.ServiceSinProgress}}
      MAPPING { CONDITIONS { InUnsatisfiedBandwidth }DO_ACTIONS { ACTIONS.CheckBandwidthIssues }}
      MAPPING { CONDITIONS { InInterfaceInActive } DO_ACTIONS { ACTIONS.NotifyForInActiveInterface }}
      MAPPING { CONDITIONS { InUnsatisfiedBandwidthSimulation } DO_ACTIONS { ACTIONS.SimulateUnsatisfiedBandwidth }}
    }//SELF_HEALING
  }//AESELF_MANAGEMENT
```

Figure 44: Customer A self-Healing for Intent 5

```
ACTION ServiceSinProgress{
    GUARDS{AESELF_MANAGEMENT.SELF_HEALING.InServiceS}
    DOES{CALL AEIP.MANAGED_ELEMENTS.ServiceS.ServiceSInt;
          CALL AEIP.FUNCTIONS.sendRunServiceSMsg}
    TRIGGERS{EVENTS.ServiceSTimOut}}
```

Figure 45: Service S action for Intent 5

```
METRICS {
  METRIC BandwidthA {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.CustomerA.getBandwidthA }
    DESCRIPTION { "This is the bandwidth for customer A!" }
    MEASURE_UNIT { "Mbit/s" }
    VALUE { 10 }
    THRESHOLD_CLASS { DECIMAL [5 ~ 10] }
  }
}
```

Figure 46: Bandwidth metric for Intent 5

provide service S as the first requirement of the intent, as can be seen in Figure 45 action `ServiceSinProgress` is in charge that calls `ServiceSInt` managed element which connects the specification to the interface designed for service S in an abstract manner.

The undesirable state for the `CustomerA` occurs when the network metric defined as `BandwidthA` falls outside of the acceptable threshold. In our example, the `InUnsatisfiedBandwidthSimulation` fluent is designed to show what happens if the metric, as shown in Figure 46 is violated due to the incompatibility of `BandwidthA` threshold range and the set value of it the simulation of an undesirable state for the customer A node. Figure 47 illustrates the reactions of `CustomerA AE` to the violation of metric or any other undesirable behaviors of the autonomic node to handle the situation.

The system checks if there is an issue with the bandwidth through the `CheckBandwidthIssues` action. The action completes by checking if the device is active, leading the system to check the interface issues through `checkInterface`. Then, if the interface is not functional, a notification message `InterfaceInActiveMsg` is sent to the `private_link` between the Controller and `CustomerA` through the

```
ACTION IMPL checkInterface {
  RETURNS { BOOLEAN }
 TRIGGERS { EVENTS.InterfaceChecked }
}

ACTION CheckBandwidthIssues {
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InUnsatisfiedBandwidth AND METRICS.BandwidthA}
  VARS { BOOLEAN deviceActive }
  DOES { deviceActive = true; CALL ACTIONS.checkInterface }
  TRIGGERS { IF (not deviceActive) THEN EVENTS.InterfaceInActive END }
  ONERR_TRIGGERS { EVENTS.InterfaceNotChecked }
}

ACTION NotifyForInActiveInterface {
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InInterfaceInActive }
  DOES { CALL AEIP.FUNCTIONS.SendInterfaceInActiveMsg }
}

ACTION SimulateUnsatisfiedBandwidth {
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InUnsatisfiedBandwidthSimulation AND METRICS.BandwidthA }
  DOES { SET METRICS.BandwidthA.VALUE = 1 }
  TRIGGERS{EVENTS.UnsatisfiedBandwidth}
  ONERR_TRIGGERS { EVENTS.CustomerAInUnsatisfiedBandwidth } }
}
```

Figure 47: Customer A actions in Intent 5

`SendInterfaceInActiveMsg` function. The `AEIP` for `CustomerA` is depicted in Figure 48. Also, it is shown in the figure that the sender of the messages is the `CustomerA AE` to `Controller` AE.

This design to have a hierarchy of specific actions and states for a particular customer node, but only for some of the autonomic elements, demonstrate the properties of ASSL to create a local network management procedure for each autonomic element. This local autonomic behavior is also accessible in terms of interactions with other autonomic elements if they are defined as `FRIENDS` to that element. Furthermore, in this way, abstract security is possible for the autonomic network.

3.3.6 INTENT 6

The ability of virtual machines (VMs) to communicate with one another and with other network devices within a data center is referred to as the connectivity of virtual machines in a

```
AEIP {

  MESSAGES {
    FINAL MESSAGE InterfaceInActiveMsg {
       SENDER { AES.CustomerA } RECEIVER { AES.Controller } MSG_TYPE { TEXT } BODY { "Interface Issue" } }
    FINAL MESSAGE RunServiceSMsg {
       SENDER { AES.CustomerA  } RECEIVER { AES.Controller } MSG_TYPE { TEXT } BODY { "Initiation of Service S" }}}//MESSAGES

  CHANNELS {
    CHANNEL Private_Link {
       ACCEPTS { AEIP.MESSAGES.InterfaceInActiveMsg , AEIP.MESSAGES.RunServiceSMsg }
       ACCESS { SEQUENTIAL }
        DIRECTION { INOUT }}}//CHANNELS

  FUNCTIONS {
    FUNCTION SendInterfaceInActiveMsg { DOES { AEIP.MESSAGES.InterfaceInActiveMsg >> AEIP.CHANNELS.Private_Link }}
    FUNCTION sendRunServiceSMsg { DOES { AEIP.MESSAGES.RunServiceSMsg >> AEIP.CHANNELS.Private_Link } }}//FUNCTIONS

  MANAGED_ELEMENTS {
    MANAGED_ELEMENT CustomerA { INTERFACE_FUNCTION getBandwidthA { RETURNS { DECIMAL } } }
    MANAGED_ELEMENT ServiceS{ INTERFACE_FUNCTION ServiceSInt{}}}//MANAGED_ELEMENTS

}//AEIP
```

Figure 48: Customer A `AEIP` for Intent 5

INTENT 6	
Intent :	*Request connectivity between VMs A, B, and C in network N1.*
Solutions :	DC Networks
Intent User :	Cloud Administrator
Intent Type :	Cloud Management Intent
Source :	[23]

Table 6: Specification/classification for INTENT 6

data center (DC) network. The connection simulation refers to specifying intent connectivity scope for DC network solutions. The purpose of linking virtual machines in a data center network is typically to give the VMs a way to share resources and interact with one another, as well as to enable management and monitoring of the VMs and the data center network as a whole. In a nutshell, the intent of networking VMs in a data center is to increase the data center's manageability and performance while allowing the VMs, as the fundamental components of the network, to share resources and communicate with one another. Virtual switches and network interfaces, which may be set up to give different degrees of connectivity and security, can be used to do this task. Intent 6 focuses on the connectivity area since the goal of the autonomic network N1 is to connect three virtual machines in a data center. The main actors are the virtual machines, and the main activity is their communication.

The abstract states of this implementation are requesting connectivity, processing these requests, and adjusting VM settings. Due to the abstract formulation of connectivity with ASSL, there is no requirement to consider exact details related to lower-level network configurations, like how to create a network bridge between the physical infrastructure of the network. Creating fluents for connectivity requests and mapping them to the proper action based on the connection name parameter to process the connectivity request is used to encapsulate the configuration details for this level of intent expression, proving one of the declarative outcomes of Intent 6 as shown under the `VM_CONNECTION` policy in Figure 49 besides its action as shown in Figure 50. The specification contains `AS` and `ASIP` tiers for general and distributed autonomic behavior, four AEs including one Controller, and three VMs as the actors to represent the local autonomic behavior. The distributed autonomic behavior is expressed under the `AS` tier as depicted in Figure 49. The `InConnectionsReqProcessed` that verifies the connections are processed is formulated abstractly. At the same time, it does not include detailed information on implementation and directly conducts the system to `VerifyConnectionsProcessed` as shown in Figure 50. The functions of this declarative outcome to process the connections on the `AE` sides also investigate the parameter names to determine the destination of `AEIP` messages as depicted in Figure 51.

This autonomic network specification has four autonomic elements to apply distributed and local autonomic behaviors, including a Controller and three others as representations of the virtual machines A, B, and C as dependent AEs to construct the `AS` architecture as outlined in Figure 52. In the `AS` tier, the request for connectivity is handled through `inConnectivityReq` fluents mapping the autonomic network to `ProcessConnectivity IMPL` action that works based on the connection name. These network states start when each of the AEs except the Controller sends an initiation message such as `Msg_InitiateVM_A` as shown in Figure 53 message through abstract `Connectivity_Link` created under the `ASIP` block to be shared among the whole architecture. The message is received on the other side of this channel to initiate `RunVM` events. After processing all the connections, a fluent called

```
ASSELF_MANAGEMENT {
  OTHER_POLICIES {
    POLICY VM_CONNECTION {
      //VM A, VM B initiation result in a connectivity between VMs A,B
      FLUENT inConnectivityReqA_B {
        INITIATED_BY { EVENTS.RunVM_A,EVENTS.RunVM_B }
        TERMINATED_BY { EVENTS.ConnectivityProcessedAB }
      }
      //VM A, VM C initiation result in a connectivity between VMs A,C
      FLUENT inConnectivityReqA_C {
        INITIATED_BY { EVENTS.RunVM_A,EVENTS.RunVM_C }
        TERMINATED_BY { EVENTS.ConnectivityProcessedAC }
      }
      //VM C, VM B initiation result in a connectivity between VMs C,B
      FLUENT inConnectivityReqB_C {
        INITIATED_BY { EVENTS.RunVM_B,EVENTS.RunVM_C }
        TERMINATED_BY { EVENTS.ConnectivityProcessedBC }
      }
      Fluent InConnectionsReqProcessed{
        INITIATED_BY{EVENTS.ConnectivityProcessedAB,
                     EVENTS.ConnectivityProcessedAC,
                     EVENTS.ConnectivityProcessedBC}
        TERMINATED_BY{EVENTS.ConnectionsProcessed}
      }
      MAPPING {
        CONDITIONS{InConnectionsReqProcessed}
        DO_ACTIONS{ACTIONS.VerifyConnectionsProcessed}
      }
      MAPPING {
        CONDITIONS { inConnectivityReqA_B}
        DO_ACTIONS { ACTIONS.ProcessConnectivity("ConnectionA_B") }
      }
      MAPPING {
        CONDITIONS { inConnectivityReqA_C}
        DO_ACTIONS { ACTIONS.ProcessConnectivity("ConnectionA_C") }
      }
      MAPPING {
        CONDITIONS { inConnectivityReqB_C}
        DO_ACTIONS { ACTIONS.ProcessConnectivity("ConnectionB_C") }
      }
    }
  }
} // ASSELF_MANAGEMENT
```

Figure 49: `AS OTHER-POLICIES` for Intent 6.

```
ACTIONS {
   ACTION IMPL ProcessConnectivity {
    // This action processes a connectivity request based
   //on the connection name which is chosen according to the two sides of the connection
      PARAMETERS { string ConnectionName  }
      GUARDS { ASSELF_MANAGEMENT.OTHER_POLICIES.VM_CONNECTION.inConnectivityReqA_B OR
               ASSELF_MANAGEMENT.OTHER_POLICIES.VM_CONNECTION.inConnectivityReqA_C OR
               ASSELF_MANAGEMENT.OTHER_POLICIES.VM_CONNECTION.inConnectivityReqB_C }
      TRIGGERS {
         IF ConnectionName = "ConnectionA_B" THEN EVENTS.ConnectivityProcessedAB    END
         ELSE
            IF ConnectionName = "ConnectionA_C" THEN EVENTS.ConnectivityProcessedAC END
            ELSE
               IF ConnectionName = "ConnectionB_C" THEN EVENTS.ConnectivityProcessedBC END

               END
            END
      }
   }
  ACTION IMPL VerifyConnectionsProcessed{
    TRIGGERS{EVENTS.ConnectionsProcessed}
    ONERR_TRIGGERS{EVENTS.ConnectionsNotVerified}
  }


} // ACTIONS
```

Figure 50: `AS` actions for Intent 6.

```
  FUNCTIONS {
//This function is used to help us to send general messages in the Connectivity_Link
    FUNCTION sendConnectivityMsg {
//instead of names of the connection we can have the static IP assigned to each VM as a string var. This is because of how we connect VMs together.
      PARAMETERS { string ConnectionName }
      DOES {
        IF ConnectionName = "ConnectionA_B" THEN ASIP.MESSAGES.Msg_InitiateVM_A >> ASIP.CHANNELS.Connectivity_Link END
        ELSE
          IF ConnectionName = "ConnectionA_C" THEN ASIP.MESSAGES.Msg_InitiateVM_B >> ASIP.CHANNELS.Connectivity_Link END
          ELSE
            IF ConnectionName = "ConnectionB_C" THEN ASIP.MESSAGES.Msg_InitiateVM_C >> ASIP.CHANNELS.Connectivity_Link END
            END
          END
      // END
      }
    }
//This function is used to help us to receive general messages in the Connectivity_Link on the receiver sides
    FUNCTION receiveConnectivityMsg {
      PARAMETERS { string ConnectionName }

      DOES {//Based on the connection name, the message of inititing connectivity will be put into connectivity_Link
        IF ConnectionName = "ConnectionA_B" THEN ASIP.MESSAGES.Msg_InitiateVM_A << ASIP.CHANNELS.Connectivity_Link END
        ELSE
          IF ConnectionName = "ConnectionA_C" THEN ASIP.MESSAGES.Msg_InitiateVM_B << ASIP.CHANNELS.Connectivity_Link END
          ELSE
            IF ConnectionName = "ConnectionB_C" THEN ASIP.MESSAGES.Msg_InitiateVM_C << ASIP.CHANNELS.Connectivity_Link END

            END
          END
      // END
      }
    }
  } // FUNCTIONS
```

Figure 51: `ASIP` functions for Intent 6.

```
//In this block, the architecture of the autonomic network is specified
//including a controller as a central connector or coordinator and three
//VMs as the autonomic elements dependent on the Controller
  ASARCHITECTURE {
    AELIST {AES.Controller, AES.VM_A, AES.VM_B, AES.VM_C}
    DIRECT_DEPENDENCIES {
      DEPENDENCY AES.VM_A { AES.Controller }
      DEPENDENCY AES.VM_B { AES.Controller }
      DEPENDENCY AES.VM_C { AES.Controller }
    }
//To connect the autonomic elements, all of them are grouped together
    GROUPS {
      GROUP ControllerGroup {
        MEMBERS { AES.Controller, AES.VM_A, AES.VM_B, AES.VM_C}
      }
    }
  } // ASARCHITECTURE
```

Figure 52: `AS` architecture for Intent 6.

`InConnectionsReqProcessed` is triggered, meaning that the event `TimeToSetVM` is activated to initiate the network state where VM settings can be adjusted through an `IMPL` action called `AdjustVMSetting`. The `IMPL` action provides the possibility of further implementation in more detail from a technical perspective for the intent users who are Cloud administrators knowledgeable about technical information in this case. More details about the Controller specification are depicted in Figure 54.

3.3.7 INTENT 7

INTENT 7	
Intent :	*Request video conference between end users A and B.*
Solutions :	Enterprise Networks
Intent User :	End Users
Intent Type :	Customer Service Intent
Source :	[23]

Table 7: Specification/classification for INTENT 7

One of the most useful network technologies is video conferencing, and different networking providers have unique techniques and equipment to make this technology

```
ASIP {
   MESSAGES {
      MESSAGE Msg_InitiateVM_A {
          SENDER { AES.VM_A }   RECEIVER { ANY } PRIORITY { 1 } MSG_TYPE {  BIN } }
      MESSAGE Msg_InitiateVM_B  {
          SENDER { AES.VM_B }   RECEIVER { ANY } PRIORITY { 1 } MSG_TYPE {  BIN } }
      MESSAGE Msg_InitiateVM_C {
           SENDER { AES.VM_C }   RECEIVER { ANY } PRIORITY { 1 } MSG_TYPE {  BIN } }
   } // MESSAGES

   CHANNELS {
// this is the public communication channel shared
//between all the autonomic network so general policies
//like general messages for initiation of the nodes are defined here.
      CHANNEL Connectivity_Link {
         ACCEPTS { ASIP.MESSAGES.Msg_InitiateVM_A,
                   ASIP.MESSAGES.Msg_InitiateVM_B,
                   ASIP.MESSAGES.Msg_InitiateVM_C}
         ACCESS { SEQUENTIAL }
         DIRECTION { INOUT } }
   } // CHANNELS
```

Figure 53: `ASIP` messages for Intent 6.

```
  AE Controller {
     AESELF_MANAGEMENT {
        OTHER_POLICIES {
           POLICY VM_CONNECTION {

              FLUENT inTakeVMSetting {
//To start VMs networking a service template
//or some VM setting should be set so this state
//of the network handles this process
                 INITIATED_BY { EVENTS.TimeToSetVM }
                 TERMINATED_BY { EVENTS.VMSettingAdjusted }
              }
              MAPPING {
                 CONDITIONS { inTakeVMSetting }
                 DO_ACTIONS { ACTIONS.AdjustVMSetting }
              }
           }
        }
     } // AESELF_MANAGEMENT

     ACTIONS {
        ACTION IMPL AdjustVMSetting {
           GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.VM_CONNECTION.inTakeVMSetting }
           TRIGGERS { EVENTS.VMSettingAdjusted }
        }
     } // ACTIONS

     EVENTS {
        EVENT TimeToSetVM { ACTIVATION {OCCURRED{AS.EVENTS.ConnectionsProcessed}} }
```

Figure 54: Controller `OTHER-POLICIES` for Intent 6.

available to their users. There are different intent types in intent classification for this technology, subsuming strategy, customer service, and network service intents. This policy is crucial to cover in the autonomic network area because of its vast usability and its different user types who might be knowledgeable about networking, like Enterprise Administrators, or Nontechnical users, such as End users. Also, the autonomic management of video conference events reduces the risk of dissatisfaction among its users since it is a real-time network policy. In real-time network events, autonomic behavior control increases the reaction time to any problem. It decreases the risk of human operator errors or time-consuming troubleshooting procedures that can cause hesitations which are not allowed in real-time events.

Meeting software on connected devices primarily makes up the infrastructure for video conferencing. Users may encounter incompatibility problems if they use infrastructure from different companies; therefore, an integrated abstract video conferencing policy can reduce this problem. The main actors involved in a video conference intent with autonomic management are the autonomic node to play the role of the central administration to take the responsibility of managing the initiation of the intent and processing it. To complete its duties, it should have access to the users that use this intent as other autonomic nodes as the subset of this system. The accessibility can be achieved by defining a managed element for each user in the management node. To represent each user, there should be an individual autonomic node. Generally, intents Intent 7 and Intent 8 are derived from a video conferencing policy; there are similar actors; however, based on the intent details, roles might differ in each case. To clarify how assignments are categorized, the controller node takes the responsibility of performing joint session messages between its multiple end users. Also, it can create video conferences, where in Intent 8, this feature is specified. To complete a video conferencing policy, the end users have similar activities, including initiating a meeting in their node, recording the meeting, and sending the meeting record to the central management. End users can request the initiation of a video conference from the central node, expressed in Intent 7.

The equivalent term for an autonomic node in ASSL is an autonomic element. The

procedures contributed in video conferencing can be formulated through ASSL fluents and then mapped to the proper actions resulting in abstract and declarative outcome formulation. The declarative outcome refers to creating a conceptual package as a goal for the network consisting of different states that occur while achieving the goal without considering how these goals are achieved. The abstract formulation is possible if details about the low-level configuration, like setting IP prefixes and configuring port numbers to connect devices, are viewed as a finite state of creating a link in the autonomic network domain for video conferencing. The autonomic network components required for video conferencing policy, as the abstract simulation of the infrastructure for Intent 7, are a Controller and two other autonomic elements as the end users A and B. The steps a network enters to establish videoconferencing are designed as fluents to shape the general abstract policy of processing a video conference.

The video conferencing policy contains fluents like `inProcessingVideoConference_A` initiated by the event `VideoConference_AReceived` when `msgVideoConferenceUser_A` is received from the end user A, and then this fluent is mapped to `processVideoConference` action. The `self-configuring` and `VideoConferencing` policies with their fluents can be seen in Figure 55.

The declarative outcomes for Intent 7 can be defined by the policies guiding the autonomic network to do different actions in an abstract manner and with the minimum amount of parametric details. Processing a meeting type, whether a scheduled or instant meeting, is an example to show the capability of ASSL to respect the *Declarative Outcome Expression* intent objective. To explicate more, the autonomic behavior resulting in such an outcome starts when the network is in `inProcessingVideo` fluent as shown in Figure 56. Then the network is conducted to `ProcessVideo` action where based on the meeting type, another action `processMeetingType` is called to guide the system to call the appropriate interface function of the managed element, which is a monitoring tool defined for each of the meeting types. The actions to complete these procedures are depicted in Figure 57. To attain *Abstract Formulation*, an example is defining an autonomic behavior to start the video for

```
ASSELF_MANAGEMENT {
 SELF_CONFIGURING{
     FLUENT inRequestVideoConference_A {
       INITIATED_BY { EVENTS.TimeToRequest_A }
       TERMINATED_BY { EVENTS.Request_AProcessed }}
     FLUENT inRequestVideoConference_B {
       INITIATED_BY { EVENTS.TimeToRequest_B }
       TERMINATED_BY { EVENTS.Request_BProcessed }}
     MAPPING {
       CONDITIONS { inRequestVideoConference_A}
       DO_ACTIONS { ACTIONS.processRequest("User_A") } }
     MAPPING {
       CONDITIONS { inRequestVideoConference_B}
       DO_ACTIONS { ACTIONS.processRequest("User_B") }}
   }//SELF_CONFIGURING
 OTHER_POLICIES {
  POLICY VideoConferencing {
     FLUENT inProcessingVideoConference_A {
        INITIATED_BY { EVENTS.VideoConference_AReceived }
        TERMINATED_BY { EVENTS.VideoConference_AProcessed }
        }
     FLUENT inProcessingVideoConference_B {
       INITIATED_BY { EVENTS.VideoConference_BReceived }
       TERMINATED_BY { EVENTS.VideoConference_BProcessed }
       }
     MAPPING {
       CONDITIONS { inProcessingVideoConference_A}
       DO_ACTIONS { ACTIONS.processVideoConference("User_A") }
     }
     MAPPING {
       CONDITIONS { inProcessingVideoConference_B}
       DO_ACTIONS { ACTIONS.processVideoConference("User_B") }
       }
     }//VideoConferencing
   }//OTHER_POLICIES
} // ASSELF_MANAGEMENT
```

Figure 55: `AS` self-configuring and VideoConferencing for Intent 7

video conferencing. This is expressed through `inStartingVideo` guiding the network to perform `StartVideo` action as outlined in Figure 58. This procedure does not include any details such as metrics or parameters.

The `inProcessingVideoConference` fluents in the `AS` tier for both users abstract the processing of video conferences by doing the `processVideoConference` action according to the user name parameter. These fluents are initiated when `msgVideoConferenceUser` for each user is received through an abstract video conference link specified under the `ASIP` tier to transmit public messages. The packages created by the ASSL toolset for `ASIP` tiers are shown in Figure 59.

```
AE Controller {

   VARS { boolean initiateWithA }

   AESELF_MANAGEMENT {
      OTHER_POLICIES {
         POLICY VideoConferencing {
            FLUENT inStartingingVideo {
               INITIATED_BY { EVENTS.TimetoStartVideo }
               TERMINATED_BY { EVENTS.VideoStarted }
            }
            FLUENT inProcessingVideo {
               INITIATED_BY { EVENTS.VideoStarted }
               TERMINATED_BY { EVENTS.VideoProcessed }
         }
            MAPPING {
               CONDITIONS { inStartingingVideo }
               DO_ACTIONS { ACTIONS.StartVideo }
         }
            MAPPING {
               CONDITIONS { inProcessingVideo }
               DO_ACTIONS { ACTIONS.ProcessVideo }
            }
         }//VideoConferencing
      }//OTHER_POLICIES
   } // AESELF_MANAGEMENT
```

Figure 56: The controller video conferencing policy for Intent 7.

Under the `AES` level, the Controller handles central management of the video conferencing policy through the states called `inStartingVideo` and `inProcessingVideo` which are illustrated in Figure 56. The former initiates the video conference to start the video, then processes the meeting type, including scheduled and instant meetings. The specification also allows for a connection between the abstract simulation and meeting software towards managed elements. The managed elements are called Meeting tools for each user with their interface functions for starting the video in each user and bringing about accessibility to interfaces expressed for the scheduled and instant meeting types. The Controller outlines negotiation messages to form interaction sessions with the End users. Thus, the End users and the Controller can exchange beginning session messages to initiate the interaction, send the video messages during this session, and then exchange the end session message to finish

```
ACTION processMeetingType {
   PARAMETERS { string MeetingType }
   DOES {
      IF AES.Controller.initiateWithA THEN
         IF MeetingType = "ScheduledMeeting" THEN
            call AEIP.MANAGED_ELEMENTS.MeetingTool_A.ScheduledMeetingInterface
         END;
         IF MeetingType = "InstantMeeting" THEN
            call AEIP.MANAGED_ELEMENTS.MeetingTool_A.InstantMeetingInterface
      END
   END
      ELSE
         IF MeetingType = "ScheduledMeeting" THEN
            call AEIP.MANAGED_ELEMENTS.MeetingTool_B.ScheduledMeetingInterface
         END;
         IF MeetingType = "InstantMeeting" THEN
            call AEIP.MANAGED_ELEMENTS.MeetingTool_B.InstantMeetingInterface
         END
      END;

      call AEIP.FUNCTIONS.sendBeginMsgs (MeetingType);
         call AEIP.FUNCTIONS.sendVideoConferenceMsg;
      call AEIP.FUNCTIONS.sendEndSessionMsgs (MeetingType)
   }
}
ACTION ProcessVideo {
   GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.VideoConferencing.inProcessingVideo }
   DOES {
      call ACTIONS.processMeetingType("ScheduledMeeting");
      call ACTIONS.processMeetingType("InstantMeeting")
   }
   TRIGGERS { EVENTS.VideoProcessed }
}
// ACTIONS
```

Figure 57: The controller actions for Intent 7.

```
ACTION StartVideo {
   GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.VideoConferencing.inStartingingVideo }
   DOES {
      IF AES.Controller.initiateWithA THEN
         call AEIP.MANAGED_ELEMENTS.MeetingTool_A.StartVideo;
         AES.Controller.initiateWithA = false
      END
      ELSE
         call AEIP.MANAGED_ELEMENTS.MeetingTool_B.StartVideo;
         AES.Controller.initiateWithA = true
      END
   }
   TRIGGERS { EVENTS.VideoStarted }
}
```

Figure 58: Action to model start video for Intent 7.

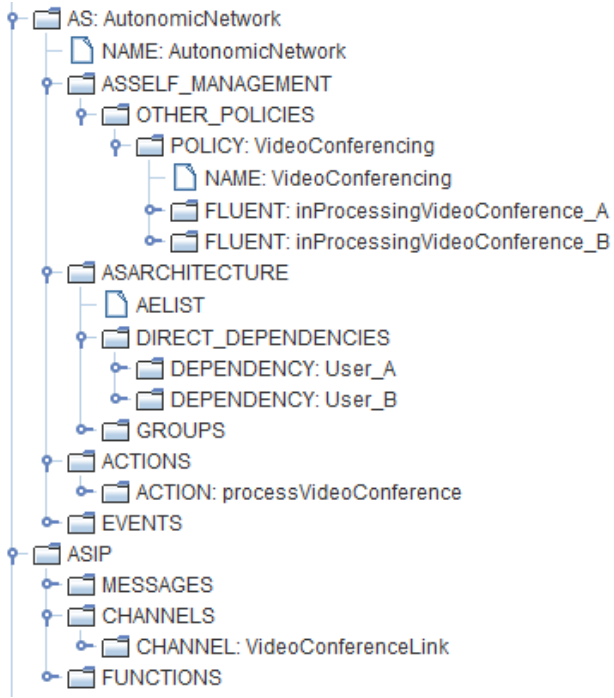

Figure 59: `AS` and `ASIP` tiers structures for Intent 7

```
//====== AEs that can use the messages and channels specified by the Controller AE ======
FRIENDS {
  AELIST { AES.User_A, AES.User_B}
}
```

Figure 60: Friends of the controller `AE` in Intent 7.

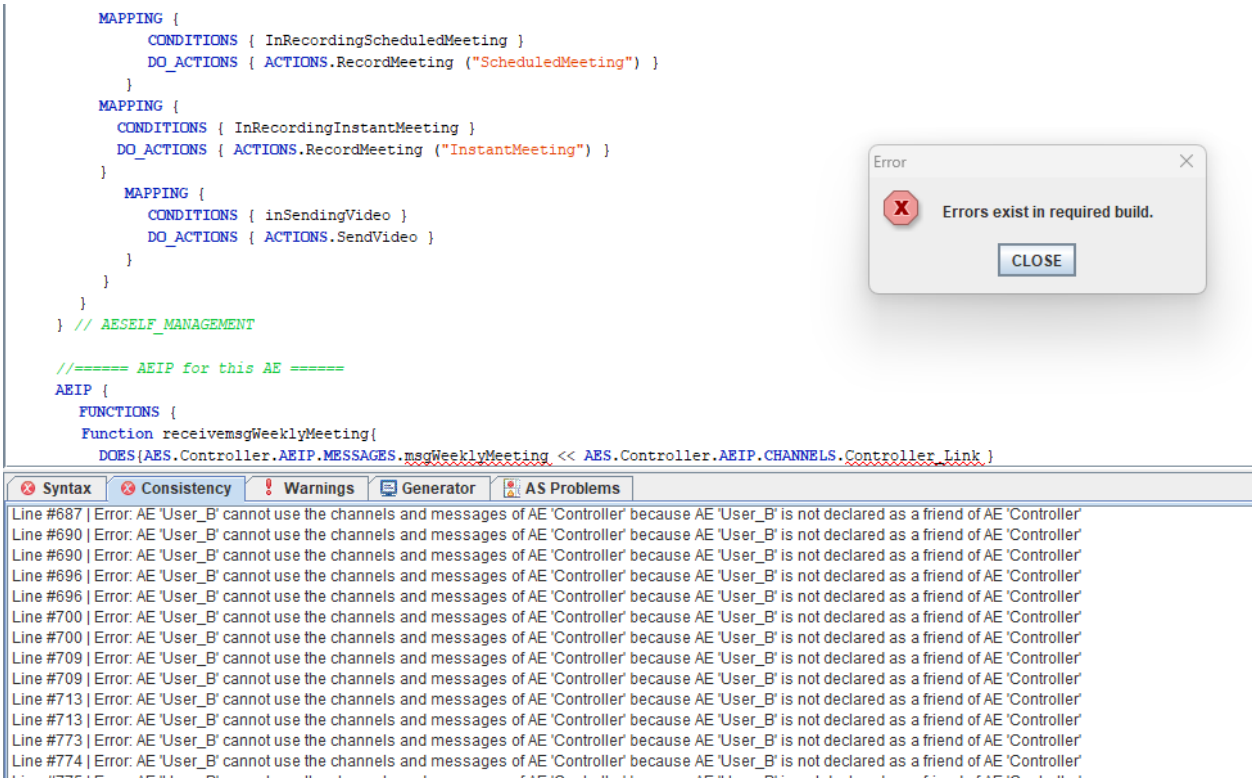

Figure 61: Accessibility error in Intent 7.

the process. The message session procedure is expressed for both scheduled and instant meeting types. The `Controller_link`, as a private channel between the Controller and the End users, transmits these messages, resulting in a simulation for a secure message interaction. At the same time, none of the AEs have access to the message unless they are defined as friends of the Controller as shown in Figure 60.

If the `FRIEND` list is not declared for autonomic elements that have interaction, the consistency error will occur, showing an accessibility error as depicted in Figure 61.

However, the definition of a `FRIEND` list is a static process. In case of adding new autonomic elements as friends to the list, the ASSL specification should regenerate the

```
// session messages to be received by User_A
MESSAGE msgScheduledMeetingBegin_A {
   SENDER { AES.Controller }
   RECEIVER { AES.User_A }
   MSG_TYPE { NEGOTIATION }
   BODY { BEGIN }
}
MESSAGE msgScheduledMeetingEnd_A {
   SENDER { AES.Controller }
   RECEIVER { AES.User_A }
   MSG_TYPE { NEGOTIATION }
   BODY { END }
}
MESSAGE msgInstantMeetingBegin_A {
   SENDER { AES.Controller }
   RECEIVER { AES.User_A }
   MSG_TYPE { NEGOTIATION }
   BODY { BEGIN }
}
MESSAGE msgInstantMeetingEnd_A {
   SENDER { AES.Controller }
   RECEIVER { AES.User_A }
   MSG_TYPE { NEGOTIATION }
   BODY { END }
}
```

```
// session messages to be received by User_B
MESSAGE msgScheduledMeetingBegin_B {
   SENDER { AES.Controller }
   RECEIVER { AES.User_B }
   MSG_TYPE { NEGOTIATION }
   BODY { BEGIN }
}
MESSAGE msgScheduledMeetingEnd_B {
   SENDER { AES.Controller }
   RECEIVER { AES.User_B }
   MSG_TYPE { NEGOTIATION }
   BODY { END }
}
MESSAGE msgInstantMeetingBegin_B {
   SENDER { AES.Controller }
   RECEIVER { AES.User_B }
   MSG_TYPE { NEGOTIATION }
   BODY { BEGIN }
}
MESSAGE msgInstantMeetingEnd_B {
   SENDER { AES.Controller }
   RECEIVER { AES.User_B }
   MSG_TYPE { NEGOTIATION }
   BODY { END }
}
```

Figure 62: Video conferencing policy session messages in the controller `AE` in Intent 7.

output. To explain more about the messaging sessions, after processing the meeting type in the Controller, the `sendBeginMsgs` function is called for each meeting type. Then `sendVideoConferenceMsg` is called to send a video conference message to the `Controller_link`. The `sendEndSessionMsgs` function is called to end the beginning sessions, and this procedure completes on the End users' sides through receiving these messages. Negotiation messages, related functions, and channel are depicted in Figure 62, and Figure 63, respectively.

On the End users' sides, the autonomic network turns into start meeting session states for both meeting types after the Controller sends the begin session messages for each meeting type. After this initiation state, the recording stage begins, where the video messages are transmitted based on the meeting type parameter by a function call to `receiveVideoConferenceMsg` followed by ending messages. The fluent to complete these transactions is `inSendingVideo` to call the `VideoPrepration` IMPL action to add any required coding, which is impossible to specify in the ASSL to the generated Java output. To show that the video is sent through this action, sending and receiving video conference

```
CHANNELS {
   CHANNEL Controller_Link {
      ACCEPTS { AEIP.MESSAGES.msgVideoConference,
         AEIP.MESSAGES.msgScheduledMeetingBegin_A, AEIP.MESSAGES.msgScheduledMeetingEnd_A,
         AEIP.MESSAGES.msgInstantMeetingBegin_A, AEIP.MESSAGES.msgInstantMeetingEnd_A,

         AEIP.MESSAGES.msgScheduledMeetingBegin_A, AEIP.MESSAGES.msgScheduledMeetingEnd_A,
         AEIP.MESSAGES.msgInstantMeetingBegin_B, AEIP.MESSAGES.msgInstantMeetingEnd_B
      }
      ACCESS { DIRECT }
      DIRECTION { INOUT }
   }
}

FUNCTIONS {
   FUNCTION sendVideoConferenceMsg {
      DOES { AEIP.MESSAGES.msgVideoConference >> AEIP.CHANNELS.Controller_Link   }
   }
   FUNCTION sendBeginMsgs {
      PARAMETERS { string MeetingType }
      DOES {
         IF MeetingType = "ScheduledMeeting" THEN
            AEIP.MESSAGES.msgScheduledMeetingBegin_A >> AEIP.CHANNELS.Controller_Link;
            AEIP.MESSAGES.msgScheduledMeetingBegin_A >> AEIP.CHANNELS.Controller_Link
         END
         ELSE
            IF MeetingType = "InstantMeeting" THEN
               AEIP.MESSAGES.msgInstantMeetingBegin_A >> AEIP.CHANNELS.Controller_Link;
               AEIP.MESSAGES.msgInstantMeetingBegin_B >> AEIP.CHANNELS.Controller_Link
         END
   END
      }
   }
   FUNCTION sendEndSessionMsgs {
      PARAMETERS { string MeetingType }
      DOES {
         IF MeetingType = "ScheduledMeeting" THEN
            AEIP.MESSAGES.msgScheduledMeetingEnd_A >> AEIP.CHANNELS.Controller_Link;
            AEIP.MESSAGES.msgScheduledMeetingEnd_A >> AEIP.CHANNELS.Controller_Link
         END
         ELSE
            IF MeetingType = "InstantMeeting" THEN
               AEIP.MESSAGES.msgInstantMeetingEnd_A >> AEIP.CHANNELS.Controller_Link;
               AEIP.MESSAGES.msgInstantMeetingEnd_B >> AEIP.CHANNELS.Controller_Link
            END
         END
      }
   }
```

Figure 63: Functions and channel in the controller for Intent 7.

message functions are called for each end user to connect them to the `ASIP` video conference simulated channel.

ASSL specification can bring about *Composability* and *Modularity* intent objectives, meanwhile, logical *Scalability* is another aspect of this specification language. Intent 7 and Intent 8 specifications are used to demonstrate this capability of ASSL. Intent 7 refers to requesting the video conference policy, and Intent 8 refers to creating a weekly meeting between the End users A and B. These intents can be specified by adding a self-configuring policy module to the video conferencing policy module without interfering with the whole structure of the autonomic network. Specifying a requesting state to process requests for video conferencing from both End users helps us to achieve the intent.

During requesting state, authentication as a security simulation is also expressed as an IMPL action with a boolean return type as depicted in Figure 65. In the current ASSL specification, abstracting the autonomic behaviors related to security is the fundamental logic of the implementation for security objectives.

3.3.8 INTENT 8

INTENT 8	
Intent :	*Create a video conference type for a weekly meeting.*
Solutions :	DC Networks
Intent User :	End Users
Intent Type :	Strategy Intents
Source :	[23]

Table 8: Specification/classification for INTENT 8

Another sample proof of the potential composability of intents is when we use the previous specification of Intent 7 to form Intent 8 by adding a self-configuring policy module to the Controller. See Table 8 for the specification/classification for this new intent. Since the Controller manages the video conference infrastructure, it can create recurring meetings if it transfers the autonomic network to the state of `InVideoConferenceCreation` as illustrated in Figure 66 compared to Figure 56 where both are the Controller self-management policies.

```
AS AutonomicNetwork {
  ASSELF_MANAGEMENT {

    SELF_CONFIGURING{
        FLUENT inRequestVideoConference_A {
          INITIATED_BY { EVENTS.TimeToRequest_A }     TERMINATED_BY { EVENTS.Request_AProcessed }
        }
        FLUENT inRequestVideoConference_B {
          INITIATED_BY { EVENTS.TimeToRequest_B }     TERMINATED_BY { EVENTS.Request_BProcessed }
        }
        MAPPING {
          CONDITIONS { inRequestVideoConference_A}
          DO_ACTIONS { ACTIONS.processRequest("User_A") }
        }
        MAPPING {
          CONDITIONS { inRequestVideoConference_B}
          DO_ACTIONS { ACTIONS.processRequest("User_B") }
        }
      }//SELF_CONFIGURING

    OTHER_POLICIES {
     POLICY VideoConferencing {
        FLUENT inProcessingVideoConference_A {
           INITIATED_BY { EVENTS.VideoConference_AReceived } TERMINATED_BY { EVENTS.VideoConference_AProcessed }
           }
        FLUENT inProcessingVideoConference_B {
          INITIATED_BY { EVENTS.VideoConference_BReceived }  TERMINATED_BY { EVENTS.VideoConference_BProcessed }
          }
        MAPPING {
          CONDITIONS { inProcessingVideoConference_A}
          DO_ACTIONS { ACTIONS.processVideoConference("User_A") }
        }
        MAPPING {
          CONDITIONS { inProcessingVideoConference_B}
          DO_ACTIONS { ACTIONS.processVideoConference("User_B") }
          }
        }//VideoConferencing
     }//OTHER_POLICIES
  } // ASSELF_MANAGEMENT
```

Figure 64: Self-configuring and other-policies in the `AS` tier in Intent 7

```
ACTION processRequest{
  PARAMETERS { string UserName  }
  GUARDS { ASSELF_MANAGEMENT.SELF_CONFIGURING.inRequestVideoConference_A OR
           ASSELF_MANAGEMENT.SELF_CONFIGURING.inRequestVideoConference_B}
  DOES {CALL IMPL ACTIONS.AuthenticateUser}
  TRIGGERS {
    IF UserName = "User_A" THEN EVENTS.Request_AProcessed   END
    ELSE
      IF UserName = "User_B" THEN EVENTS.Request_BProcessed END
    END
  }
}

ACTION IMPL AuthenticateUser{
  RETURNS{boolean}
}
```

Figure 65: Authentication `IMPL` action in Intent 7

```
AE Controller {

  VARS { boolean initiateWithA }

  AESELF_MANAGEMENT {

    SELF_CONFIGURING{
      FLUENT inVideoConferenceCreation {
        INITIATED_BY { EVENTS.TimeToCreateVideoConference }
        TERMINATED_BY { EVENTS.VideoConferenceCreated}
        }
      MAPPING {
        CONDITIONS { inVideoConferenceCreation }
        DO_ACTIONS { ACTIONS.CreateVideoConference }
        }
   }//SELF_CONFIGURING

    OTHER_POLICIES {
      POLICY VideoConferencing {
        FLUENT inStartingingVideo {
          INITIATED_BY { EVENTS.TimetoStartVideo }
          TERMINATED_BY { EVENTS.VideoStarted }
          }
        FLUENT inProcessingVideo {
          INITIATED_BY { EVENTS.VideoStarted }
          TERMINATED_BY { EVENTS.VideoProcessed }
          }
        MAPPING {
          CONDITIONS { inStartingingVideo }
          DO_ACTIONS { ACTIONS.StartVideo }
          }
        MAPPING {
          CONDITIONS { inProcessingVideo }
          DO_ACTIONS { ACTIONS.ProcessVideo }
          }
        }//VideoConferencing
      }//OTHER_POLICIES
  } // AESELF_MANAGEMENT
```

Figure 66: Controller policies for Intent 8.

However, in Intent 8 a self-configuring policy to create a weekly video conference is added.

Also, as shown in Figure 67, the packages created from the controller based on the specifications for Intent 7 and Intent 8 are similar except for those sections that an arrow point to them. These minor modifications only add to the functionality of Intent 7 without disrupting the previous expressions. For instance, `inRecurringMeeting` fluent and its related action called `ProcessRecurringMeeting`, with required message, events are added to form

Figure 67: Controller package comparison between Intent 7 and Intent 8.

the autonomic behavior related to the creation of a video conference under the self-configuring policy. This addition does not interfere with previous expressions, and new autonomic behavior can work with the previous model. This composability logic is similar on the user sides as shown in Figure 68.

As depicted in Figure 66, to inform the whole network of this weekly meeting, `msgWeeklyMeeting` transmits from the Controller to the End users through the Controller's private channel, and the End users can receive the message on their side when they call `receivemsgWeeklyMeeting` function in the `ProcessRecurringMeeting` action as shown in Figure 69. This action is the result of abstract formulation of an autonomic behaviour to create a recurring meeting for end users, since it does not include any details and only

Figure 68: User A package comparison between Intent 8 and Intent 7.

```
ACTION ProcessRecurringMeeting{
  GUARDS{AESELF_MANAGEMENT.SELF_CONFIGURING.inRecurringMeeting}
  DOES{ CALL AEIP.FUNCTIONS.receivemsgWeeklyMeeting
        }
 }
```

Figure 69: Action of users Intent 8.

focuses on what should be happened.

Logical scalability is possible if the specification allows for adding new autonomic elements as users without a negative effect on the logic and resources of the autonomic network.

The potential of *Scalability* intent objective is met through the hierarchical design of end-user definition as AEs, controlled by the Controller and division of responsibilities between them. Video conferencing procedures are divided into shared and individual basis tasks. The Controller's functionality focuses on the shared duties between the end users to respect video conferencing related policies. The end users take the local responsibilities, such as requesting video conferences. Therefore, this separation of tasks and hierarchical design results in the system's modularity on the one hand and allows the addition of other end users to the autonomic network without any concerns for the interference of their task to integrated policy specifications on the other hand.

3.3.9 INTENT 9

INTENT 9	
Intent :	*Request automatic life-cycle management of VM cloud resources.*
Solutions :	DC Networks
Intent User :	Cloud Administrator
Intent Type :	Cloud Resource Management Intent
Source :	[23]

Table 9: Specification/classification for INTENT 9

Our next selected intent is related to cloud computing resource management. See Table 9 for the specifications/classification of this intent. Cloud computing comprises physical infrastructure and virtual instances of the software to create complex and scalable networks. In particular, data centers are the most critical users of this technology since virtualization can optimize physical hardware and help companies to make an enormous amount of network resources without actually adding more physical networking devices. Due to the remarkable capacity of this technology, it is critical to have intent-based management to reduce the management complexity caused by cloud computing. Intent-based management can encapsulate the complex configuration of the low-level devices, which are the infrastructure for the virtual machines. In addition, expanding the number of virtual instances is essential due to data centers' growing demand for new networking resources.

These constant adjustments require autonomic management to check the life cycle of the instances to handle the modifications without any interruptions in the network performance. Since intent-based networking focuses on the network goal rather than the procedures to achieve the goal, the flexibility of network management increases because adding new virtual instances will not interfere with the network performance. In any case of issues, the network can handle its problems through autonomic behavior. Cloud administrators are the users of the intent that request an automatic life-cycle of virtual resources to meet the *Cloud Resource Management* intent type.

In general, the life cycle of a virtual instance contains four stages: (1) provisioning the resources, (2) staging to initiate any preliminary tasks before starting the instances, (3) running to boot the instances, and (4) repairing to give the possibility of stopping, suspension, and termination of the virtual instances. The stages are the states to form the control loop for the appropriate autonomic behavior. The life-cycle states are shared among the virtual instances. Therefore, a central conductor autonomic node like a Controller can operate the autonomic behavior in the shared control loop. Other virtual instances can be the dependent autonomic elements to this Controller to request the life-cycle or other individual activities.

The implementation design for this scenario consists of two steps. First, the `AS` tier specifies a general self-configuration state for the whole network rules, the `ASIP` tier for the public interactions among the network, and the `AE` called the Controller to provide the life-cycle procedure is designed. The encapsulation of any technical low-level configuration details in this hierarchy refers to the abstract formulation of the autonomic behaviors by ASSL. The autonomic system can perform the autonomic behavior related to the life cycle under three self-chop policies: self-configuring to specify life-cycle fluents, self-healing to prepare a repair state to prepare the possibility of recovering states for the instances such as stopping, suspending, or terminating a VM instance, and self-scheduling to record the whole life-cycle. Having all the policies to achieve the intent demonstrates the declarative outcome of ASSL because fluents are the finite states for the autonomic node to be mapped

```
ACTIONS{
  ACTION Manage{
    GUARDS{ASSELF_MANAGEMENT.SELF_CONFIGURING.inManaging}
    DOES{CALL ASIP.FUNCTIONS.sendStartLifeCycleMsg;
         CALL ASIP.FUNCTIONS.receiveStartLifeCycleMsg}
    TRIGGERS{EVENTS.Managingdone}
  }
}
```

Figure 70: AS self-configuring policy for Intent 9

to actions. The actions also perform their activities by calling functions to transfer messages and other actions; however, the detailed procedure of achieving the intent is encapsulated. Then by the *Composability* and potential *Scalability* intent objectives offered by ASSL, a virtual machine instance is created as the dependant autonomic element of the Controller. The dependent devices can express the network state where they request the autonomic life cycle. As a result, adding a state to the whole logic does not interfere with the previous tasks done by the Controller. AS tier contains the self-configuring policy to operate a general management action called `Manage` accessible for the entire network by transmitting the `msgStartLifeCycle` designed in the `ASIP` tier through abstract `Public_link`. Figure 70 depicts the action in the `AS` tier, and Figure 71 depicts the functions for transferring messages in the `ASIP` tier.

The separation of `AS` from `AE` tier based on the essence of the rules divided into public and private demonstrate the local management characteristics of the ASSL. This means that the AS-specified regulations provide a general autonomic behavior. In contrast, the `AE` regulations are the means for local management of the `AE`, where the rules are expressed individually and not shared among all the AEs.

In the `AE` tier, there are three policies defined. First, self-configuring policy to handle `InProvisioning`, `InStaging`, `InRunning` fluents that guide the autonomic control loop to actions for `AllocateResources` to allocate resources for the VM instances, `StartBooting` to initiate the VM instances for staging state, and `RunInstances` to send a message related to the running of the VM instances called `msgRun`. Figure 72 shows the implementation of

```
ASIP{
  MESSAGES{
      MESSAGE msgStartLifeCycle{
        SENDER { ANY }
        RECEIVER { ANY }
        PRIORITY { 1 }
        MSG_TYPE {TEXT }
        BODY { "Start LIFECYCLE of the VMs " }}
        }
  CHANNELS {
    CHANNEL Public_Link {
      ACCEPTS { ASIP.MESSAGES.msgStartLifeCycle}
      ACCESS { SEQUENTIAL }
      DIRECTION { INOUT } }
  }
  FUNCTIONS {
    FUNCTION sendStartLifeCycleMsg {
      DOES { ASIP.MESSAGES.msgStartLifeCycle >> ASIP.CHANNELS.Public_Link }
    }
    FUNCTION receiveStartLifeCycleMsg {
      DOES { ASIP.MESSAGES.msgStartLifeCycle << ASIP.CHANNELS.Public_Link }
    }

  }
}//ASIP
```

Figure 71: `ASIP` tier for Intent 9.

```
SELF_CONFIGURING{
      FLUENT InProvisioning{INITIATED_BY{EVENTS.TimeToStartLifeCycle} TERMINATED_BY{EVENTS.ProvisioningDone}}
      FLUENT InStaging{INITIATED_BY{EVENTS.ProvisioningDone} TERMINATED_BY{EVENTS.StagingDone}}
      FLUENT InRunning{INITIATED_BY{EVENTS.StagingDone} TERMINATED_BY{EVENTS.RunningDone}}
      MAPPING{CONDITIONS{InProvisioning} DO_ACTIONS{ACTIONS.AllocateResources}}
      MAPPING{CONDITIONS{InStaging} DO_ACTIONS{ACTIONS.StartBooting}}
      MAPPING{CONDITIONS{InRunning} DO_ACTIONS{ACTIONS.RunInstances}}
}//SELF_CONFIGURING
```

Figure 72: Controller self-configuring policy for Intent 9.

the fluents in the Self-configuring policy block and their corresponding actions.

Secondly, the Self-healing policy, as seen in Figure 74, takes the responsibility of modifying the VM instances by moving the autonomic network to `InRepairing` state where there is a possibility of transmission for three messages regarding stopping, suspension, and termination of the VM instances. Finally, the Self-scheduling policy defined under other policies expresses the fluent `inLifeCycle` where the recording of the VM instances' life-cycle is simulated with its action.

Figure 75 shows `Repair` action transferring the specified messages by calling the

```
ACTION AllocateResources{
   GUARDS { AESELF_MANAGEMENT.SELF_CONFIGURING.InProvisioning}
   DOES{CALL IMPL ACTIONS.allocation}
   TRIGGERS{EVENTS.ProvisioningDone}}

ACTION IMPL allocation{}

ACTION StartBooting{
   GUARDS { AESELF_MANAGEMENT.SELF_CONFIGURING.InStaging}
   DOES{CALL IMPL ACTIONS.Boot}
   TRIGGERS{EVENTS.StagingDone}
}
ACTION IMPL Boot{
   TRIGGERS{EVENTS.StagingDone}
}

ACTION RunInstances{
   GUARDS { AESELF_MANAGEMENT.SELF_CONFIGURING.InRunning}
   DOES{CALL AEIP.FUNCTIONS.sendRunMsg}
   TRIGGERS{EVENTS.RunningDone, EVENTS.RunningStates}

}
```

Figure 73: Controller actions for self-configuring policy for Intent 9.

```
SELF_HEALING{
      FLUENT InRepairing{INITIATED_BY{EVENTS.RunningStates} TERMINATED_BY{EVENTS.RepairingDone}}
      MAPPING{CONDITIONS{InRepairing} DO_ACTIONS{ACTIONS.Repair}}
  }//SELF_HEALING

OTHER_POLICIES {
     POLICY SELF_SECHEDULING{
      FLUENT inLifeCycle{INITIATED_BY{EVENTS.TimeToRecordLifeCycle} TERMINATED_BY{EVENTS.LifeCycleRecorded}}
      MAPPING{CONDITIONS{inLifeCycle} DO_ACTIONS{ACTIONS.RecordLifeCycle}}
    }//SELF_SECHEDULING
```

Figure 74: Controller self-healing and other-policies for Intent 9.

```
ACTION Repair{
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InRepairing}
  DOES{CALL AEIP.FUNCTIONS.sendStopMsg;
       CALL AEIP.FUNCTIONS.sendSuspendMsg;
       CALL AEIP.FUNCTIONS.sendTerminateMsg}
  TRIGGERS{EVENTS.RepairingDone}
}

ACTION RecordLifeCycle{
  GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.SELF_SECHEDULING.inLifeCycle}
  DOES{CALL AEIP.FUNCTIONS.sendLifCycleMsg}
  TRIGGERS{EVENTS.LifeCycleRecorded}
}
```

Figure 75: Controller actions for self-healing and other-policies for Intent 9.

```
SELF_CONFIGURING{
  FLUENT InRunningVm{INITIATED_BY{EVENTS.TimeToRunVM} TERMINATED_BY{EVENTS.VMstarted}}
  FLUENT InRequestLifeCycle{INITIATED_BY{EVENTS.VMstarted} TERMINATED_BY{EVENTS.LifeCycleRequested}}
  MAPPING{CONDITIONS{InRunningVm} DO_ACTIONS{ACTIONS.runVM}}
  MAPPING{CONDITIONS{InRequestLifeCycle} DO_ACTIONS{ACTIONS.RequestLifeCycle}}
  }//SELF_CONFIGURING
```

Figure 76: Self-configuring policy for Intent 9.

`sendStopMsg`, `sendSuspendMsg`, `sendTerminateMsg` unctions.

The composability feature of ASSL specification allows the addition of VM instances as autonomic elements under the `AE` tier. Also, other VM instances can be added to the current `AE` tier with a similar design resulting from the scalability potential for the ASSL. However, due to the static structure of the grouping architecture in the current version of the ASSL, the additional VM instances should be added statically to the `ASARCHITECTURE` section, and the output should be regenerated. In this scenario, one VM instance is the dependent `AE` to the Controller. Its Self-configuring policy includes two fluents: `InRunningVm` to initiate the VM instance through its action `runVM`, and `InRequestLifeCycle` to receive the life-cycle of the VM instance with its action `RequestLifeCycle` that calls function `receiveLifeCycleMsg`. Figure 76, and Figure 77 illustrate the procedures, and action of the self-configuring policy of the `VM_INSTANCE AE`.

Figure 78 shows the self-healing policy for the `VM_INSTANCE AE` that leads the

```
ACTION RequestLifeCycle{
  GUARDS { AESELF_MANAGEMENT.SELF_CONFIGURING.InRequestLifeCycle}
  DOES{CALL AEIP.FUNCTIONS.receiveLifeCycleMsg}
  TRIGGERS{EVENTS.LifeCycleRequested}
  }
```

Figure 77: Action for self-configuring policy for Intent 9.

```
SELF_HEALING{
  FLUENT InStateSimulation{INITIATED_BY{EVENTS.TimeToStateSimulation} TERMINATED_BY{EVENTS.StateSimulated}}
  MAPPING{CONDITIONS{InStateSimulation} DO_ACTIONS{ACTIONS.modifyState}}
  }//SELF_HEALING
```

Figure 78: Self-healing policy for Intent 9.

```
ACTION modifyState{
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.InStateSimulation}
  DOES{
        CALL AEIP.FUNCTIONS.receiveStopMsg;
        CALL AEIP.MANAGED_ELEMENTS.VM_INSTANCE_Interface.stop;
        CALL AEIP.FUNCTIONS.receiveSuspendMsg;
        CALL AEIP.MANAGED_ELEMENTS.VM_INSTANCE_Interface.suspend;
        CALL AEIP.FUNCTIONS.receiveTerminateMsg;
        CALL AEIP.MANAGED_ELEMENTS.VM_INSTANCE_Interface.terminate
        }
  TRIGGERS{EVENTS.StateSimulated}
}
```

Figure 79: Action for self-healing policy for Intent 9.

system to a `InStateSimulation` fluent where there is a simulation of VM state modification through the action `modifyState`. During this action, messages for stopping, suspension, and termination of the VM are received by Function calls, `receiveStopMsg`, `receiveSuspendMsg`, `receiveTerminateMsg`, respectively.

Also, as seen in Figure 79, after each function call, there is another call to the corresponding managed element defined as `VM_INSTANCE_Interface`, and shown in Figure 80, which connects the ASSL specification to the actual VM instance system. This connection in the current version of the ASSL is an abstract simulation, and its consistency can be verified by the consistency checker of the ASSL compiler.

```
MANAGED_ELEMENTS{
  MANAGED_ELEMENT VM_INSTANCE_Interface{
    INTERFACE_FUNCTION suspend{}
    INTERFACE_FUNCTION stop{}
    INTERFACE_FUNCTION terminate{}
  }
}
```

Figure 80: Interface for the `VM_INSTANCE` managed element for Intent 9.

3.3.10 INTENT 10

INTENT 10	
Intent :	*Inspect traffic for the dorm!*
Solutions :	Enterprise Networks
Intent User :	Application Developer
Intent Type :	Operational Task Intent
Source :	[17]

Table 10: Specification/classification for INTENT 10

Many intent classifications [23] include monitoring policies mainly in strategy intents, and also these policies are also crucial for network service and underly network service intents. In practice, Network administrators or operators use monitoring tools under flow-based and deep packet inspection tools to do the monitoring. SNMP configuration, and open-source monitoring software like Wireshark[43] are examples of monitoring tools. These tools require a technician aware of networking knowledge for management, which makes the monitoring process not autonomic. Therefore, creating autonomic behaviors to use these tools or systems to make the process autonomic is beneficial to many networks because monitoring is a continual task that alleviates extra failure expenditures as it tracks the network and detects any abnormal behavior to prevent network down times. Also, Intent 10 is used to observe network traffic usage for planning network strategies. A monitoring tool or software can take the responsibility of monitoring, and an autonomic behavior can be specified to express monitoring properties by controlling these devices. For instance, Intent 10 determines this autonomic behavior for a location which is the dormitory.

```
AE Controller{
 AESELF_MANAGEMENT{
  OTHER_POLICIES{
    POLICY MONITORING{
      FLUENT inStartMonitoringTool{
          INITIATED_BY{EVENTS.TimeToRunMonitoringTool}
          TERMINATED_BY{EVENTS.MonitoringToolStarted}
      }//FLUENT

      FLUENT inDataSourceSelection{
          INITIATED_BY{EVENTS.TimeToRunSelectDataSource}
          TERMINATED_BY{EVENTS.DataSourceSelected}
          }//FLUENT


     FLUENT inTrafficInspection{
          INITIATED_BY{EVENTS.TimeToInspectTraffic}
          TERMINATED_BY{EVENTS.TrafficInspected}
                              }//FLUENT

      MAPPING {
            CONDITIONS { inStartMonitoringTool }
            DO_ACTIONS { ACTIONS.StartMonitoringTool }
             }

      MAPPING {
            CONDITIONS { inDataSourceSelection }
            DO_ACTIONS { ACTIONS.SelectDataSource }
               }

      MAPPING {
          CONDITIONS { inTrafficInspection }
          DO_ACTIONS { ACTIONS.InspectTraffic }
                     }

    }//MONITORING
    }//OTHER_POLICIES
  }//AESELF_MANAGEMENT
```

Figure 81: Controller `OTHER-POLICIES` named as monitoring policy for Intent 10.

The abstract formulation of this intent with ASSL results in `inStartMonitoringTool` in the Controller when `StartMonitoringTool` actions performed without specifying any parametric details as the fluent is shown in Figure 81, and the action is shown in Figure 82.

The declarative outcome of Intent 10 with ASSL specification is to define a monitoring policy conducting the system to the desired autonomic behavior to reduce the need for

```
ACTIONS{
    ACTION StartMonitoringTool{
        GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.MONITORING.inStartMonitoringTool }
        DOES{
          CALL AEIP.MANAGED_ELEMENTS.MonitoringTool.InitiateProgram
            }
        TRIGGERS { EVENTS.MonitoringToolStarted}  }
    ACTION SelectDataSource{
        parameters{string DataSourceLoc}
        GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.MONITORING.inDataSourceSelection }
        DOES{
          IF DataSourceLoc="DORM" THEN CALL ASIP.FUNCTIONS.SendMonitoringMsg END
          }
        TRIGGERS{EVENTS.DataSourceSelected}}
    ACTION InspectTraffic{
        GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.MONITORING.inTrafficInspection }
        DOES{
          CALL ACTIONS.InspectTrafficType("ingress");
          CALL ACTIONS.InspectTrafficType("egress")
        }
        TRIGGERS{EVENTS.TrafficInspected}}

    ACTION InspectTrafficType{
      parameters{string TrafficType}
      DOES{
        IF TrafficType="ingress" THEN Call AEIP.MANAGED_ELEMENTS.MonitoringTool.IngressTrafficChecker END
         ELSE
        IF TrafficType="egress" THEN Call AEIP.MANAGED_ELEMENTS.MonitoringTool.EgressTrafficChecker END
        END;
         CALL ASIP.FUNCTIONS.ReceiveMonitoringMsg
      }
      TRIGGERS{EVENTS.monitoringDone}
    } }//ACTIONS
```

Figure 82: Action in the controller for Intent 10.

a network operator to manage the monitoring tools in more detail compared to abstract formulation. Thus, `SelectDataSource`, and `InspectTraffic` as shown in Figure 82 expressed with parametric details to support more complex conditions.

Furthermore, to formulate the monitoring procedure in a high-level abstract policy, the system should self-configure the autonomic network and consider a monitoring tool as its managed resource. This monitoring tool can be called during the monitoring phase, while other required circumstances to monitor traffic can form different stages that the network should pass. From this high-level perspective, there is no need to directly run the monitoring configuration on all of the systems connected to a network and estimate the procedures for monitoring tools in detail for all the nodes. ASSL specification meets this condition to view the autonomic network besides the states it can have to achieve a policy. The

main component of this specification is the Controller as the autonomic element to manage the monitoring phase. The hierarchy of Intent 10 comprises a self-configuring policy in its `AS` tier as the general step of self-configuration of the network and the `ASIP` tier to transmit a monitoring message through its public channel. Also, the Controller forms three states for the monitoring policy under the `OTHER-POLICIES` block. The abstract phases to define monitoring are fluents called `inStartMonitoringTool` to start the initiation of the monitoring tool and `inDataSourceSelection` to check the data source location where its traffic should be monitored where the location parameter is Dorm in this case, and `inTrafficInspection` to inspect the traffic. As the managed element, the monitoring tool defines three interface functions to run the monitoring program and check ingress and egress traffic separately. If it is egress or ingress, the traffic type is defined as a parameter for `InspectTrafficType`, and this action is called by the `InspectTraffic` action. All the actions are shown in Figure 82.

3.3.11 INTENT 11

INTENT 11	
Intent :	*Manage this network.*
Solutions :	Enterprise Networks
Intent User :	Enterprise Administrator
Intent Type :	Network Intent
Source :	[23]

Table 11: Specification/classification for INTENT 11

The practice of overseeing and controlling a computer network's performance and administration is known as network management. This entails responsibilities including keeping an eye out for difficulties on the network, resolving the issues and configuring the network while it is safe and adheres to all applicable regulations. Policies forming network management goals help the network to preserve its dependability. In particular, in terms of having autonomicity, the autonomic network is capable of performing the appropriate actions based on the network status. Although, enterprise administrators can use Intent 11

in enterprise networks to reduce the complexity of network management in the large-scale network, this intent can be beneficial to smaller networks, too. Despite the scale and complexity of a network, all networks need management. Therefore, covering Intent 11 in network management is vital since it can boost performance on the one hand and perform time-consuming tasks such as troubleshooting on the other hand with minimized human interaction. The flexibility to select which policy to apply as a self-management policy to the network can be defined based on the network scale and business requirements of that particular network. Thus, besides enterprise networks and their administrators, other technical users can use Intent 11 in different network solutions or scope, based on [23]. Since the management goal is the most vital and fundamental goal of an autonomic network, this concept is helpful for all three network solutions, Carrier, DC, and Enterprise network. Each solution can adjust the policies based on their particular scope or scale.

In managing and administering a network, several actors play roles: Network administrators, network management software, protocols, and network devices. By reducing the human role in management procedure, policies should be formed to create the structure of desirable autonomic behavior according to the intent. Self-chop policies: self-configuration for configuring the network devices, self-healing for resolving a potential issue in the network, self-optimization to enhance the performance, and other policies to control the different aspects of the system are among the applicable policies in network management. A controller device can take responsibility for the system's main reactions and contributions. Therefore, this central management can handle the main instructions of the policies. The centralized control of the autonomic behaviors facilitates the controlling procedure of the policies as the primary policies are in one autonomic node, and their related instructions are sent to the dependent hosts as other autonomic nodes. Each Host can have its self-regulation properties derived from the primary management behavior.

ASSL architecture for Intent 11 consists of `AS` tier for a general `ALARM` process to handle public notification among the entire autonomic domain. `ASIP` tier provides the messaging interactions of notification procedures. The `AES` tier has one Controller `AE` to

```
ACTION troubleshoot{
  GUARDS { AESELF_MANAGEMENT.SELF_HEALING.inTroubleshoot}
  DOES {  CALL AEIP.FUNCTIONS.sendTroubleshootMsg }
  TRIGGERS { EVENTS.troubleshootDone}
  }
```

Figure 83: Troubleshooting action for self-healing policy for Intent 11.

```
ACTION Tshoot {
  GUARDS { AESELF_MANAGEMENT.OTHER_POLICIES.SELF_REGULATION.inTshoot }
  DOES{CALL AEIP.FUNCTIONS.receiveTroubleshootMsg;
       CALL IMPL ACTIONS.UpdateSoftware }
  TRIGGERS{EVENTS.TshootDone}
}

ACTION IMPL UpdateSoftware{ returns {Boolean}}
```

Figure 84: Action for self-regulation policy for Intent 11.

control self-management policies, and the Host `AE` is the dependent node to the Controller, which can be any device in the network that should be managed. Under the Controller AE, there are three self-chop policies defined for different aspects of the management procedure: `Self_configuring`, `Self_healing`, and `Self_optimizing`. `Self_protection` is not considered in this implementation directly because security is a low-level mechanism resolved by ANIMA's work [28]. Different intent types are covered in this specification based on [23]. `Self_configuring` policy refers to the *Network and Underlay Network Intent* and *Network and Underlay Network Service Intent* types. `Self_optimizing` refers to strategy intent type since it includes traffic engineering through fluents called `inHighTraffic`, `inTransferTraffic` as shown in Figure 87. Operational task intent types, such as updating the system's software, are covered in the Host as part of the system's reactions when the network encounters a problem, and it should start troubleshooting. The action `Tshoot` is shown in Figure 84.

Each self-management policy, such as self-configuring related to the configuration of the autonomic network, has a significant amount of implementation details in low-level concepts from vendor-based commands designed for different devices, like switches

and routers. Nevertheless, ASSL encapsulates the details and complexity of low-level configurations. Intent 11 in the context of ASSL can demonstrate both *Abstract Formulation* and *Declarative Outcome Expression* intent objectives. These two objectives refer to the capability of ASSL to express the intent in a high-level language. The abstract formulation properties of ASSL express three self-chop policies under the `AS` tier to maintain the high performance for the autonomic network without mentioning the exact details. The autonomic network should configure itself, resolve problems, and optimize its performance in a network resource such as bandwidth by transferring extra traffic. ASSL can simulate these autonomic behaviors by fluents in the Host: `inConfig`, `inTshoot`, and `inTransfer` under the self-regulation policy as shown in Figure 85. There are no precise details about a metric or variable in this part of the expression.

The declarative outcome is attained when there is a deeper look with more concrete details from the intent. For instance, Intent 11 has fluents to specify network states besides the system's actions to each state in the formation of the Controller's autonomic behavior. Some of these fluents result in actions consisting of more details, such as network metrics or variables, to observe and modify the system's behavior based on the specific variables defined in the `AE` tier shown in Figure 86.

The fluents consist of three self-chop policies to form the autonomic behavior based on Intent 11 is shown in Figure 87, and then using the declared variables in one of the actions called `recomputeSysParams`, and `IMPL` action called `generateTrafficLoad` are depicted in Figure 88 as examples to show *Declarative Outcome Expression* intent objective by ASSL expression. It is noticeable that due to the simulation-only nature of the ASSL we injected metrics manually when it was needed.

In terms of distributed and local behavior management, Intent 11 has a hierarchical architecture with AS, ASIP, and AES. `NotificationManagement` is an `AS` policy to provide a general notification for the system named as `ALARM` distributed among the entire network as shown in Figure 89. Unlike the autonomic policy similar to `self_regulation` in the Host as depicted in Figure 85 is the Host's local policy and not accessible by the whole network.

```
AE Host{
  AESELF_MANAGEMENT{
    OTHER_POLICIES{
      POLICY SELF_REGULATION{
        FLUENT inConfig{
            INITIATED_BY{EVENTS.TimeToStart}
            TERMINATED_BY{EVENTS.started}}
        FLUENT inTshoot{
            INITIATED_BY{EVENTS.TimeToTshoot}
            TERMINATED_BY{EVENTS.TshootDone}}
        FLUENT inTransfer{
            INITIATED_BY{EVENTS.TimeToTransfer}
            TERMINATED_BY{EVENTS.TransferDone}}

        MAPPING {
          CONDITIONS { inConfig }
          DO_ACTIONS { ACTIONS.config}
        }
        MAPPING {
           CONDITIONS {inTshoot}
           DO_ACTIONS { ACTIONS.Tshoot }
         }
        MAPPING {
           CONDITIONS { inTransfer}
           DO_ACTIONS { ACTIONS.Transfer}
         }
      }
    }
  }//AESELF_MANAGEMENT
```

Figure 85: AE host for Intent 11.

```
AS AutonomicNetwork{

  VARS{
      decimal dtr; //amount of data
      decimal Tduration; //time
      decimal bnd  //bandwidth
      }
```

Figure 86: AS variables for Intent 11.

```
AE Controller{
   AESELF_MANAGEMENT {
      SELF_CONFIGURING{
        FLUENT inConfiguration {
          INITIATED_BY { EVENTS.TimeToConfigure }
          TERMINATED_BY { EVENTS.configurationDone,EVENTS.ConfigurationFailed} }
        MAPPING{
           CONDITIONS{inConfiguration}
           DO_ACTIONS{ACTIONS.Configure}}
      }//SELF_CONFIGURING
      SELF_HEALING{
        FLUENT inTroubleshoot{
          INITIATED_BY { EVENTS.ConfigurationFailed }
          TERMINATED_BY { EVENTS.troubleshootDone}  }
        MAPPING{
           CONDITIONS{inTroubleshoot}
           DO_ACTIONS{ACTIONS.troubleshoot}}
      }//SELF_HEALING
      SELF_OPTIMIZING{
        FLUENT inHighTraffic {
          INITIATED_BY { EVENTS.highTraffic }
          TERMINATED_BY { EVENTS.normalTraffic }  }
        FLUENT inTransferTraffic {
          INITIATED_BY { EVENTS.transferRequested }
          TERMINATED_BY { EVENTS.transferEnd }  }
        MAPPING {
          CONDITIONS { inHighTraffic }
          DO_ACTIONS { ACTIONS.startTransmission, ACTIONS.recomputeSysParams }
        }
        MAPPING {
          CONDITIONS { inTransferTraffic }
          DO_ACTIONS { ACTIONS.transferTraffic }}
      }//SELF_OPTIMIZING
    }//AESELF_MANAGEMENT
```

Figure 87: AE controller for Intent 11.

```
ACTION recomputeSysParams{
   GUARDS { AESELF_MANAGEMENT.SELF_OPTIMIZING.inHighTraffic}
   DOES {
        IF AutonomicNetwork.dtr<10 THEN
        IF AutonomicNetwork.dtr>8 THEN
        IF AutonomicNetwork.Tduration= 1 THEN
        AutonomicNetwork.bnd=AutonomicNetwork.dtr/AutonomicNetwork.Tduration  END
             END
          END}
   TRIGGERS { EVENTS.normalTraffic}
       }

 ACTION IMPL generateTrafficLoad{
   RETURNS{decimal}
 }
```

Figure 88: Recompute action for Intent 11.

```
ASSELF_MANAGEMENT {
  OTHER_POLICIES{
    POLICY NotificationManagement{
      FLUENT ALARM{
        INITIATED_BY{EVENTS.TimeToNotify}
        TERMINATED_BY{EVENTS.NotificationSent}}
      MAPPING{
        CONDITIONS{ALARM}
        DO_ACTIONS{ACTIONS.Notify}}
  }}//OTHER_POLICIES
}//ASSELF_MANAGEMENT
```

Figure 89: AS notification management policy for Intent 11.

Another example of local and distributed network management behaviors can be outlined through fluents related to troubleshooting. Troublshoot state expressed as a self-healing feature of the Controller fluent called `inTroubleshoot` shown in Figure 87 guides the autonomic network to an action called `troubleshoot`. This action sends a function `sendTroubleshootMsg` to the `Private_Link` as the abstract communication channel between the AEs. On the other hand, The Host expresses a `inTshoot` fluent depicted in Figure 85, which is similar to the troubleshooting state; however, the functionality of its corresponding action `Tshoot` can be localized. Thus, besides receiving the troubleshooting message through `receiveTroubleshootMsg`, it can perform other tasks such as `UpdateSoftware`, which is only specified in the Host but not the Controller. The actions related to this description `troubleshoot`, and `Tshoot` are shown in Figure 83, and Figure 84.

The architecture of the intent specification is shown in Figure 90. This architecture illustrates the dependencies between the Controller and the Host, where the Controller is the central management autonomic node. This means that only devices listed as a friend of the Controller can have access to the policies of the Controller, which results in an abstract security level. Furthermore, with this architecture, the local and distributed management policies accessibility can be distinguished.

Intent 11 provides the *Composability* intent objective since different autonomic behaviors forming policies can accompany each other without interfering with each other's tasks. These

```
ASARCHITECTURE {
   AELIST {AES.Host, AES.Controller}
   DIRECT_DEPENDENCIES { DEPENDENCY AES.Host { AES.Controller }}
   TRANSITIVE_DEPENDENCIES { DEPENDENCY AES.Controller  {AES.Host }}
   GROUPS {
      GROUP management {
         MEMBERS { AES.Host, AES.Controller }
         COUNCIL { AES.Controller}}} }//ASARCHITECTURE
```

Figure 90: AS architecture for Intent 11.

policies can handle different aspects of network management, while there is no limitation to adding more policies to the autonomic network specified by ASSL. An example of multiple policies working together is depicted in Figure 87. ASSL can potentially add more hosts to its architecture to provide the *Scalability* intent objective in terms of adding resources to the system without interfering with the main intent expression at an abstract level. However, the shortcoming of ASSL in creating a dynamic architecture causes the regeneration of code after adding a new autonomic element. Regarding monitoring capabilities, ASSL can self-monitor its metrics and other autonomic behaviors through features like `ASSLMONITOR` package [42]. In case of any modifications in the observations, the autonomic network is conducted to the proper reaction. This internal monitoring is also achieved through analyzing the `GUARDS` as conditions to be checked before performing an action. However, due to the lacking of an actual connection between the managed elements and the ASSL specification, these observations are internal to check formal statements declared as intents inside the ASSL.

3.4 Summary

In Chapter 3, the intent objectives as stated Section 3.1 are suggested to solve the lack of an integrated intent definition under the context of the autonomic network. In addition, the possibility of using ASSL is scrutinized in Section 3.2. Finally, how ASSL can be used to express intents under the context of autonomic networks are studied by specifying concrete

examples based on [23] in Section 3.3. Also, this section assesses the features of ASSL specifications that match with intent objectives.

Chapter 4

Evaluation

In Chapter 2, we investigated current technologies in intent-based network management, and we identified two main problems: (1) There is a lack of a comprehensive definition of *Intent* that integrates the different perspectives as expressed by different research groups on the topic, and (2) there is a solution gap for an *Intent* expression method/language and an associated intent deployment solution within the context of *Autonomic Networks*. In Chapter 3, we defined a set of intent objectives to solve the first problem (Section 3.1) and suggested the Autonomic System Specification Language (ASSL) as the solution to the second problem. Then we specified different concrete examples in ASSL, based on the intent classification in RFC 9316 [23], to demonstrate how this formal language can express a wide variety of network management intents (Section 3.3). It is important to note that we do not compare our solution with other technologies since each of them focused on narrow and particular criteria of intent, whereas we are focusing on breadth of application as a solution. The exception to this narrow focus was NEMO [30]. However, there was insufficient documentation available for this technology to permit us to formulate a meaningful comparison.

In this chapter, we proceed to evaluate the appropriateness of ASSL to express and deploy intents based on the examples that we gave in Section 3.3 with regards to each of the objectives that we have identified in Section 3.1.

4.1 Intent-Based Autonomic Networking Using ASSL

The Autonomic System Specification Language (ASSL) is a formal language used to specify the behavior and requirements of autonomic systems. Considering an autonomic network as an autonomic system, ASSL can express the goals, objectives, internal organization, and behavior of the autonomic network at an abstract level. It can also outline the limitations and needs that must be satisfied for the network to operate correctly and the laws and regulations that direct how the system should be run. The main objective of employing ASSL is to make it possible to develop and implement autonomic systems that are more dependable, efficient, and adaptable to changing circumstances. ASSL can be used to define the actions and specifications of an autonomic network. ASSL is an appropriate solution to autonomic networking driven by intents since it makes it possible to build and deploy more dependable, efficient, and responsive networks to changing conditions. ASSL can express self-chop, and other policies to simulate the self-management of an autonomic network by guiding the network to take proper actions during different network states, such as failure. In addition, there is a possibility to define other policies if an intent contains policies rather than the four self-chop policies. By defining these policies in the ASSL specification, it is possible to ensure that the autonomic network can adapt to changing conditions and maintain its reliability, availability, and performance based on the intent objectives. In general, ASSL can specify intent as self-chop or other policies under the autonomic network.

Self-configuring policies specify the steps an autonomic network should take to adjust itself automatically in response to changing circumstances or demands. A self-configuring policy is mainly about the configuration states; for instance, when the system should add extra resources (such as bandwidth or processing power as network metrics) to meet the demand for those resources when it surpasses a specific limit. Some examples of self-configuring policy are depicted in Figure 7 and Figure 9, to model general device configuration, and QoS configuration steps in Intent 1. Figure 36 in Intent 3, Figure 55 and Figure 64 in Intent 7, Figure 66 in Intent 8, Figure 72, and Figure 76 in Intent 9 show other examples of self-configuring policy expressed by ASSL.

Self-healing policies specify what steps the network or system should take to identify and resolve issues as they arise automatically. A self-healing policy may mandate that the system shut down, restart a broken component, or reconfigure to fix a failed configuration such as what is designed in Figure 20 for Intent 1. Figure 38 in Intent 4, Figure 43 and Figure 44 in Intent 5,and Figure 74 in Intent 9 illustrate examples of self-healing policy expressed by ASSL.

Self-optimization policies outline a system or network's steps to autonomously optimize performance in response to shifting circumstances or demand. For instance, a self-optimization strategy may state that the system should modify traffic routing to reduce delays or increase energy efficiency, such as the specification of traffic transferring due to high volume in Figure 87 for Intent 11.

Self-protection policies outline a system or network's steps to defend itself from outside threats or attacks. An example of a self-protection policy would be to mandate that the system automatically deploys security patches to guard against hostile intrusions, unauthorized access, or security threats. Security patches should be designed as software to connect with our ASSL specification as managed elements. Due to the shortcoming of the current ASSL regarding actual distribution, and connection with managed resources, this policy is not designed completely in the thesis scope, while security is also handled by ANIMA [28]. However, some examples to cover security are implemented such as Figure 65 in Intent 7, and also security is abstracted through `AEIP`, and `ASIP` interactions to limit the accessibility of messages, defining `FRIEND` lists. Two examples of `ASIP` and `AEIP` expressions are Figure 32 and Figure 34, with their `FRIEND` list in Intent 2. If we do not declare the `FRIEND` accessibility, the accessibility error as Figure 35 is shown. Another example for `FRIEND` list is Figure 60 for Intent 7.

If a policy cannot be categorized under the self-chop properties, `OTHER-POLICES`, another self-management category, is designed in ASSL to fulfill those policies. Intent 6, and Intent 7 are two concrete examples expressing `VM_CONNECTION`, and `videoconference` policies as specific self-management policies. Figure 49 and Figure 54 in Intent 6, Figure 55, Figure 56

and Figure 64 in Intent 7, Figure 66 for Intent 8, Figure 81 in Intent 10, Figure 85 and Figure 89 in Intent 11 outline the examples expressing other self-management policies.

4.2 Mapping Intent Objectives vs. ASSL Intent Examples

Intent objectives are aims or objectives that users or administrators set in autonomic networks to control how the network behaves and ensure that it fulfills its users' needs and demands. Multiple granularity levels can be used to specify intent objectives depending on the network's requirements. It is essential to consider that the current version of ASSL provides a simulation of the problem, and to ensure that the objectives are achieved, the network elements must be subjected to the proper configuration commands or actions, which are generated by a tool like a software agent and applied to them. It means that, in this thesis scope, our evaluation showed the practical aspect of the ASSL specification has a deficiency to connect to the software and devices outside of ASSL due to the shortcoming of ASSL that the current compiler does not offer a functional distributed specification to work with software agents. This should be handled in future work. Thus, the intent problems are modeled at an abstract level as simulations, and the consistency and the models are verified through formal consistency checkers inside the ASSL compiler.

As stated in Section 3.1, the solution to the intent specification language should cover the intent objectives to fulfill autonomic network goals. In the following, we list all the intent objectives that we had identified and argue that each objective is met by using ASSL as a solution to express and deploy intents, as evidenced by the concrete examples that we gave in Section 3.3.

- *Abstract Formulation* [3]: Almost all the specification examples can meet this criterion since abstract formulation refers to the expression of desired autonomic behavior at the highest level of abstraction. This is attained by separating the low-level technical configurations from the autonomic behavior. Thus, instead of focusing on what

configuration should be done in the network to achieve an intent, we focused on the goals we should plan for the network by converting the goals and their conditions into formal statements in ASSL. Examples of the abstract formulation are all the fluents as high-level representations of autonomic network states mapped to specific actions that did not include details like metrics or calls to other actions or the interface functions of managed elements that might contain details such as metrics. Omitting details such as metrics or other action calls as sub-steps of a procedure in this objective is due to focusing on results without achieving them. In all the concrete examples, policies are defined as the desired autonomic behavior for the system that has some fluents without directing the system to simple actions without numeric details, such as in some fluents from Intent 9; thus, all the related cells in Table 12 are checked with a positive sign.

- *Declarative Outcome Formulation*: This objective refers to formal declarative statements to define a desirable autonomic behavior, however, in more detail compared to abstract formulation. In the ASSL specification for our concrete examples, fluents conduct the autonomic network simulation to actions with more detailed information, such as metrics. Concrete examples such as Intent 1 and Intent 5 contain metric declarations such as Figure 22 and Figure 46 to demonstrate the capability of ASSL to express declarative outcomes. Other examples are Figure 25 for Intent 2, and Figure 86 for Intent 11, including metrics or variables that can declare the outcome of an in intent through more detailed measurements compared to abstract formulation, although they are highly abstracted compared to low-level configurations, and do not include any technical details related to the networking devices. Thus, details of measurements to achieve an intent are considered as metrics and variables to be monitored to check if the intent is respected or not, without any concern for the low-level configuration. Since all the concrete examples include at least one state that leads the autonomic network to a more detailed procedure, like metrics, calls to interface functions, and parameters, all the related cells in Table 12 are checked with a positive sign.

- *Portability*: First, ASSL being implemented using Java, all the ASSL specifications

can be run on all operating systems. We successfully tested them on Linux and Windows. The only requirement for the system to run the ASSL specifications is its ability to run Java, by having the Java Virtual Machine (JVM) installed. Therefore, intents expressed in ASSL are portable from a simple execution perspective. However, portability, in the case of the intent, is an ideal concept to attain because an intent expression method should be able to run all the intents without any modifications to control networking elements that operate technologies that are developed by different vendors. Vendors have diverse command modeling, making the portability objective challenging. In order to deal with this problem, ASSL has some features to abstract the portability by minimizing the adjustment of the intent: the managed elements controlled by ASSL are abstract Java objects that can simulate the actual software interfaces of the controlled devices. Thus, if a set of generic commands is designed to be compatible with all the vendors, to be considered under the control of the managed elements as proxies, in case of a change between vendors, only the managed elements should adjust to the new vendor. This procedure will remove the requirement to modify the intent specification itself. Since this concept applies to all the concrete examples, all the related cells in Table 12 are checked with a negative sign as there is a potential for ASSL to achieve it, but it is not achieved yet.

- *Distributed and Local Behavior Management*: Since ASSL has a hierarchical architecture, and each of the actions has its specific block separate from other actions, this objective can be demonstrated through almost all concrete examples. Public autonomic behaviors like alarm generation statements were defined under the AS tier in a distributed autonomic manner accessible by the whole autonomic network. In contrast, the actions specified under each `AE` block are the means to determine the localized autonomic behaviors of that `AE`. Because all the concrete examples have the `AS`, `AE`, and `ASIP` hierarchy at least, each of these levels has its functionalities where `AS` and `ASIP` specifications demonstrate the distributed management. As of right now, ASSL is simulating a distributed system, but not practically. To execute it

as a distributed system, however, only code generation needs to be implemented. The `AE` sections refer to localization, and all the related cells in Table 12 are checked with a positive sign.

- *Composability*: Composability needs to be demonstrated from two different angles. First, the combination of different intents. Intents are composed of policies to define autonomic behaviors, so the possibility of adding policies to a current specified policy with minimal disruption in the whole specification results in composability. In some concrete examples like Intent 7, first, we created an intent with its own policy and then added other policies to that previous specification, and those separate policies could work together under the same autonomic system with no interference. The capability of policies working beside each other also brings the potential *Modularity* of ASSL formal specifications. Secondly, in some examples, such as Intent 9, intent functionality was created for the autonomic behavior, which was extended by adding more actions to complete the autonomic behavior in some other aspects. Thus, ASSL specification provides a simulation of autonomic behaviors, which are composable by adding two intents together as different policies or adding functionality to the behaviors defined in the ASSL to increase or modify the tasks based on the intent requirement. To implement Intent 7, first, a policy is specified for video conferencing for a Controller AE and its user AEs. Then, a state is added to cover requesting action in this intent. Then to achieve Intent 8, another group of functionalities is added to the previous intent structure, which results in the combination of intents showing the composability feature of ASSL. With the same logic, Intent 2, Intent 3, and Intent 4 can be in one group. To prove the claim with the execution of specifications, we combined some intents together and ran the specifications before and after this combination. The results of further assessment of composability through execution of different intents together are stated in Section 4.2.1. Another way to demonstrate composability was to create AEs with some autonomic behavior and then add other policies, such as the process that happens for all other intents, where different policies can work together

but separately. This specification type allows adding more policies to the autonomic system when required. Therefore, since all the intent specifications have one of the two reasons to demonstrate the composability objective, their cells are checked as positive in Table 12. The only exception is Intent 6, which has no signs in the cell. This is because the essence of this intent focused on the connectivity of the AEs. It was not required to specify this intent in a composable manner. Although its cell is empty, there is no reason not to allow adding other policies to this intent. To conclude this analysis, in the future version of the ASSL, where all the specifications are distributed pragmatically, there might be conflict in running more than one separate ASSL intent specifications aiming to control the same network system since each of the intents' actions can potentially affect another intent's specified goal. However, it might be possible to reduce this potential risk by creating a self-healing policy to recognize and resolve the conflict. Notably, we would need a mechanism to make each of the intents either: (1) automatically integrated or (2) have a centralized orchestrator to execute many different individually defined intents in one system.

- *Efficiency* [42]: Since ASSL provides a high-level specification of intent and the autonomic network concepts together in one formal language without requiring other modeling languages such as YANG, it is efficient from an expressiveness perspective. Also, it encapsulates many complexities from the intent users through its level of abstraction. Additionally, it is also efficient from the programming perspective in terms of expressiveness and developer time. It means that to express an autonomic behavior with ASSL there is a lower amount of coding required compared to expressing that with other programming languages such as Java. For instance, in all of the concrete examples, the number of lines of generated Java code as the output of ASSL is far greater than the number of lines of the ASSL specification code. This evaluation applies to all the specification examples, and thus, all the related cells in Table 12 are checked with a positive sign.

- *Scalability* [22]: The ASSL specification of almost all concrete examples shows the

potential *structural* scalability of ASSL. This is demonstrated when in all the scenarios, we could define an autonomic element as the Controller to define central autonomic behaviors and then add other autonomic elements as its users with minimum disruption to the tasks done by the system. For instance, in Intent 1 and Intent 9, we implemented the intent with only the Controller as the autonomic element and then involved the autonomic elements named as Host by adding their specifications. The autonomic network intent specification could work in both states, and adding the new specifications added newly specified behaviors. This addition of autonomic elements outlined that without changing the primary resource, we can add a simulation of a new autonomic element. However, we should mention that this feature was demonstrated only at a formal specification of simulated autonomic behaviors. Thus, we mentioned it as potential scalability, although it can be proven practically when the ASSL compiler is improved to a real distributed system to work with software agents. This also shows the minimum intrusion level of ASSL components for each other when adding a new autonomic element to the structure. We only had to add some minimal tasks like calls to the new element to make it synchronized with the previous system specification. However, the static essence of ASSL architecture led us to regenerate the code after adding new elements. As an example, the results of packet generation for the abstract scalability of Intent 1 are shown in Figure 12, Figure 13, Figure 14, Figure 15, and Figure 16. Also, the lack of dynamism in `ASARCHITECTURE` as the grouping ASSL structure caused us to consider only the simulation of the reconfiguration state in Intent 3 by defining the corresponding fluents and actions. The current version of ASSL cannot dynamically and practically add the new device to the topology. Nevertheless, considering the simulation of the intent, the lack of dynamism in grouping nodes and the practical addition of the new device can be resolved in the future version of ASSL because there is no impediment to the improvement of the ASSL compiler. In conclusion, ASSL's static design is a dynamism issue since it depicts the system architecture in a static manner that does not accurately reflect the dynamic

character of the network and its surroundings in case of adding or removing autonomic elements. Designing and putting into practice an autonomic network that may change with the environment and react to unforeseen events may be challenging as a result since after each addition or removal of autonomic elements, this situation will affect the relationships defined between the autonomic nodes and the system should regenerate the specification code to adapt to the change. The regeneration after these changes may also affect the speed of management to react to these situations. All the concrete examples except Intent 10 are implemented with one Controller and Hosts or other AEs that depend on them. Therefore, they can demonstrate the potential structural scalability of ASSL. This realization is due to the minimum disruption of the primary autonomic behavior designed in the central autonomic element, which results in minimized impacts on the intent specification. For Intent 10, the cell is empty in Table 12 since it only expresses the goal through the Controller; however, there is no obstacle to adding other AEs to this specification. To conclude, ASSL can express scalability in terms of specifying more resources, such as AEs, or managed elements at an abstract level. Real scalability refers to either (1) deploying an intent on networks of varying scales without having to change the formulation of the intent, or (2) deploying a large number of intents on a network. As we focused on intent expression rather than deployment, the actual deployment on a networking testbed is out of scope for this thesis, and we were thus not able to test for neither of these aspects of scalability.

- *Monitoring Capabilities*: In ASSL, there is a capability of internal observation of conditions so that the autonomic system can track if the autonomic behavior is executed based on the output provided by the ASSL control loop. In the case of metrics, also we could observe that if the metric was violated, the `GUARD` as the preconditions of action was not met, and therefore the action was not executed. However, this monitoring is only a simulation since ASSL cannot directly read the metric from actual networking devices. After an enhancement in the ASSL compiler, when it can practically connect

to networking devices as its managed element through a software agent, besides the internal monitoring capabilities of ASSL, monitoring software can work with it to improve this aspect of ASSL. Therefore, these criteria are met partially in the current version of ASSL since it depends on the ASSL connection to actual networking devices. This realization applies to all the concrete examples, and all the related cells in Table 12 are checked with a negative sign. This refers to the potential of ASSL to support monitoring tools if its compiler is improved to work with actual software agents, on the one hand, and the internal observation capability it has to check its control loop. To conclude, ASSL can monitor any intent condition or metrics specified. Although, in terms of working with a monitoring tool outside of ASSL as a managed element, there should be a software agent to connect them.

- *Security* [3]: This objective primarily concerns low-level configuration, which is investigated in ANIMA's work as they build a secure infrastructure for autonomic networks in [28]. This infrastructure might be combined with ASSL specifications in its enhanced version. However, in the current ASSL specification of concrete examples, there was an abstract simulation secure interaction concept where autonomic behaviors defined in one autonomic element were not accessible by others unless they were described as `FRIENDS`. Also, private channels between AEs were the abstract means to transmit messages between AEs but not the whole autonomic system, similar to public messages expressed in `ASIP` tiers. As a result, this capability was partially met in the ASSL specifications that we evaluated. This realization applies to all the concrete examples, and as a result, all the related cells in Table 12 are checked with a negative sign. This means that ASSL applies abstract security based on some features that it has to limit accessibility. To conclude, although security is a much lower-level concept than what ASSL focuses on, there are security aspects that ASSL can provide regarding accessibility. An example of consistency error because of lack of `FRIEND` declaration is Figure 61 for Intent 7.

- *Autonomic Reporting* [3]: This objective depends on the ability of ASSL to connect

Table 12: Intent objectives and concrete examples comparison

Intent Objective \ Intent Number	Intent 1	Intent 2	Intent 3	Intent 4	Intent 5	Intent 6	Intent 7	Intent 8	Intent 9	Intent 10	Intent 11
Abstract Formulation	+	+	+	+	+	+	+	+	+	+	+
Declarative Outcome Formulation	+	+	+	+	+	+	+	+	+	+	+
Portability	–	–	–	–	–	–	–	–	–	–	–
Distributed and Local Behavior Management	+	+	+	+	+	+	+	+	+	+	+
Composability	+	+	+	+	+		+	+	+	+	+
Efficiency	+	+	+	+	+	+	+	+	+	+	+
Scalability	+	+	+	+	+	+	+	+	+		+
Monitoring Capabilities	–	–	–	–	–	–	–	–	–	–	–
Security	–	–	–	–	–	–	–	–	–	–	–
Autonomic Reporting	–	–	–	–	–	–	–	–	–	–	–

(+) Support, (-) Partially Support, or Potentially.

to the actual networking devices to monitor real metrics, assess autonomic behaviors based on them, and report them to the users. Although the output of ASSL is the generated Java code that can show if the autonomic behaviors are executed correctly based on the specification, we could not achieve this objective. For the current ASSL compiler, there is a possibility to check if the autonomic statements work appropriately based on the design, but there is not a dynamic report of the real autonomic network devices. This explanation applies to all the concrete examples; all the related cells in Table 12 are checked with a negative sign. It means that ASSL has the potential to apply this objective if its compiler is enhanced.

4.2.1 Further Evaluation of Composability

We demonstrated our further evaluations for composability through the combination of Intent 2 and Intent 3. First, we compiled the specifications for each of these intents, where the results of generated `AS` packages can be seen in Figure 91 and Figure 92. Then we added these two intents without modifying any part of the individual specifications and compiled the code. The result of package generation can be seen Figure 93. Before and after

Figure 91: `AS` generated package in Intent 2.

the combinations, the codes were consistent, and therefore the packages could be generated. In Figure 93 the `SELF_CONFIGURING` is extracted from Intent 3 with its related actions which are marked with green shapes, and `SELF_HEALING` is extracted from Intent 2 with its related actions marked with red shapes.

Moreover, the following figures demonstrate some parts of the execution of the generated Java code where there is a trace of formal statements that we defined to form the intent-based autonomic behaviors. Figure 94 and Figure 95 show some lines from Intent 2 and Intent 3 individually, and Figure 96 illustrates the execution of the specification where the two previous intent are combined to form a new intent. In this trace, lines 53 to 65 and 68 are traces related to detecting and notifying a new host and reconfiguring the system because of this event. Also, lines 66 and 68 are sample traces caused by the self-healing policy from Figure 95.

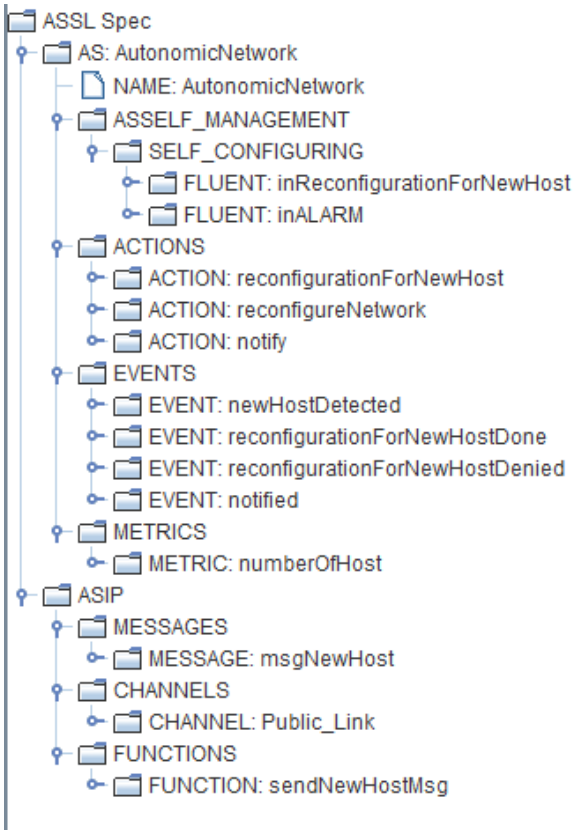

Figure 92: `AS` generated package in Intent 3.

4.3 Coverage of Different Categories of Intent

As stated in Section 3.3, the concrete examples were chosen from different intent classifications to evaluate the expressiveness and coverage of ASSL for intent specification as much as possible. Different classifications are mentioned in [23] to categorize the intents as groups based on various criteria. The first categorization is based on the solutions and intent of users. The solution network is divided into *Carrier Networks*, *Data Center Networks*, and *Enterprise Networks*. Carrier networks refer to telecommunication companies to provide services to customers. Data Center networks are responsible for a larger scale of networking requiring a higher-performance and more complex computing like cloud computing. Enterprise networks cover networking for specific organizations. Intent users

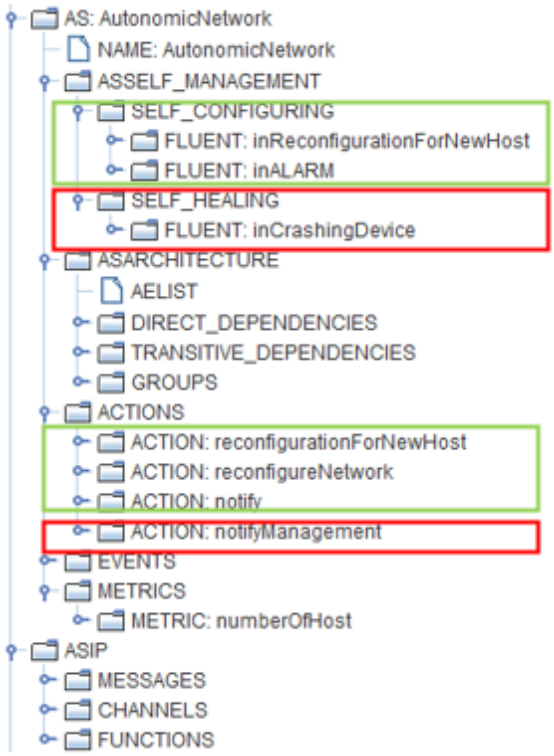

Figure 93: `AS` generated package after the combination of Intent 2 and Intent 3.

```
17 ***************** AS STARTED SUCCESSFULLY ****************
18 ************************************************************
19 EVENT 'generatedbyassl.as.autonomicnetwork.events.NEWHOSTDETECTED': has occurred
20 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INRECONFIGURATIONFORNEWHOST': has been initiated
21 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INALARM': has been initiated
22 ACTION 'generatedbyassl.as.autonomicnetwork.actions.RECONFIGURATIONFORNEWHOST': has
   been performed
23 EVENT 'generatedbyassl.as.autonomicnetwork.events.RECONFIGURATIONFORNEWHOSTDONE':
   has occurred
24 ACTION 'generatedbyassl.as.autonomicnetwork.actions.RECONFIGURENETWORK': has been
   performed
25 ACTION 'generatedbyassl.as.autonomicnetwork.actions.NOTIFY': has been performed
26 EVENT 'generatedbyassl.as.autonomicnetwork.events.NOTIFIED': has occurred
27 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INRECONFIGURATIONFORNEWHOST': has been terminated
28 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INALARM': has been terminated
```

Figure 94: Excerpt of ASSL controller output execution Intent 2.

```
43 ***************** AS STARTED SUCCESSFULLY ****************
44 ************************************************************
45 EVENT 'generatedbyassl.as.aes.host.events.TIMETOSENDGENERALNOTIF': has occurred
46 FLUENT 'generatedbyassl.as.aes.host.aeself_management.self_healing.INGENERALNOTIF
   ': has been initiated
47 ACTION 'generatedbyassl.as.aes.host.actions.NOTIFYFORGENERALNOTIF': has been
   performed
48 EVENT 'generatedbyassl.as.aes.host.events.ISMSGGENERALNOTIFSENT': has occurred
49 FLUENT 'generatedbyassl.as.aes.host.aeself_management.self_healing.INGENERALNOTIF
   ': has been terminated
50 EVENT 'generatedbyassl.as.aes.controller.events.TIMETORECEIVEDGENERALNOTIFMSG': has
    occurred
51 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_healing.
   INGENERALNOTIF': has been initiated
52 ACTION 'generatedbyassl.as.aes.controller.actions.CONFIRMGENERALNOTIF': has been
   performed
53 EVENT 'generatedbyassl.as.aes.controller.events.MSGGENERALNOTIFRECEIVED': has
   occurred
54 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_healing.
   INGENERALNOTIF': has been terminated
55 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_healing.
   INCHECKINGHOSTTOOL': has been initiated
56 ACTION 'generatedbyassl.as.aes.controller.actions.CHECKHOSTTOOLSTATUS': has been
   performed
57 EVENT 'generatedbyassl.as.aes.controller.events.TOOLOK': has occurred
58 FLUENT 'generatedbyassl.as.aes.controller.aeself_management.self_healing.
   INCHECKINGHOSTTOOL': has been terminated
```

Figure 95: Excerpt of ASSL controller output execution Intent 3.

```
51 ***************** AS STARTED SUCCESSFULLY ****************
52 ************************************************************
53 EVENT 'generatedbyassl.as.autonomicnetwork.events.NEWHOSTDETECTED': has occurred
54 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INRECONFIGURATIONFORNEWHOST': has been initiated
55 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INALARM': has been initiated
56 ACTION 'generatedbyassl.as.autonomicnetwork.actions.RECONFIGURATIONFORNEWHOST': has
    been performed
57 EVENT 'generatedbyassl.as.autonomicnetwork.events.RECONFIGURATIONFORNEWHOSTDONE':
   has occurred
58 ACTION 'generatedbyassl.as.autonomicnetwork.actions.RECONFIGURENETWORK': has been
   performed
59 ACTION 'generatedbyassl.as.autonomicnetwork.actions.NOTIFY': has been performed
60 EVENT 'generatedbyassl.as.autonomicnetwork.events.NOTIFIED': has occurred
61 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INALARM': has been terminated
62 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INRECONFIGURATIONFORNEWHOST': has been terminated
63 EVENT 'generatedbyassl.as.autonomicnetwork.events.NEWHOSTDETECTED': has occurred
64 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INRECONFIGURATIONFORNEWHOST': has been initiated
65 FLUENT 'generatedbyassl.as.autonomicnetwork.asself_management.self_configuring.
   INALARM': has been initiated
66 EVENT 'generatedbyassl.as.aes.host.events.TIMETOSENDGENERALNOTIF': has occurred
67 FLUENT 'generatedbyassl.as.aes.host.aeself_management.self_healing.INGENERALNOTIF
   ': has been initiated
68 ACTION 'generatedbyassl.as.autonomicnetwork.actions.RECONFIGURATIONFORNEWHOST': has
    been performed
```

Figure 96: Excerpt of ASSL controller output execution Intent 2 and Intent 3.

Table 13: Mapping of RFC9316 intent classification vs. ASSL intent examples.

Intent Solution	Intent Users	Intent Number
Carrier Networks	Network Operators, Service Designers, App Developers, Service Operators, Customers / Subscribers	Intent 1 Intent 3 Intent 4 Intent 5 Intent 10 Intent 11
DC Networks	Cloud Administrators, Underlay Network Administrators, Application Developers, Customers / Tenants	Intent 2 Intent 4 Intent 6 Intent 9 Intent 11
Enterprise Networks	Enterprise Administrators, Application Developers, End Users	Intent 3 Intent 4 Intent 7 Intent 8 Intent 11

in all three groups vary from professional users like administrators and operators to non-professional users like customers and end users. The concrete examples are designed to cover all these categories as depicted in Table 13. In this table, Intent 11 is considered as an example for all the categories. This is due to the essence of this intent that aims to manage the network, as management is the most fundamental intent that applies to all networks. However, this intent is considered to be used by technical intent users who are aware of networking concepts.

We assessed ASSL capabilities as an intent expression language by evaluating the expressiveness of ASSL in concrete examples chosen from various intent classifications. We also tried to select the intents from different categories to investigate the shortcoming of ASSL. Based on our analysis of specified intents, ASSL can specify almost every intent, at least at its highest abstract level. However, the most challenging part is related to the steps where there is a requirement to have interaction with actual networking devices and other tools as managed resources. In other words, we could specify policies and autonomic behaviors as a simulation but not an actual distributed specification code that can run the

system pragmatically. This also affects the specification of two intent categories. Firstly, in intents related to application developers as intent users and second in intents categorized as operational tasks such as adding a new device to the system. In both cases, there is a requirement to interact with actual networking devices, applications, or software. Despite the simulation-only nature of current ASSL specifications of intent, due to the capability of ASSL to check the consistency of specifications and show the tracking of the autonomic behavior when we ran generated Java code as the output of the specification, we could verify that the autonomic behavior is respected based on the chain of events and states that we specified to create self-management policies as intents. The vital classification for selecting the scenarios to specify with ASSL relates to the context of intents. In [23], a classification divides various networking concepts into intent types used by diverse users. The classification contains practical concepts, such as configuration, or network migration, as well as abstract concepts, such as creating design models. All the intent types are extracted from different scenarios that might happen in different scales of network solutions. We selected scenarios from all different groups to cover almost every category and adequately analyze the ASSL specifications' pros and cons. All the scenarios can be studied in two phases:

- *Modeling Level*: This is the highest abstract level where we modeled and expressed states, contexts, timings, and actions as formal statements.
- *Practical Level*: The applicability of the expressions on the networking devices by focusing on filling the gap between the specifications and real network scenarios.

For the first phase, we could express all the context at an abstract level by simulating the concepts as fluent, actions and events with their specified conditions. Therefore, theoretically, there is no lack of expressiveness power of the ASSL. Also, in two stages, we analyzed the consistency of the expressions through the consistency checker property of ASSL. An example of this test, and the code generation after this step are shown in Figure 97, and Figure 98 for Intent 10. Then, we assessed if the autonomic system can respect the autonomic behavior by running the generated Java code and tracking the traces of it. The results are concluded as Table 14.

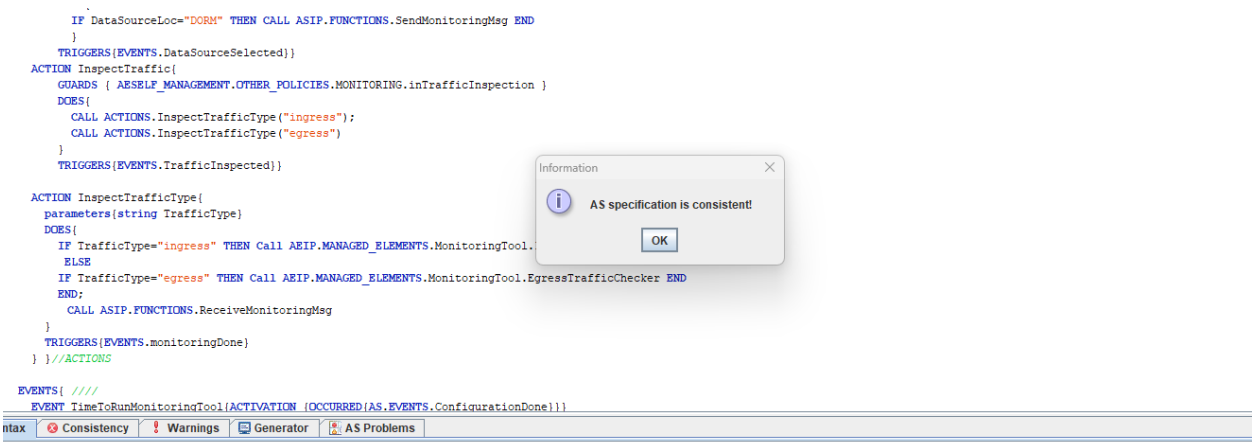

Figure 97: Consistency check example for Intent 10.

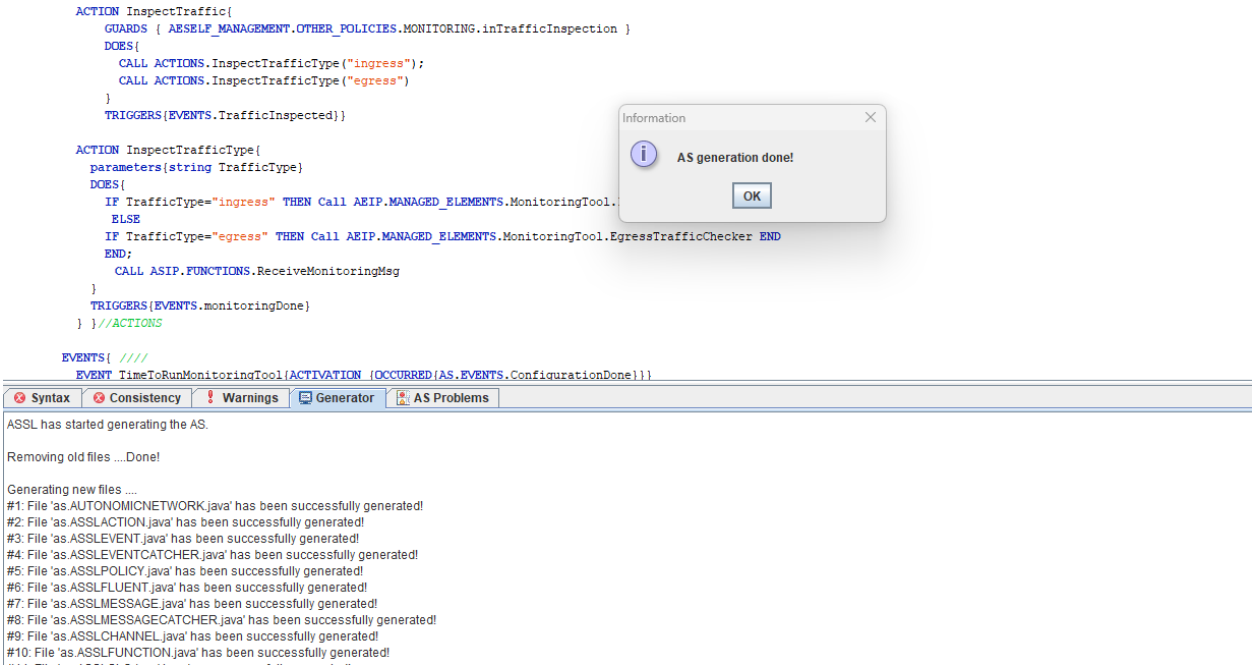

Figure 98: Code generation example for Intent 10.

Table 14: Mapping of RFC9316 intent contexts classification vs. ASSL intent examples.

Intent Type	Intent Contexts	Intent Number
Customer Service Intent	Self-service or SLA based services, Service operator orders	Intent 1 Intent 5
Network and Underlay Network Service Intent	Service operator orders, Intent-driven network Configuration, Verification, Correction, Optimization, Intent created by the underlay network administrator	Intent 5 Intent 2 Intent 3 Intent 4 Intent 11
Network and Underlay Network Intent	Network configuration, Automated life-cycle management of network configuration, Network resources (switches, routers, routing, etc.)	Intent 2 Intent 3 Intent 4 Intent 9 Intent 11
Cloud Management Intent	DC configuration, VMs, DB servers, Application servers, Communication between VMs	Intent 2 Intent 6 Intent 7 Intent 8
Cloud Resource Management Intent	Cloud resource life-cycle management (Policy-driven self-configuration, Auto-scaling, Recovery/optimization)	Intent 9
Strategy Intent	Security, QoS, Application policies, Traffic steering, Configuring and monitoring policies, Alarm generation for non-compliance, Auto-recovery, Design models and policies for network, Network service design Design workflows, Design Models, Policies	Intent 1 Intent 2 Intent 3 Intent 5 Intent 10 Intent 11
Operational Task Intents	Network migration, Device replacements Network software upgrades, Task Automation	Intent 3 Intent 11

4.4 Summary

In Chapter 4, we brought some examples of intent specifications to show the capabilities of ASSL to express self-management policies in Section 4.1. Then, we analyzed if ASSL could meet the intent objectives from Section 3.1 and brought different specification examples for analysis in Section 4.2. Moreover, we categorized the results of the evaluations in Table 12, and Table 13. Additionally, we scrutinized mapping the concrete examples to RFC 9316 [23] intent classifications and presented the results in Table 14.

Chapter 5

Conclusion and Future Work

In this final chapter, we present an overview of all the arguments made in this thesis, followed by representing the limits we encountered in our suggested solution during this research and how future work can be done to solve each of them.

5.1 Conclusion

The growing complexity and dynamic nature of contemporary computer networks created a demand for intent-based networking, where networks can be managed based on *Intent*, which is the desirable network goal. On the other hand, *Autonomic Networking* aims to facilitate network management by applying self-management policies while minimizing the need for human involvement in management procedures. Instead of manually configuring the network's low-level components, autonomic networks can govern the networks based on the *Intent*. By automating network management and facilitating operators' expression and enforcement of their desired network behavior, intent-based networking under autonomic networking seeks to address these issues. To achieve this purpose, there are two problems:

- *Lack of a Comprehensice Intent Definition*: To solve this issue, *Intent Objectives* are introduced in Section 3.1.

- *Lack of a Language or Framework to Express and deploy Intents in Autonomic Networks*: To solve this problem, *Autonomic System Specification Language (ASSL)* is suggested as the framework designed for autonomic systems to specify *Intent* in Chapter 3.

There are various advantages to intent specification in autonomic networks using Autonomic System Specification Language (ASSL). Firstly, ASSL is a framework designed specifically for autonomic systems, as it allows the specification of self-management policies to create autonomic behaviors through features like control loops. Due to its features as mentioned in Section 3.2, and illustrated by concrete intent examples selected from different categories of intent classification from RFC 9316 [23] in Section 3.3, ASSL can provide the means to achieve an appropriate range of the *Intent Objectives* described in Section 3.1. Mapping of *Intent Objectives* to ASSL intent specification examples is evaluated in Section 4.2, and depicted in Table 12. Also, for some *Intent Objectives* such as *Portability* and *Security*, ASSL has the potential to handle them, or provide them to some extent, although the current version has some shortcomings.

Also, ASSL can have an appropriate coverage of diverse intent classifications from RFC 9316 [23], in terms of *Intent Users* and *Intent Solutions*, as illustrated in Table 13 and Table 14 in Section 4.3. It is essential to mention that the ASSL coverage of intent classification in terms of modeling different intents in a high-level abstraction is appropriate; however, in the case of operational tasks, or interaction with actual networking devices through software agents, it requires some improvements in its compiler.

In conclusion, utilizing ASSL for intent specification in autonomic networks as a top-down approach has several advantages, including covering a reasonable number of *Intent Objectives* and *Intent Classifications*. At least in almost every case, it can handle the modeling and high-level specifications. However, to make it practical for autonomic networks, there are gaps in a software agent to connect ASSL interface functions to networking devices on the one hand and the ability to distribute ASSL specification components from each other.

- *Abstract Formulation* can be attained by ASSL specification since all the intent concrete

examples can represent abstract network goals without concerns for the steps to achieve that specific goal. Self-management policies, fluents, and actions help the autonomic network to formulate such abstraction for autonomic network behaviors.

- *Declarative Outcome Formulation* can be achieved since network goals as intents and constraints as metrics as foundations of autonomic network behaviors can be represented by ASSL features in more detail. Declarative outcome formulation contains more detail compared to the abstract formulation. ASSL can assess if the defined constraints are met and if the autonomic behavior respects the intent criteria. To formulate declarative outcomes, self-managed policies, fluents, actions, and metrics are some useful ASSL components.

- *Portability* is a potential for ASSL specification at an intent specification perspective due to the abstraction level of ASSL specifications, which does not consider any low-level configurations. As a result, different vendors can use this specification language to define high-level network goals. However, for the interaction of managed elements interfaces from ASSL with actual networking devices, some vendor-based commands might be required based on the device settings or configurations. On the other hand, at an execution level, ASSL can be executed on all devices capable of running Java. This operational view is out of the scope of this thesis that focuses on the expressiveness of ASSL for intent-based autonomic networking. Nevertheless, portability will be enhanced if a template of commands is considered for these interactions to be interpreted for all different managed element interactions.

- *Distributed and Local Behavior Management* is achieved by ASSL since we expressed two levels of private and public autonomic behaviors in all the concrete intent examples. The private autonomic behaviors refer to AE blocks localized for autonomic elements. The global perspective is specified under the AS and ASIP tiers to cover the autonomic rules shared among the whole autonomic network.

- *Composibilty* was discussed in two different perspectives as mentioned in Section 4.2.

It is possible to consider ASSL as a specification language with the potential of composability since it can define different self-management policies in one autonomic network without conflicts. However, injecting diverse ASSL intent specifications to rule one autonomic network at an operational level in a real networking test bed was not in the scope of this thesis.

- *Efficiency* is provided by ASSL as this specification language does not require modeling languages such as YANG as a part of it, and it has the tools to express network intents under the autonomic network scheme and verify the autonomic behavioral models through its consistency checking. Also, its generated code is efficient in several code lines compared to the coding required to specify the autonomic network behaviors.

- *Scalability* is achievable at an abstract level as we could define different concrete intent examples that have the potential to add more AEs to the specification with minimal modification in the intent specification. Although this potential is proven theoretically, the operational estimations to assess scalability while there is a large number of networking devices or intents are outside the scope of this thesis.

- *Monitoring Capabilities* are attained by the ASSL concrete examples as all of the defined autonomic behaviors were verified through inner control loops of the ASSL to verify the consistency of the specifications based on the semantics of the ASSL, which apply to autonomic systems.

- *Security* is a low-level concern. Nevertheless, ASSL can define some abstract features, such as friends, to control the accessibility of the interaction between autonomic elements in an autonomic system.

- *Autonomic Reporting* refers to the ability of the specification to connect to the actual networking devices. Since we chose another direction in this thesis, we do not focus on concrete autonomic reports. However, through the generated Java code after execution, the autonomic behaviors can be traced in all of the concrete examples.

5.2 Limitations

Based on our evaluation of different concrete intent examples specified with ASSL, there are some challenges to using this framework for intent expression under the autonomic networking context.

- *Static Architecture*: ASSL lacks the dynamism to add or remove an autonomic element to a specified structure, which results in the regeneration of code after a change in the architecture. Examples are Intent 2, and Intent 4. This also can impact the evaluation of the scalability objective. We encountered this challenge as we added or removed an AE for the specification we tried to design during the implementation of Intent 1 and Intent 9.

- *Lack of Distribution in Practice*: ASSL specification provides autonomic behavior, which is distributed abstractly, but since the final generation of code is compiled as one unified package, each intent specification model cannot be injected into an autonomic node separately. This means that the whole specification, including all the hierarchy components, should be executed in all the devices. This shortcoming can affect the evaluation of scalability and composability because although we could express these objectives through all the concrete examples, we encountered the challenge of testing these objectives in a real networking test bed due to lack of resources like timing and a pragmatic distributed ASSL compiler to interact with the test bed devices.

- *Lack Practical Interaction with Managed Resources*: Managed elements in ASSL model an interface function to connect the ASSL specification to the actual networking devices or software. However, ASSL only simulates this modeling, requiring a software agent to create the communication for real interaction. The shortcoming of ASSL in the distribution of specification blocks impedes the system from using a software agent connection for its interface functions because of the dependency between ASSL compiling structures. Therefore, ASSL does not directly read metrics from other resources, and we had to define the metric values manually. Nevertheless, if the

distribution problem is resolved through compiler enhancement, this problem can be resolved.

5.3 Future Work

As pointed out in Section 5.2, ASSL has shortcomings that cause a challenge to test intent specification at a practical level in a real distributed networking test bed. Nevertheless, the ASSL compiler does not impede alleviating these problems to ameliorate its capabilities. The possible future solution to ASSL shortcomings that might improve the functionalities of ASSL are as follows:

- *Distributed Execution of ASSL Specification of Components*: The ASSL specification for autonomic elements should become modular in terms of being able to be compiled individually from other specification blocks. To explain more, in the current version of ASSL, all the specifications for the autonomic elements are compiled together under one file and generate one integrated packaging as its output. Thus, to make each AE specification applicable to individual devices, the ASSL compiler should add this modularity to its compiling procedure. This improvement also allows software agents to work with ASSL specifications on each device to connect the device by interpreting the specifications.

- *Dynamic Autonomic System Architecture*: ASSL has a static architecture. It is not possible to represent dynamic addition or removal of Autonomic Elements at run time. Adding this dynamic capability would require changes to the semantics of ASSL, and to the supporting run-time system. This improvement could also positively affect attaining other intent objectives that rely on dynamism, such as scalability and composability.

As the ASSL will be extended toward better capacities for autonomic networking, it can also be implemented to integrate other existing solutions for autonomic networking, such as

ANIMA and the GRASP protocol. In doing so, not only could ASSL express and execute the network intent through its formal statements independent of the underlying network infrastructure and technology but this high-level specification capacity can be combined with ANIMA and GRASP if it is translated into the configuration for network devices by a software agent or an Autonomic Service Agent (ASA) when ASSL shortcomings are resolved. As a result, ASSL can apply intent specifications to the autonomic network domain, which is also secured through ANIMA low-level mechanisms.

References

[1] Harald T. Alvestrand. A Mission Statement for the IETF. RFC 3935, October 2004.

[2] Michael H. Behringer, Brian E. Carpenter, Toerless Eckert, Laurent Ciavaglia, and Jéferson Campos Nobre. A Reference Model for Autonomic Networking. RFC 8993, May 2021.

[3] Michael H. Behringer, Max Pritikin, Steinthor Bjarnason, Alexander Clemm, Brian E. Carpenter, Sheng Jiang, and Laurent Ciavaglia. Autonomic Networking: Definitions and Design Goals. RFC 7575, June 2015.

[4] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. Association for Computing Machinery.

[5] Mehdi Bezahaf, Eleanor Davies, Charalampos Rotsos, and Nicholas Race. To All Intents and Purposes: Towards Flexible Intent Expression. In *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*, pages 31–37, 2021.

[6] Carsten Bormann, Brian E. Carpenter, and Bing Liu. GeneRic Autonomic Signaling Protocol (GRASP). RFC 8990, May 2021.

[7] Carsten Bormann and Paul E. Hoffman. Concise Binary Object Representation (CBOR). RFC 8949, December 2020.

[8] Javier Cámara, Kirstie L Bellman, Jeffrey O Kephart, Marco Autili, Nelly Bencomo, Ada Diaconescu, Holger Giese, Sebastian Götz, Paola Inverardi, Samuel Kounev, et al. Self-aware computing systems: Related concepts and research areas. In *Self-Aware Computing Systems*, pages 17–49. Springer, 2017.

[9] Brian E. Carpenter, Bing Liu, Wendong Wang, and Xiangyang Gong. GeneRic Autonomic Signaling Protocol Application Program Interface (GRASP API). RFC 8991, May 2021.

[10] Alexander Clemm, Laurent Ciavaglia, Lisandro Zambenedetti Granville, and Jeff Tantsura. Intent-Based Networking - Concepts and Definitions. RFC 9315, October 2022.

[11] Toerless Eckert, Michael H. Behringer, and Steinthor Bjarnason. An Autonomic Control Plane (ACP). RFC 8994, May 2021.

[12] Saad Haji, Subhi Zeebaree, Rezgar Saeed, Siddeeq Ameen, Hanan Shukur, Naaman Omar, Mohammed M.Sadeeq, Zainab Ageed, Ibrahim Mahmood, and Hajar Yasin. Comparison of software defined networking with traditional networking. *Asian Journal of Computer Science and Information Technology*, 9:1–18, 05 2021.

[13] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. Intent-Driven Composition of Resource-Management SDN Applications. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 86–97, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Mike Hinchey and Emil Vassev. The art of developing autonomic systems - the assl approach. In *2010 4th IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 9–12, 2010.

[15] IBM Corporation. An architectural blueprint for autonomic computing. Technical report, IBM Corporation, 2006.

[16] The Internet Research Task Force (IRTF). Accessed: 2023-02-08.

[17] Arthur S. Jacobs, Ricardo J. Pfitscher, Rafael H. Ribeiro, Ronaldo A. Ferreira, Lisandro Z. Granville, Walter Willinger, and Sanjay G. Rao. Hey, Lumi! Using Natural Language for Intent-Based Network Management. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 625–639. USENIX Association, July 2021.

[18] Arthur Selle Jacobs, Ricardo José Pfitscher, Ronaldo Alves Ferreira, and Lisandro Zambenedetti Granville. Refining Network Intents for Self-Driving Networks. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*, SelfDN 2018, pages 15–21, New York, NY, USA, 2018. Association for Computing Machinery.

[19] Sheng Jiang, Brian E. Carpenter, and Michael H. Behringer. General Gap Analysis for Autonomic Networking. RFC 7576, June 2015.

[20] Sheng Jiang, Zongpeng Du, Brian E. Carpenter, and Qiong Sun. Autonomic IPv6 Edge Prefix Management in Large-Scale Networks. RFC 8992, May 2021.

[21] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, jan 2003.

[22] Benjamin Lewis, Lyndon Fawcett, Matthew Broadbent, and Nicholas Race. Using p4 to enable scalable intents in software defined networks. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 442–443, 2018.

[23] Chen Li, Olga Havel, Adriana Olariu, Pedro Martinez-Julia, Jéferson Campos Nobre, and Diego Lopez. Intent Classification. RFC 9316, October 2022.

[24] Open Network Operating System (ONOS). Accessed: 2022-04-08.

[25] Olga Ormandjieva, Heng Kuang, and Emil Vassev. Reliability self-assessment in reactive autonomic systems: Autonomic system-time reactive model approach. *ITSSA*, 2(1):99–104, 2006.

[26] Olga Ormandjieva and Emil Vassev. ASSL specification of a self-scheduling mechanism in team-robotics modeled with the AS-TRM. *System and Information Sciences Notes*, 2(1):132–137, September 2007.

[27] Ying Ouyang, Chungang Yang, Yanbo Song, Xinru Mi, and Mohsen Guizani. A Brief Survey and Implementation on Refinement for Intent-Driven Networking. *IEEE Network*, 35(6):75–83, 2021.

[28] Max Pritikin, Michael Richardson, Toerless Eckert, Michael H. Behringer, and Kent Watsen. Bootstrapping Remote Secure Key Infrastructure (BRSKI). RFC 8995, May 2021.

[29] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. Merlin: A Language for Managing Network Resources. *IEEE/ACM Transactions on Networking*, 26(5):2188–2201, 2018.

[30] Yoshiharu Tsuzaki and Yasuo Okabe. Reactive configuration updating for intent-based networking. In *2017 International Conference on Information Networking (ICOIN)*, pages 97–102, 2017.

[31] Emil Vassev. *ASSL: Autonomic System Specification Language – A Framework for Specification and Code Generation of Autonomic Systems*. LAP Lambert Academic Publishing, November 2009. ISBN: 3-838-31383-6.

[32] Emil Vassev, Michael G. Hinchey, and Joey Paquet. Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track*, pages 1652–1657, New York, NY, USA, March 2008. ACM.

[33] Emil Vassev and Mike Hinchey. ASSL specification and code generation of self-healing behavior for NASA swarm-based systems. In *Proceedings of the 6th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'09)*, pages 77–86. IEEE Computer Society, 2009.

[34] Emil Vassev and Mike Hinchey. ASSL specification model for the image-processing behavior in the NASA Voyager mission. Technical report, Lero - The Irish Software Engineering Research Center, 2009.

[35] Emil Vassev and Mike Hinchey. ASSL: A software engineering approach to autonomic computing. *IEEE Computer*, 42(6):90–93, 2009.

[36] Emil Vassev and Mike Hinchey. Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. In *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09)*, pages 246–253. IEEE Computer Society, 2009.

[37] Emil Vassev and Mike Hinchey. Developing experimental models for NASA missions with ASSL. *Electronic Proceedings in Theoretical Computer Science*, 20:88–94, mar 2010.

[38] Emil Vassev, Mike Hinchey, and Joey Paquet. A Self-Scheduling Model for NASA Swarm-Based Exploration Missions using ASSL. In *Proceedings of the Fifth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'08)*, pages 54–64. IEEE Computer Society, April 2008.

[39] Emil Vassev, Heng Kuang, Olga Ormandjieva, and Joey Paquet. Reactive, distributed and autonomic computing aspects of AS-TRM. In Joaquim Filipe, Boris Shishkov, and Markus Helfert, editors, *Proceedings of The 1st International Conference on Software and Data Technologies (ICSOFT2006)*, pages 196–202. INSTICC Press, September 2006.

[40] Emil Vassev and Serguei A. Mokhov. Developing Autonomic Properties for Distributed Pattern-Recognition Systems with ASSL: A Distributed MARF Case Study. *LNCS Transactions on Computational Science, Special Issue on Advances in Autonomic Computing: Formal Engineering Methods for Nature-Inspired Computing Systems*, XV(7050):130–157, 2012. Accepted in 2010; appeared February 2012.

[41] Emil Vassev, Olga Ormandjieva, and Joey Paquet. ASSL specification of reliability self-assessment in the AS-TRM. In *Proceedings of the 2nd International Conference on Software and Data Technologies (ICSOFT 2007)*, pages 198–206, July 2007.

[42] Emil I. Vassev. *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD thesis, Department of Computer Science and Software Engineering, 2008.

[43] Shaoqiang Wang, DongSheng Xu, and ShiLiang Yan. Analysis and application of wireshark in tcp/ip protocol teaching. In *2010 International Conference on E-Health Networking Digital Ecosystems and Technologies (EDT)*, volume 2, pages 269–272, 2010.

[44] Qin Wu, Stephane Litkowski, Luis Tomotaki, and Kenichi Ogaki. YANG Data Model for L3VPN Service Delivery. RFC 8299, January 2018.

[45] Engin Zeydan and Yekta Turk. Recent Advances in Intent-Based Networking: A Survey. In *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pages 1–5, 2020.