IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received September 29, 2021, accepted November 10, 2021, date of publication November 23, 2021,
date of current version December 9, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3129990

Towards a Self-Driving Management System for the Automated Realization of Intents

KRISTINA DZEPAROSKA[1], (Member, IEEE), NASIM BEIGI-MOHAMMADI[1], (Member, IEEE), ALI TIZGHADAM[2], (Member, IEEE), AND ALBERTO LEON-GARCIA[1], (Life Fellow, IEEE)
[1]Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON M5S 1A1, Canada
[2]Department of Technology Strategy and Business Transformation, TELUS Communications Inc., Toronto, ON M4C 1L7, Canada

Corresponding author: Kristina Dzeparoska (kristina.dzeparoska@mail.utoronto.ca)

This work was supported in part by the TELUS Communications Inc., and in part by the Natural Sciences and Engineering Research Council (NSERC) Collaborative Research and Development.

ABSTRACT Network management faces the interrelated challenges of increasing network complexity, meeting sophisticated business requirements, and being subject to human oversight. Self-driving networks possess the key properties to overcome such challenges. We present and implement a management system that addresses several elements of a self-driving network. Our system leverages intents, a policy-based paradigm and autonomic control loops. Intent-based networking allows us to formalize how an intent can be provided as input to a control loop, and how the complexities can be abstracted from the user. To realize and assure the intent, autonomic networking enables us to create Monitor-Analyze-Plan-Execute (MAPE) loops. Finally, we execute the control loops using a policy-based approach. We propose a policy abstraction to support requirements at different levels of abstraction, and an Application Programming Interface (API) layer to reduce management complexity from the user perspective. We propose a formal policy information model to model policies across layers of abstractions and to support simplified mapping and strong consistencies among various policy abstraction levels. We have implemented our proposal and present a proof-of-concept use-case to showcase the intent refinement.

INDEX TERMS Autonomic networks, intent-based networking, network management, policy-based management, self-driving network.

I. INTRODUCTION

Network management poses a major challenge in today's environment where complex and large-scale networks must support diverse applications and services with stringent service levels. In this current ecosystem, demands for performance, availability, reliability, and security are becoming increasingly difficult to deliver to the levels required by existing and emerging applications. The next generation management system must now exercise software control that operates on heterogenous and distributed computing and networking resources spanning multiple layers that include wireless access, virtualized computing, electronic routing and switching, and optical paths. And, as business requirements become more sophisticated, it will be increasingly difficult for human operators to translate the multiplicity of requirements into the numerous changes that need to be made in the network software and equipment. It is evident that there is a need for an automated, and practically human-independent way of translating and realizing business requirements. In addition, the dynamicity of networks, rapid traffic changes and system failures, require networks to be self-adaptive so that business requirements can be met continuously from deployment to termination. Clearly the era of manual network management is ending and the advent of network automation is upon us [1]–[6]. In general, the challenges that network management is faced with today call for substantial changes that include a more data-driven, and autonomic approach, that is, one where networks learn to drive themselves.

A self-driving network is a network that can measure and control itself, with the control part relying on learning and data analytics [5]. A self-driving network would include the following four elements: one, the use of high-level user-provided intents to guide the network; two, the use of control loops to realize and assure intents; three, the execution of the control loops; four, the use of learning strategies to adjust and adapt the control loops to environmental changes. In this

The associate editor coordinating the review of this manuscript and approving it for publication was Eyuphan Bulut.

paper we explore the first three elements, and leave the fourth for future work.

Our primary motivation is to overcome network complexity challenges associated with business requirements and human oversight, while taking into consideration the elements of a self-driving network. To address this we leverage a set of well-established contributions such as intent-based networking, autonomic computing, policy-based management and recent enabler technologies. Our focus in this paper is the design and implementation of an automated, intent-based management system to intelligently refine intents into configurable actions through a set of autonomically derived policies.

For this, our work has the following objectives. First, the system should understand and process high-level user requirements in the form of intents. An intent is an abstract, high-level objective, often stated in human-natural language, that specifies what a system should do, without specifying how. In particular, business requirements can be expressed using intents. Second, there needs to be a mechanism for intelligent conversion of intent into a set of policies. Third, intent assurance requires the ability to detect and adapt, hence autonomic management becomes a must. The emergence of network softwarization, enhanced device programmability, massive scale monitoring, and big data analytics and machine learning provide the foundation for autonomic management. A well-known approach for autonomic management is the MAPE-K control loop that delivers the self-x properties (self-healing, self-configuration, self-optimization and self-protection) [7], [8]. The above objectives correspond to the first two elements of a self-driving network.

Fourth, the system should be able to execute the control loops, and for this we turn to a policy-based approach. Policy-based Management (PBM) is the current approach for network management [9] that typically involves action policies (specified as event-condition-action (ECA) rules). PBM itself does not currently support autonomic behaviour primarily because the enabling technologies have only recently become available. This fourth objective corresponds to the third element of a self-driving network which we realize as follows. We define a policy information model to support the conversion from intent to low-level action policies, and to ensure synchronization and strong mapping across abstractions. The policy model should deal with policy definers (i.e., authors) at various abstraction levels, and ideally it should represent various forms of policies (including goal and utility, in addition to action) that can be involved to refine an intent. To establish a coherent flow of policies we define a hierarchical policy representation. For each refined intent, there is a policy tree that includes the policies generated during the refinement of the intent, and as such the policy tree represents the policy execution flow within the control loops. The fifth and final objective is that the system should have easily extendable functionalities and logic. For this we organize our system in a three-layered, hierarchical architecture that consists of applications (that accept intents), APIs, and platform components. The applications and platform components offer functionalities that are enabled through workflows, whereby each workflow is organized as a set of sequenced jobs.

Our contributions are as follows:

- We propose a system that realizes the first three elements of a self-driving network. Our proposal receives as input intents, and refines and assures the intents through control loops. During refinement, each intent is transformed and then decomposed into a set of policies.
- We propose a policy abstraction and an API layer to support user input at different levels of abstraction and to process the policies.
- We define a formal unified policy model for modeling policies at any layer of abstraction within our management system. The model is flexible enough to support a wide variety of network and service management policies in the form of goal, utility, and action policies. The model ensures consistency and integrity of policies across various layers of abstractions with various roles.
- We have implemented our proposal towards a self-driving management system to showcase the practical integration of intent-based networking, PBM and control loops. We present a network intent use-case to demonstrate the intent refinement using our approach.

The rest of the paper is organized as follows: we review related work in Section II, starting with PBM and Autonomic Network Management followed by Policy models, IBN proposals and self-driving networks. We provide summary tables of IBN-related proposals closest to our work. In Section III, we outline our proposal for an Autonomic Management System, in which we introduce the architecture and its associated components followed by policy abstraction and API layer, and we conclude the section by explaining the intent refinement concept. Next, in Section IV, we present our policy model first, followed by an integrity and consistency analysis to ensure quality of policies, and end the Section with a processing model to generate policies for intent refinement, and define a Policy Tree. Finally, in Section V, we provide a proof of concept to demonstrate the refinement of an intent, and a resulting policy tree, followed by a discussion on performance and use-case generalization. Last, we provide our conclusions, and future work directions in Section VI.

II. RELATED WORK

We review policy-based and autonomic network management first, followed by policy models and intent-based networking, and conclude with self-driving networks. We also provide two summary tables of IBN-related proposals.

A. POLICY-BASED MANAGEMENT (PBM)

PBM separates the rules that govern system behaviour from system functionalities, with the goal to allow system behaviour to adapt without having to restart the system [9]. Typical PBM components are: a policy management tool (PMT) to define policies, a policy repository (PR) to store policies, event monitoring, a policy decision point (PDP) to interpret and determine policies to be enforced, and agents or policy enforcement points (PEP) to enforce policies.

In general, there are two well-established policy types: high-level (human-friendly) and low-level action policies (domain or device-related) [10]. According to Kephart and Walsh [11] high-level policies are further divided into goal and utility function policies, a classification inspired by knowledge-based agents [12]. An *action policy* dictates a single or set of actions to be taken when the system is in some state, i.e. *if (condition), then (action)*, where condition specifies a state or set of possible states that satisfy the condition. A *goal policy* specifies either a single desired state, or one or more criteria to characterize a set of equally desirable states. A *utility policy* is realized through a utility function that assigns a utility value to each possible state, hence it generalizes goal policies that perform binary classification. Utility policies are more powerful than goal and action, but entail a recurring computation to select the state with highest utility. Utility and goal policies are enforced through action policies. In PBM, action policies in the form of ECA are frequently used [9].

Differences in PBM proposals originate from factors such as targeted systems, policy models, architecture, technologies, or use-case dependencies. For example, [13] focuses on PMT to propose validation and transformation of business-level policies to technology-level policies. The policy translation method is defined per-discipline, making the method less generic as rules need to be defined for each discipline. Verma *et al.* [14] enabled autonomic behavior within devices so that devices can generate policies for their operations; each device must have local policy refinement, PDP and PEP, resulting in a localized approach that does not account for global network states. Wickboldt *et al.* [15] presents Software-Defined Networking (SDN) management requirements and challenges, and identifies more SDN-focused research is needed to better understand the PBM scope within SDN.

Our proposed autonomic management system combines PBM and MAPE-K control loops. We provide a mapping of our system to the PBM framework. We separate some of the combined capabilities in the PBM framework components and by doing so we ensure a clear separation between component functions. Based on our design choices, multiple components can enforce policies at different levels of abstractions, hence the PEP functionality in our system is distributed.

B. AUTONOMIC NETWORK MANAGEMENT

Autonomic computing (AC) can facilitate management and operations of complex computing systems and reduce the level of human support required [16]. An AC architecture to deliver the self-x properties is the *MAPE-K* closed-control loop [7] which enables sequential loops involving *Monitor-Analyze-Plan-Execute*, supported by *Knowledge*. The application of the autonomy concept to network management results in an autonomic network management system (ANMS) [17].

ANMS-related surveys, challenges and proposals are discussed in [16]–[23], in general there are differences in architectures, policy models, scope and requirements, and target systems. For example, FOCALE [24] uses control loops (with a PBM-based approach to some of the components) and the DEN-ng information model and policy continuum. ANEMA [25] uses goal, behavioral and utility function policies (behavioral policies are used to guide network devices to react to changes), with a per-device approach that does not consider network-wide decisions. STAR [26] is an autonomic resource management proposal with a focus on SLA violations. IETF's Autonomic Networking Integrated Model and Approach (ANIMA) [27] working group explores autonomic networking of managed devices with the goal to provide the self-x properties, however, control loops are currently not covered. Some proposals include learning ([21], [28]–[32]), for example [21] uses agents to implement cognitive control loops, and [32] proposes a multi-agent network automation architecture that follows the Generic Autonomic Network Architecture (GANA) reference model from the European Telecommunication Standards Institution (ETSI) [33]. The failure to deploy autonomic network management in the past is discussed and reconsidered in light of new enabler technologies in [23]. Research efforts in [34]–[39] explore the potential to combining ANMS and SDN. SDN controllers that communicate south-bound with data plane devices, and north-bound with applications, provide the hierarchy and interfaces to support autonomic management.

We propose an architecture for autonomic management where the logic is separated from the functionalities offered by the system, and is defined per application to achieve any management task. To accomplish this objective, we propose a policy abstraction to capture user requirements at different levels. The policy abstraction is presented through a unified formal policy model. We propose an API layer to provide functionalities needed to process the policy abstraction. The proposed autonomic management system is flexible to cover a wide array of management scenarios due to its well-defined and modular components that can be easily extended.

C. POLICY MODELS

A policy information model facilitates a policy language for formal representation of policies. The IETF/DMTF Policy Common Information Model (PCIM) [40] is used to represent device and application characteristics to enable vendor-agnostic device control. Strassner [41] proposed the Directory Enabled Networks (DEN)-ng information model that models policy based on ECA paradigm across policy continuum. Although DEN-ng was extended in [42] with a *PolicyRuleComponentStructure* so that specific rule structures can be defined for non-ECA policies, it does not natively model goal and utility policies. NOVI [43] is another information model for network management that supports virtualization and federation and models policy using ECA.

A survey on policy languages in network and security management is presented in [10]. Beckett *et al.* [44] proposed the Probane language to help network admins specify routing policies in a high-level manner. Probane encodes the flow of routing information along policy-compliant paths and raises the level of abstraction for imperative policies related to traffic engineering (TE) tasks. Frenetic [45] provides a high-level language for TE that aims to ease the work of network operator, however, both languages are designed for OpenFlow-based TE, and it is not clear if they can represent network engineering policies [46]. The lack of formalism and heterogeneity in intent and policy processing in [47]–[49] incur sophisticated mapping functions and analysis to map the attributes among policy abstraction types. This additionally introduces the challenge of inconsistency and synchronization across layers of abstraction [50]. The languages proposed in [51]–[53] are specifically tailored to model QoS and access control policies.

The policy models discussed above do not account for the policies involved in the whole intent life-cycle, including policies in the assurance loop [4]. Most focus on the high-level requirement translation and intent formulation, and most are specifically designed to model traffic or network engineering tasks, which limits their applicability to represent a wide variety of policies in autonomic management. Further, they do not capture various forms of policies including goal, utility and action at the same time.

Our model models the policies involved in the complete intent life-cycle. Due to our generic policy formalism, our model is capable to represent various forms of policies. Last, our unified model models policy across layers of abstraction which allows for simplified mapping and strong synchronization among various policies during deployment and run-time.

D. INTENT-BASED NETWORKING (IBN)

To the best of our knowledge the first IBN standardisation effort was made by the Open Networking Foundation (ONF) with their proposal to fill the northbound interface (NBI) gap between SDN controllers and SDN applications [54]. In addition to ONF, the 3rd Generation Partnership Project (3GPP), European Telecommunications Standards Institute (ETSI), International Telecommunication Union (ITU) and Internet Engineering Task Force (IETF) have their own IBN-related groups as well. For example, the IETF Network Management Research Group has several efforts underway [55]–[57] to define intent, the intent life-cycle, and intent assurance, as well as the relation between intent and policy, and the use of control loops. A recent survey on existing intent-based frameworks, activities and standardization efforts, as well as challenges is given in [58].

A number of well-known open-source projects such as NEMO, ONOS Intent Framework, OpenDaylight NIC, OpenStack's GBP and Congress have explored IBN, and have implemented a limited set of intents that are focused mainly around SDN-based network capabilities. In academia, proposals have focused on different aspects of IBN, for example DOVE [59] is a network virtualization proposal for multi-tenant network services that uses an intent-based approach. Han *et al.* [48] presents an intent-based VN management platform for network slicing and automated configuration with SDN. Intent-based NBIs are presented in DISMI [60] for controllers to provide application-centric networking, and in [61] to orchestrate VNF service chaining. Similarly, the iNDIRA tool [62] interacts with SDN NBIs to support IBN, and INSPIRE [63] refines intents into VNF service chains. Other research proposals for IBN that present IBN frameworks are included in [46], [64]–[75]. Although SDN seems to be the typical architecture for implementing IBN, IBN should not be limited to SDN [4].

Intent or policy refinement are concepts that exist in the literature to refer to the processes that transform either an intent or a policy to formal policies that can be subsequently decomposed to actions. For example, INSPIRE [63] considers intents as high-level abstract policies that IBN should translate to low-level configurations, Moffett and Sloman [76] explored the refinement of high-level policies into a set of more specific policies to form a policy hierarchy, Craven *et al.* [77] described a method for policy refinement that consists of decomposition, operationalization, deployment and re-refinement, and Jacobs *et al.* [49] refines natural language intents into network configurations (using ML to translate intents to the Nile intent language first, and then to SONATA-NFV commands).

PBM can facilitate the policy aspect of an intent, such that an intent can be uniformly realized through a set of policies, and an autonomic control loop can be used to ensure the self-x properties during the intent's life-cycle. IBN can be a solution towards an intelligent network that automatically converts, deploys and optimizes itself to achieve a target state based on an intent, however to attain this, IBN has some challenges that need to be addressed, e.g, continuous closed-loop verification, automated deployment, conflicts [4].

In Tables 1 and 2, we provide a summary of IBN-related proposals and their main features. None of these proposals consider the complete set of objectives that our paper seeks to realize, in order to support automated intent realization.

Our proposal provides intent refinement and assurance through a PBM paradigm and control loops. We define a set of abstractions that allow clear mapping during the intent refinement, and we detect inconsistencies using our formal policy model that is applicable during the intent's life-cycle.

E. SELF-DRIVING NETWORKS

A self-driving network can measure, analyse and control itself such that the network can predict changes and adapt without the intervention of an operator [5], [6], [49], [78]–[80]. Such a network should take as input a high-level goal and automatically generate the necessary measurements, inferences and decisions to execute [5], [81]. The above outlines

TABLE 1. Summary table of intent-based networking proposals.

Proposal, Intent Specification *(language, formal model)*, **Architecture, Refinement**	**Intent Validation**	**Intent Assurance**	**Supported Intents**	**Motivation**
Cohen et al. [59]: An intent-based network management abstraction proposal realized over a network virtualization architecture (DOVE: Distributed Overlay Virtual Ethernet network). Intent specification using network blueprints given as *directed graphs where vertices represent policy domains, and edges represent policies.* Main architecture: DOVE policy controllers (maintain blueprints, resolve, validate), and DOVE switches (enforce policies). Resolve, validate and map policies to network device instructions for deployment.	1) DPC validates policy requests and details (IP, MAC)	Not Specified	Network (connectivity services), QoS, security	Network virtualization
Han et al. [48]: An intent-based virtual network (VN) management platform proposal. Intent format: an object containing *resource, condition, priority, instruction* (object is based on the ONOS intent framework). Architecture layers: Intent (manage, compile, map, conflict check); Virtual abstraction (for tenant VNs), Virtualization (translate VN objects to physical objects), Abstraction (abstract protocols, infrastructure) and Protocol adaptation (install flow rules) layer. Compose, translate and map intents to network details for intent deployment.	1) Conflict resolution during intent translation and composition	Not Specified	Virtual Network (e.g., connectivity, network service chaining)	Network virtualization
DISMI [60]: Dynamic Intent-driven Service Management Interface An intent-based, application-centric networking proposal for transport networks. Intent Model: *Action, Constraint, Calendaring, Priority.* Model is supported by the main classes of primitives (verbs, nouns, modifiers) and their respective sub-classes. Main components: DISMI database, Service Validator, Intent Compiler, SDN Controller. Convert and decompose DISMI primitives to simpler primitives, and then, to ONOS intents.	1) Conformance checks 2) Conversion to network details	Not Specified	Network connectivity (e.g., bandwidth, delay, availability, calendaring)	IBN for transport networks
iNDIRA [62]: Intelligent Network Deployment Intent Renderer Application An intent-based service description framework for automated provisioning and network control. Intent input is provided via templates, or high-level queries (using a chat bot). Intents are modeled using an Intent RDF graph composed of: *subject (service or condition), relationship (has arguments), and objects (parameters).* Architecture: Intent parser, Ontology checker, Intent renderer, Other Renderers: topology, time, bandwidth, and provision renderer). Uses ontology to translate intents to RDF graphs, that are then rendered to network configurations.	1) Conflict Checks 2) Permission Checks	Not Specified: To some extent through automatic device config. to satisfy intents.	Network provisioning services (e.g., connect, tap, disconnect, QoS), and file transfer services	Multi-domain network control
INSpIRE [63]: Integrated NFV-based Intent Refinement Environment An intent refinement proposal that translates intents into service function chains (SFC). Intents are composed using a controlled language *(service, flow, preposition, expression).* Architecture: Traffic Classifier (traffic objects for traffic steering), Service Chaining (context objects for SFC), and Service Chain Graph Builder (cluster, select, and chain VNFs). Translates intents to context and traffic objects, that are used to construct the SFC and to steer traffic.	1) Policy Validation (syntax) 2) Conflict Detection	Not Specified	Service function chain (e.g., QoS, security)	Refine intents into service function chains composed of VNFs and physical middleboxes
Robotron [64]: A Network Management System to translate design intents to device configurations. Network component data model: *Object, Value (object data), Relationship (reference to other objects).* Management Architecture: Network Design, Configuration Generation, Deployment, Monitoring (to ensure desired state) and FBNet (a central information repository). Translate high-level designs into object changes, then build configurations based on object states, and deploy configurations to network devices.	1) Network design checks (data integrity) 2) User checks 3) Monitoring	Not Specified: To some extent through Monitoring	Network configurations (e.g., IP, switches, circuits)	Reduce management effort and error by minimizing human interaction with network devices

TABLE 2. Summary table of intent-based networking proposals.

Proposal, Intent Specification *(language, formal model)*, **Architecture, Refinement**	**Intent Validation**	**Intent Assurance**	**Supported Intents**	**Motivation**
Davoli et al. [66]: An intent-based framework for end-to-end, multi-domain service management. Service function chaining (SFC) template (ETSI MANO inspired): *source, destination, QoS, VNF list.* REST API: action (add, update, delete), SFC direction, and message containing the SFC template. Architecture (ETSI MANO compliant): VNF Manager and NFV Orchestrator (to program VIMs and SDN controllers), Virtualized Infrastructure Manager (VIM) (manage resources), SDN controller. Translate, resolve and map high-level specifications to ONOS intents to be compiled to flow rules.	1) Data plane monitoring service	Not Specified	Service function chaining of virtual network functions	Multi-domain end-to-end service management
Janus [46]: An intent-based management proposal that extends PGA to represent QoS and dynamic policies, and explores policy configuration as an optimization problem. Policy Graph Abstraction (PGA) [53] is a policy management framework that expresses policies as *policy graphs: vertex (endpoint group), edge (rules).* Janus leverages PGA to express and compose intents, and extends the policy graph with *edge attributes (e.g., metric values, logical labels).* Main components: Interface, Graph Composer, Policy Configurator (perform optimizations to maximize number of satisfied policies and minimize path changes; generate network configurations). Translate intents to policy graphs, compose policy graphs, and generate network configurations.	1) Graph composition analysis 2) Conflict resolution	Not Specified	Dynamic and network QoS (e.g., bandwidth, jitter, latency)	Graph-based policy representation
LMS [73]: A Label Management System (LMS) that maintains relationships and interactions between network entities to support intent composition and translation. Defines label naming syntax, label namespace, and label trees to represent network node dependencies. Architecture: Applications, Intent-based framework (e.g., PGA [53]), LMS (Label Namespace Builder, Label Mapper, Label Repository), and Cloud Management System (e.g., OpenStack). A policy management framework (e.g, PGA) can use the labels and label trees to specify high-level policy intents, compose policies, and generate infrastructure rules.	Not Specified	Not Specified	Network-related (traffic control per network protocol, e.g., HTTP, SSH)	Intent-driven cloud management
Jacobs et al. [49]: An intent refinement proposal that uses machine learning to translate intents, and uses operator feedback to improve the model. Intent representation using *Nile*, an intent language with an extensible grammar *(e.g., command, QoS, metric, constraint, rules, targets, location, five tuple, date time).* Refinement components: Entities Extractor, Intent Translator, Intent Deployer. Chat-bot interface to extract entities from user intents. Translate the extracted entities to Nile using neural sequence-to-sequence model. Receive operator feedback (to validate intent, and improve the model). Last, compile the Nile intent to a network policy for deployment.	1) Operator validation of Nile intent 2) Conflict check (extracted intent vs network configuration)	Not Specified	Network-related (e.g., SLA, QoS, network security, performance)	Intent refinement as a step towards enabling self-driving networks
Our Proposal: An Autonomic Management System (AMS) proposal for automated intent realization. Policy-based Monitor-Analyze-Plan-Execute control loop execution for intent refinement and assurance. Policy Model: *Definer, Enforcer, Action, Constraint (Resource, Temporal, Spatial)*. Supports utility, goal and ECA policies. Policy abstraction and API layer: Declarative, Definitive, and Imperative. Architecture overview: Applications, API Layer, AMP (MAPE-k components), and Infrastructure. Intents are transformed to declarative policies first, that are then decomposed to definitive policies. Definitive policies decompose to one or more definitive and/or imperative policies. Imperative policies, once executed, realize the intent. Policy trees are created during refinement, with intent as root node, and policies (declarative, definitive, and imperative) as child nodes that are organized by their order of enforcement.	1) Sanity check (integrity, permissions) 2) Consistency analysis (conflict detection & resolution) 3) Monitoring state 4) *Network Twin (*future work)	Supported: Through Control Loops and Policy Trees	Network, compute, and storage (e.g., network optimization, traffic and network engineering, monitoring, management)	Towards a self-driving management system for automated intent realization

four elements: high-level goal to specify desired behaviour, autonomic control loop to realize the goal, execution of the control loop, and learning algorithms for inference.

Currently, there is no accepted framework yet, and most of the literature focuses on specific elements of a self-driving network, for instance Jacobs *et al.* [49] focuses on the intent refinement to translate intents expressed in natural languages; Madanapalli *et al.* [82] explores quality of experience by detecting deteriorated states and applying corrective actions; Zerwas *et al.* [83] proposes self-driving network benchmarks; and Pasandi and Nadeem [84] proposes a self-driving approach for the design and evaluation of network protocols.

To develop a reliable data-driven self-driving network, data quality and ML are required. In this context [85] surveys the use of unsupervised learning for next-generation networks. Proposals focused on learning include: deep reinforcement learning to provision and coordinate VNF services [86]; reinforcement learning to optimize service provisioning policies for the optical domain [87]; [88] uses a measure-learn-decide-action control loop within data and control planes for local control, and a management plane to revise globally; [89] explores the adaptability in SDN, NFV through ML-enhanced observation, composition, and control; and, [90] studies fault detection for IoT networks.

The focus of our work is on the integration of the first three elements of a self-driving network: receive high-level goals (intents) as inputs, use control loops to realize goals, and use policies for the execution of the loops. Our proposal incorporates these into a rule-based starting point first, with future work to include the fourth element, learning.

III. AUTONOMIC MANAGEMENT SYSTEM

In this section, we describe our proposed *Autonomic Management System (AMS)* for automated resolution of intents. First, we summarize our objectives for this system: (a) to support and to realize intents; (b) to have a mechanism for intelligent conversion of intent into a set of policies; (c) to provide assurance for intents, and to enable the self-x properties for autonomic behaviour, and to comply with existing policies.

Network management may operate on multiple domains. A *domain* represents a local grouping of one or more networks and devices within a collective infrastructure group, typically governed by a single authority. Multi-domain settings imply that there are at least two domains involved, each with its own specific organization in terms of network and infrastructure control. Hence, multi-domain settings can consist of domains under single or multiple authorities.

To support the above objectives we propose the following. First, we use a policy-based approach to realize the requirements imposed by intents. An intent is refined so that it is transformed to *domain*-specific terms first, and then decomposed to determine if, and what the necessary actions should be. Second, for the autonomic management system to support high-level user inputs, and to simplify overall management, we propose a policy abstraction and an API layer. The *policy abstraction* allows users of the autonomic management system (humans or applications) to define policies at different levels of abstraction. The *API Layer* exposes the platform functionalities to the applications, moreover it abstracts the details and unifies the format, and supports the intent refinement and policy decomposition. Third, an autonomic control loop to assure intents and to provide the self-x properties. Thus, the system enables creation of MAPE loops for a specific logic, so that it can monitor, analyze, plan and execute to transition to desired states in compliance with policies and intents.

Figure 1 shows our autonomic management system which has three layers: Applications (top layer), Application Programming Interfaces (middle layer) and Autonomic Management Platform (AMP) (bottom layer).

Users can interact with the AMP through the applications. A user can be internal (e.g. the service provider itself), or external (e.g. a customer of the service provider such as an enterprise). Users can choose from a pre-loaded set of intents that are offered as a catalog of intents, or, users can build new intents and add to the catalog. The latter requires careful consideration of rules and vocabulary, so as to ensure that new intents can be transformed to domain-specific terms. Applications are built by application developers who can be internal or external to the described system. Each application is built to support one or more related intents, e.g. an application can offer a VPN service to an external user, where the user can define specifics such as over the top SD-WAN VPN with real-time application oriented routing and support of a number of QoS classes. Other example applications include: an operator service for maximizing or minimizing core network throughput (an example of such a network intent is presented in Section V, to showcase the intent refinement process); a monitoring service; a fast MAPE-K control loop service to ensure existing intents are met.

Each application has an Autonomic Manager (AM) that consists of application-specific MAPE-K components. Applications consume infrastructure resources through AMP services to deliver content and services to their users according to their business objectives defined by intent. The AMP services consist of platform MAPE-K components to manage the infrastructure. These services are made available to applications through the API layer. As such, an AM can use APIs from the AMP to form the required feedback control loop. To support and guide the AM, as well as to interact with the platform, each application needs to have a pre-defined workflow logic. A *workflow* is an organized set of jobs that are sequenced in a specific order and with a set of checks and balances.

In general, the platform may have different workflows to support commonly requested services. Using the APIs, applications can define and use their own workflows, use the platform workflows, or use a hybrid approach. Any of these approaches enables per application MAPE-K control loops to be instantiated. A loop requires enforcing a set of

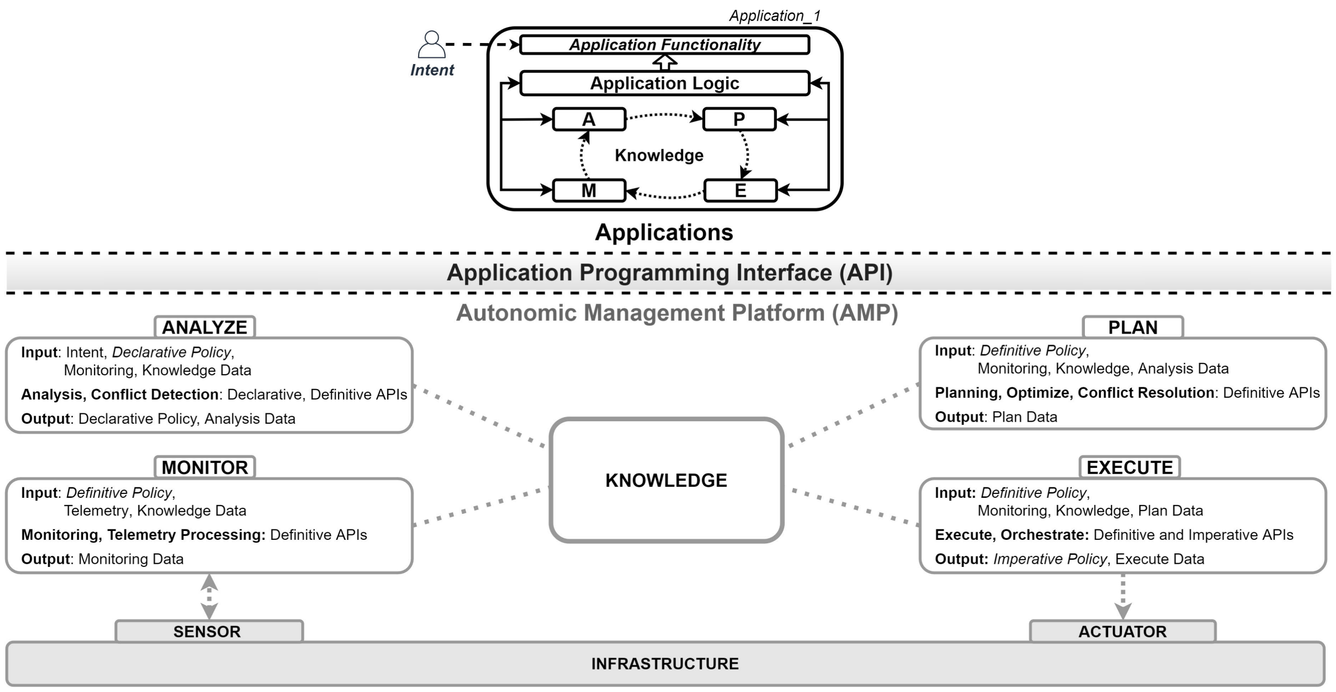

Application_1
Intent
Application Functionality
Application Logic
A
P
Knowledge
M
E
Applications
Application Programming Interface (API)
Autonomic Management Platform (AMP)
ANALYZE
Input: Intent, Declarative Policy, Monitoring, Knowledge Data
Analysis, Conflict Detection: Declarative, Definitive APIs
Output: Declarative Policy, Analysis Data
PLAN
Input: Definitive Policy, Monitoring, Knowledge, Analysis Data
Planning, Optimize, Conflict Resolution: Definitive APIs
Output: Plan Data
KNOWLEDGE
MONITOR
Input: Definitive Policy, Telemetry, Knowledge Data
Monitoring, Telemetry Processing: Definitive APIs
Output: Monitoring Data
EXECUTE
Input: Definitive Policy, Monitoring, Knowledge, Plan Data
Execute, Orchestrate: Definitive and Imperative APIs
Output: Imperative Policy, Execute Data
SENSOR
ACTUATOR
INFRASTRUCTURE

FIGURE 1. Overview of the Autonomic Management System (AMS). Applications use Application Programming Interfaces (APIs) to invoke the Autonomic Management Platform (AMP) components and their respective functions, to enable per application control loops.

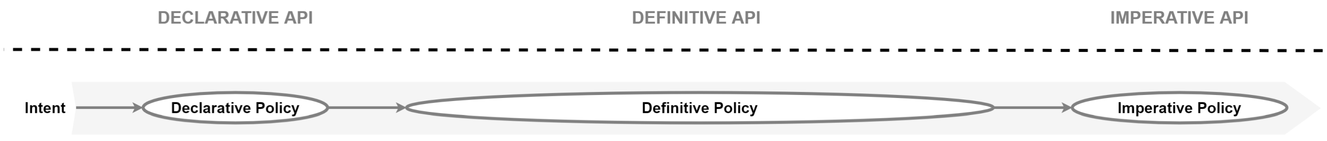


FIGURE 2. Intent refinement through the Policy Abstraction and API layer. The refinement occurs from left to right: each policy abstraction type is processed using the corresponding API class (shown on top of each policy abstraction type). The Declarative API is used to transform an intent and generate a Declarative Policy; the Definitive API to generate and enforce Definitive Policies, and the Imperative API to generate and enforce Imperative policies that realize the intent. Applications shown in Figure 1 use the API classes to invoke the AMP components.

policies **within each component** to derive the actions needed to realize the user input.

A. POLICY ABSTRACTION AND API LAYER

Next, we describe the policy abstraction that is used to specify the requirements at different abstraction levels, followed by a description of the API layer. Before getting to the details of policy abstraction, we first overview intent and policy definitions which lead us to policy abstraction.

In general, an Intent specifies *what should be done, not how*, in a high-level manner, typically using human-natural language. An intent needs to be refined before it can be realized and enforced within a single domain, multi-domain or federation settings. In our Autonomic Management System (AMS), an intent must first be transformed, and then decomposed to a set of policies. Hence, the output of a transformed intent is a high-level policy in domain-specific terms. A policy is *a formal representation of some desired behaviour (state).* A policy can be decomposed to additional policies, a rule or set of rules to administer, manage and control resources and processes such that a desirable state is achieved.

1) POLICY ABSTRACTION

We propose three policy abstraction types to support user inputs at different levels of abstraction as shown in Figure 2. A *Declarative policy* defines *what should be done, not how* in a high-level manner, and in domain-specific language. It is derived by transforming an intent into domain-specific terms and enhanced with domain-specific knowledge from a system's knowledge base. A set of declarative policies can result from a single intent. For example, an intent in multi-domain or federation settings requires a declarative policy per domain, or per provider and per domain, wherein each declarative policy is represented by domain-specific terminology and knowledge. A *Definitive policy* defines *what should be done* in a granular manner by specifying what should be done by Analyze, Plan and Monitor components, so as to guide the steps towards a desired decision making. A set of definitive policies essentially determines the changes that should be done in the system to achieve the desired outcome. An *Imperative policy* defines *how* and not what should be done, by specifying an action, or a set of actions for execution to attain the intent objective, goal or desired outcome. It is derived and enforced by the Execute

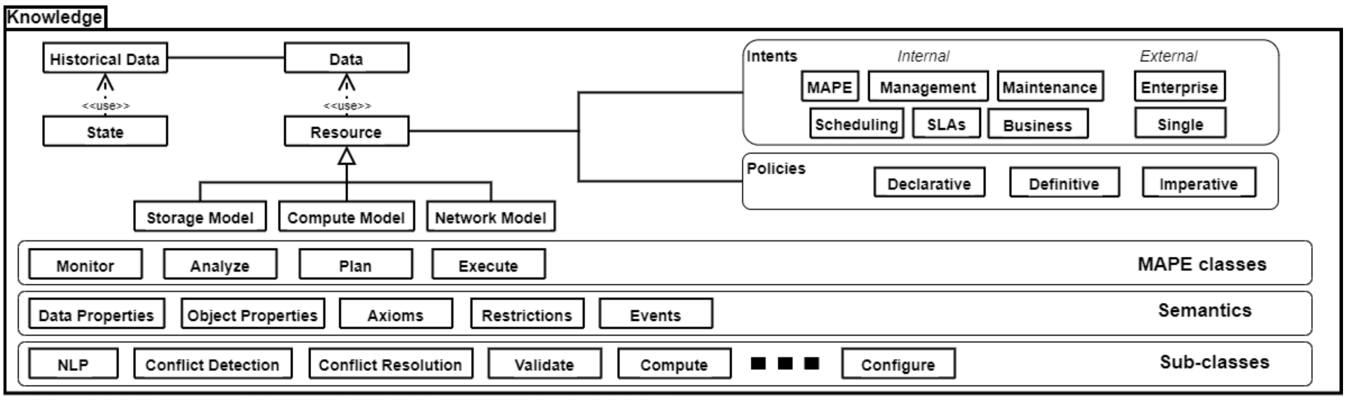


FIGURE 3. UML class diagram for the *Knowledge* component of the AMP.

component based on the necessary changes as determined through definitive policies.

During the intent refinement process the following transitions occur from one policy type to another: first, each intent is transformed to declarative policies. The declarative policies cannot immediately be realized, so they need to be decomposed to definitive policies. Each definitive policy can generate one or more definitive and/or imperative policies. An imperative policy, or a set of imperative policies, once executed, realize the intent. Typically, imperative policies are executed in the form of an action policy (e.g. using any part of the ECA structure discussed in II-A).

2) API LAYER

In Figure 2, we also propose an API layer that consists of three classes. The APIs expose the AMP components and functionalities for use by applications at various levels of abstraction to help reduce management complexity from user perspective. Our management system can receive and refine an intent (input) through any of the policy abstraction types, into single or multiple action policies (output) using the APIs. Each API class corresponds to the processing required by each policy abstraction type and are defined as *Declarative, Definitive and Imperative API*.

The Declarative API provides functions within the MAPE-K components that enable the transformation of an intent to a declarative policy, e.g. Natural Language Processing. The Definitive API involves a diverse set of functionalities provided by Analyze, Plan, Monitor and Knowledge. Through definitive APIs, analysis and planning are performed using monitoring and knowledge data, so that changes that need to be enforced as actions are determined. The Imperative API is used to enforce such actions across infrastructure resources and involves Execute only.

B. AMP COMPONENTS AND FUNCTIONALITIES

We now present the platform MAPE-K components and their functionalities through UML class diagrams. We do not show class methods or attributes, but we do discuss some of their functionalities.

1) KNOWLEDGE

Central to the architecture, Knowledge contains a pool of data that the rest of the components rely on. In Figure 3, we present the UML class diagram for *Knowledge*.

The *Data* class receives input from the *Monitor* component to store pertinent real-time monitoring data that can be consumed by the rest of the MAPE-K components. The monitoring data includes events of interest and processed telemetry data, specific to the enforced monitoring policies. To prevent exhausting storage resources, retention policies are used to specify the data retention period. The UML classes *State* and *Resource* use the *Data* class for updates. *Resource* models all the assets and services that are available for consumption, both virtual and physical, e.g. network, compute, storage. These resources are represented through the *Network Model*, *Compute Model* and *Storage Model* classes. These classes store processed data in their respective repositories. The resource model can be extended to include other resources considering policies of various roles, which we leave for future work. *Policies* maintains policy repositories for the intent refinement process (declarative, definitive, imperative). We consider *Knowledge* to hold both traditional content (e.g. repositories for real-time and historical monitoring data, policy repositories, topology data), as well as custom content and classes as described next.

In order to benefit from powerful and expressive APIs that invoke each component, we extended *Knowledge* to include the following content: the MAPE-K components, their classes, semantics that define properties and relations (e.g. data and object properties, axioms), state repository (records of optimal states). We allow exposing the APIs through our *Knowledge* component. In this manner, the APIs can be used to perform the necessary tasks and to adhere to the self-x properties.

2) MONITOR

This component gathers real-time monitoring data from infrastructure sensors. Based on the type of resource, there could be more than one sensor deployed. The collected data is processed (e.g. normalized, aggregated), and stored in *Knowledge*.

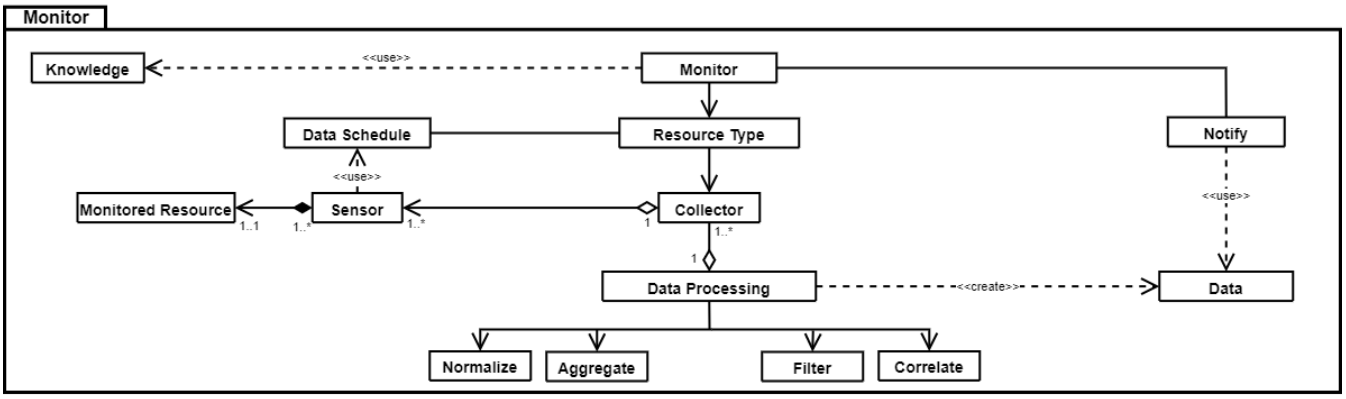


FIGURE 4. UML class diagram for the *Monitor* component of the AMP.

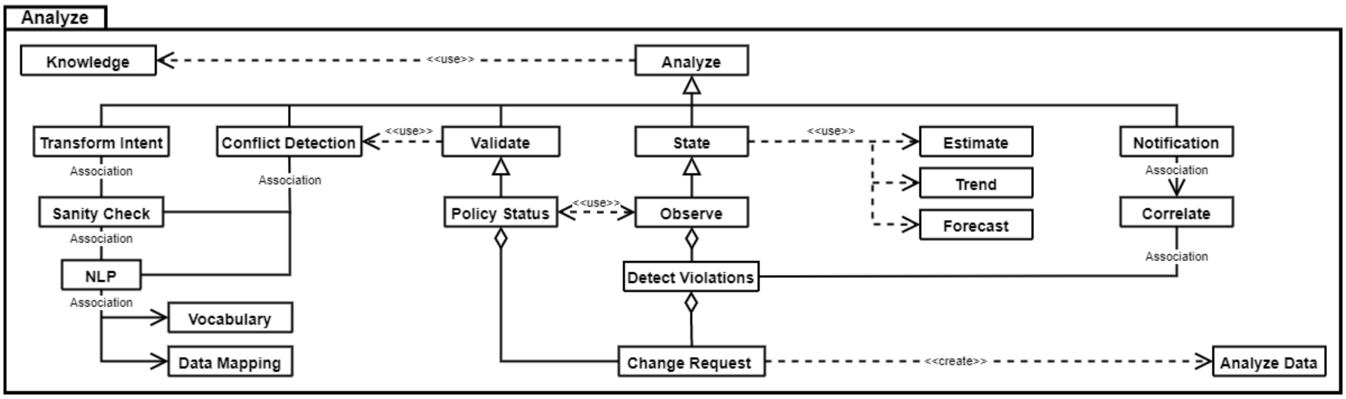


FIGURE 5. UML class diagram for the *Analyze* component of the AMP.

Figure 4 shows the UML class diagram for *Monitor* which uses data (e.g. *Policies* from *Knowledge*) for the *Notify* class to determine and notify about changes in resource state and events of interest. Often these notifications need to be shared with *Analyze*, however the manner in which this occurs depends on the application logic.

To enable data collection, based on *Resource Type*, a *Data Schedule* is created and used by the *Sensor* class. The *Data Schedule* contains data collection metrics that define the required processing on the collected data which is completed by the *Data Processing* class. Figure 4 shows the processing classes, e.g. normalize, aggregate. The output from *Data Processing* creates *Data*, which is stored in *Knowledge*, whereas *Notify* can use this monitoring data directly. If a new type of monitoring data needs to be collected for which sensors have not been configured yet, then, imperative policies for sensor configuration are enforced through *Execute*, followed by definitive policies that are enforced by *Monitor*, so that the new data collection can begin.

3) ANALYZE

This component enables several types of analysis to support intent transformation and policy decomposition. Analyze can be invoked by different inputs to support different types of analysis as shown in Figure 5, for example: 1) Intent transformation and conflict detection, 2) Validation of declarative policies, 3) State analysis, 4) Notification analysis. In general, the output from *Analyze* contains relevant *Analyze Data* (e.g. ordered violations) which is used by *Plan*.

When an intent is provided as input to *Analyze*, the output is a declarative policy. The transformation occurs through several steps, and user feedback may be requested in some steps. Our goal is to eventually have the system minimize human intervention. As shown in Figure 5, the *Transform Intent* class starts the analysis of an intent. Natural Language Processing is used to parse and map the intent to domain-specific language through the *NLP* class which makes use of the *Vocabulary* class. Vocabularies are typically created per application and are available within *Knowledge* to be used by applications. The *Sanity Check* class in conjunction with the *Data Mapping* and *Conflict Detection* classes, help to verify and transform the intent through the following steps:

- **Authentication and Authorization** is required at the user, application, API and platform level to verify privileges for the resource requesting the service.
- **Intent to Policy Mapping** enables the transformation. The *Sanity Check* class is used to verify the format, e.g. if values are within the allowed range. If input values

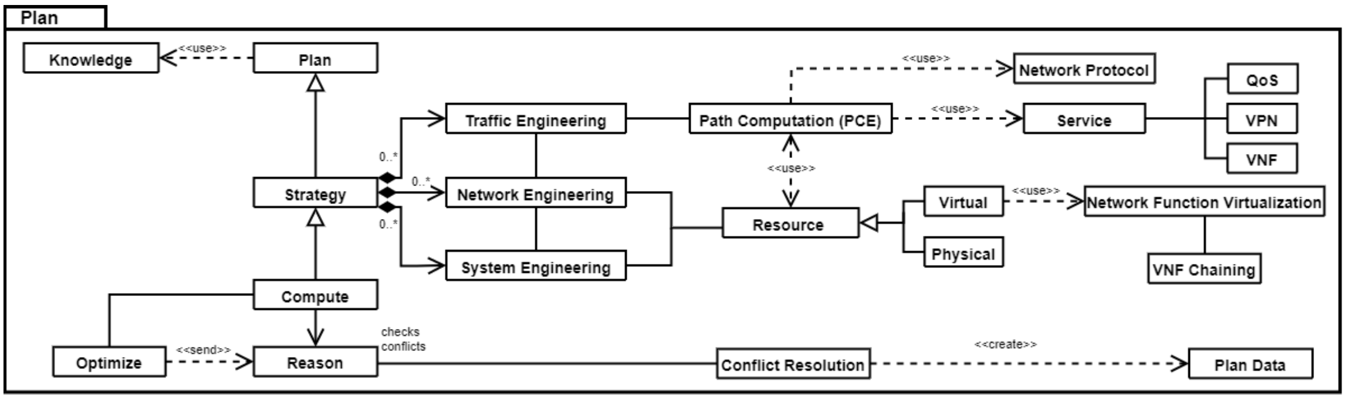


FIGURE 6. UML class diagram for the *Plan* component of the AMP.

are missing for mandatory attributes, or are invalid, user feedback is requested. The intent is processed with *NLP* using the *Vocabulary* class which requires access to adequate vocabularies. The *Data Mapping* class is used to map the output from *NLP* to a data model.[1]

- **Conflict Detection** is used to verify if a declarative policy is in conflict with existing ones. If a conflict is detected, *Plan* needs to be invoked to determine possible conflict resolutions. Based on the conflict type, it can either be solved automatically, or suggestions are presented to the user. The *Conflict Detection* class is not limited to declarative policies, and can be used for any conflict detection required at any layer of policy abstraction.

Upon completion of the above steps, the intent is transformed to one or more declarative policies, that need to be further decomposed to definitive policies. Optionally, declarative policies can be communicated to the user to ensure user's intent is captured. A declarative policy is further analyzed, so that one or more definitive policies are generated, each to perform a specific step of the analysis. Some policies could be defined as goal or utility types of policies, as discussed in II-A. Definitive policies can result with diverse resource analysis, e.g. routing and forwarding tables, traffic demands, resource status (e.g. health, utilization), policy violations and conflicts, permissions and restrictions, business objectives, risk and failure assessments (e.g. Shared Risk Resource Groups). For the above, *Analyze* uses available content from *Knowledge*, e.g. traffic demand data, topology data, policy repositories, resource status, virtual network functions. Through definitive policies, *Analyze* can perform analysis to determine the current or future state of the system and resources with respect to the policy requirements.

We briefly describe some of the other types of analysis: The *Validate* class can be used to process a new declarative policy, or to examine the status of existing intents through their respective policies. Both scenarios are completed by establishing the *Policy Status* using *Observe* under a specific *State*. The policy status is a logical value that specifies whether a policy is enforced or not. If policy status returns "False", it indicates that the policy is not enforced (e.g. it has been deactivated, or as a result of some policy violation). The *Monitor* component can determine events of interest to assist *Analyze* as described before. Such information is provided to the *Notification* class for further analysis and correlation to detect possible policy violations. The *Detect Violations* class is used to check for policy violations.

[1] An information model is presented in Section IV.

4) PLAN

This component receives a definitive policy and uses *Knowledge* data, where the *Analyze Data* is available. *Plan* should determine an optimal solution that minimally affects the other policies. *Plan* follows a similar process as *Analyze*, in that it processes one or more definitive policies. *Plan* can determine possible solutions, and provide conflict resolution. The *Optimize* class is used to review the solutions, and where possible, it can perform different optimizations. The output from *Plan* contains the identified solution as part of the *Plan Data*, which is to be used by *Execute* to enforce the solution.

In Figure 6, we show the UML diagram for *Plan*. *Plan* uses *Knowledge* data to reason about possible strategies. The *Strategy* class is used to invoke the necessary planning classes (e.g. *Traffic, Network or System Engineering*). *Traffic Engineering* is mainly used for finding optimal network paths for services, or optimal network resource utilization. The *Path Computation Engine (PCE)* class is used to find such paths, and it can account for different protocols and services (e.g. *QoS, VPN, VNF*). For the above, *PCE* uses the network model (available from *Knowledge*, as shown in Figure 3) through the *Resource* class in Figure 6. The *Network Engineering* and *System Engineering* classes are used to explore options such as resource or capacity planning. To accomplish this, they consider all the available resources, both virtual and physical. These resources are available within *Knowledge* as shown in Figure 3 under the *Resource* class which includes the *Network Model*, *Compute Model* and *Storage Model*. As an example, the *Virtual* class can use the *Network Function Virtualization* and *VNF chaining* classes.

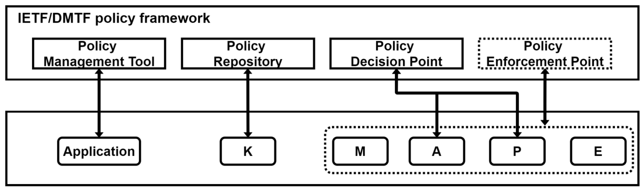


FIGURE 7. Mapping of the general IETF/DMTF PBM framework to our policy framework for autonomic management.

Based on the discussed strategies, possible solutions identified through any of the engineering classes can be provided to *Optimize*. Lastly, *Reason* is used to review the possible solutions, and *Conflict Resolution* is used to resolve any identified conflicts from *Analyze*. The output (*Plan Data*) can then be used as input to *Execute*.

5) EXECUTE

This component receives a definitive policy and uses *Knowledge* data where the *Plan Data* is available. Accordingly, *Execute* decomposes the received definitive policy to one or more imperative policies. The imperative policies are often represented as an action type of policy with details for device-specific configurations. These action policies are instantiated on the applicable resources.

Each ECA policy that Execute generates is a single transaction. However, this transaction is in fact a consolidated set of actions. A transaction is successful, only if the complete set of actions are successfully executed. A transaction is unsuccessful, if any action from the set of actions fails, in which case we must revert the changes.

6) MAPPING OF OUR FRAMEWORK FOR AUTONOMIC MANAGEMENT TO THE PBM FRAMEWORK

Our proposal can be mapped to the general PBM framework requirements identified in the PBM literature (e.g. [9], [91], [92]). Figure 7 shows the mapping between the traditional IETF/DMTF PBM framework and our proposal. For each MAPE-K component, we specify in braces the general applicable PBM component: Monitor {event monitoring system, Policy Enforcement Point (PEP)}, Analyze and Plan {Policy Decision Point (PDP), policy analysis, PEP}, Execute {PEP, policy deployment model, enforcement agents} and Knowledge {policy repository (PR), Management Information Base (MIB)}.

The Policy Enforcement Point (PEP) needs to be present in each of the MAPE components, as each enforces policies at different levels of policy abstraction. For example, Execute as a PEP enforces imperative policies, while Monitor and Plan enforce definitive policies, and Analyze enforces declarative and definitive policies. The policy console and policy management tool (PMT) framework requirements correspond to the applications in our proposal.

C. GENERIC INTENT REFINEMENT WORKFLOW

Algorithm 1 describes the generic workflow for an application to establish a MAPE-K loop to refine intents. There can be multiple active applications at a given time. This implies that multiple loops can co-exist with some running more frequently, each having the goal of finding mitigation strategies to reinforce their existing intents. For example, applications could support fast loops, where new intents may not be immediately included in the policies that are enforced at that time. On the other hand, applications can support slow loops that receive new intents within the loop and that try to enforce both existing and new intents with some overall optimization goal in mind. For instance, a service provider may wish to enforce all intents (both existing and new) while maximizing the network throughput in a loop that runs, for example, every eight hours. The inputs to the algorithm typically involve an intent or a list of intents, a policy model (described in section IV), a resource model, and all the available AMS resources (e.g. network, compute, storage). The algorithm results in a set of policies generated as a result of a single intent. The output typically includes actions (i.e. action policies) that are actuated on the resources.

Algorithm 1: Generic Intent Refinement Algorithm

```
input  : {intentList} – list of intents
input  : policyModel – models the policy at various layers of
           abstraction.
input  : resourceModel – network, compute, storage
input  : resources – network, compute, storage
output : {actions}
1 Receive {intentList}. Transform {intentList} to declarativePolicies
    using policyModel. Perform integrity and consistency analysis on
    declarativePolicies;
2 Monitor resources: topology, flow tables, demands, resource utilization
    are read from resources and updated in the resourceModel;
3 Analyze resourceModel to determine if declarativePolicies are
    enforced. If {violatedPolicies} are detected, generate a policy using
    policyModel to pass {violatedPolicies} to 4;
4 Plan resolutions by reasoning and learning over resourceModel,
    PolicyModel, and {violatedPolicies} to find {actions} to mitigate
    {violatedPolicies}. Report {violatedPolicies} & {actions} to user for
    authorization (optional); if authorised, generate a policy using
    policyModel to send possibly modified {actions} to 5;
5 Execute {actions} on resources;
6 Generate policies using policyModel to monitor newly enforced
    policies. Go to 2.
```

In Algorithm 1 first intents are received by the application, where they are transformed into a set of declarative policies (line 1). Integrity and consistency analysis is performed next on declarative policies (as explained in Section IV). Monitoring data is collected at regular intervals from the resources and is used to update the resource model stored in *Knowledge* (line 2). Any of the MAPE-K components can receive real-time data from *Monitor* or use data from *Knowledge*.

The resource model and policy model are analyzed (line 3) to determine if all declarative policies are enforced, and to identify if any policies are violated. *Analyze* decomposes a declarative policy to a set of definitive policies to perform the necessary analysis. Based on the analysis outcome (e.g. violated policies) relevant data is passed to *Plan* in the form of a definitive policy generated using the policy model. *Plan* reasons about different solutions (line 4) in accordance with the definitive policies it receives from *Analyze*. Alternatively, the solutions can be confirmed with the user before they are sent to Execute for execution.

The definitive policies that mandate some actions to be enforced are passed to *Execute* (line 5). Accordingly, *Execute* may generate one or more imperative policies to enforce the necessary actions. Further, in *Execute*, some policies are generated to install new sensors to monitor data related to the new enforced policies. Then the application generates monitoring policies targeting Monitor to monitor the new policies (line 6). Once the algorithm reaches the last step (line 6), it loops back to line 2.

In this section, we addressed part of the objectives we defined in Section I for our Autonomic Management System. In doing so, we have enabled the following:

- support for user intents and their refinement: intelligent and automated intent to policy conversion; intent assurance using control loops.
- three-tiered architecture with separation between applications and platform components, policy abstraction and corresponding API classes.
- support for new services by extending or adding new logics as guidance to the system.

Next, we address the policy model, and the hierarchical policy representation objectives for our AMS.

IV. POLICY MODEL

Information models are required to formalize the policies to be analyzed by the system. There are two main requirements from our autonomic management system to be addressed by a policy model. First, the policy model should facilitate mapping from one layer to another leading to strong synchronization and consistencies among various policies, especially at run-time [50]. Hence, we require a unified policy model that models policies across various layers of abstraction within our management system. Second, the model should be flexible enough to represent various types of policies that can be involved in the refinement of an intent including utility and goal policies, in addition to action policies.

In order to define a policy model that is oblivious to the level of abstraction and policy type, we define policy as an action that is constrained with respect to resource, spatial, and temporal attributes to achieve a desirable state. The actions and constraints can be atomic or non-atomic depending on the level of abstraction. In the non-atomic case, the action and constraints are further decomposed into other actions and constraints at lower abstraction levels.

Hence, based on above definition, we formally define Policy $\overrightarrow{P}$ as:

$$\overrightarrow{P} = (D, E, A, \overrightarrow{C}). \tag{1}$$

where D denotes the policy definer that defines the policy, E is an entity or a group of entities that enforce the policy, and A is an action. $\overrightarrow{C}$ is a vector of constraints that apply to action A in terms of the resources,[2] location, and time, defined as:

$$\overrightarrow{C} = (\overrightarrow{R}, \overrightarrow{T}, \overrightarrow{S}). \tag{2}$$

Here $\overrightarrow{R}$ is a vector of tuples consisting of resources (r_i) and their metrics ($\overrightarrow{m_i}$) defined as:

$$\overrightarrow{R} = ((r_0, \overrightarrow{m_0}), \ldots, (r_K, \overrightarrow{m_K})). \tag{3}$$

We emphasize that in our policy definition, the term **resource** is used to express and generalize any entity that can be consumed by another entity. An entity is something that exists separately from other things and has a distinct identity of its own. For example, entity can be a human, application, software or hardware component, etc. In particular, in this work we consider policies as resources. In this context the term "resource" is applied more broadly and therefore it does not solely refer to the conventional use of the term "resource" (e.g., a network resource such as capacity), which may be limited or scarce. If we refer explicitly to network resources, the term "network resource" will be used.

For each r_i, $\overrightarrow{m_i}$ is a vector of metrics defined as:

$$\overrightarrow{m_i} = (m_{i0}, \ldots, m_{iL}). \tag{4}$$

We define legal operations for each metric, m_{ij}, per each resource r_i. For example for the resource *Link*, we define following operations for the metric *Utilization*: {max, min, sum, subtract, percentile, average, $>$, $<$, $=$, $\neq$}.

Resource r_i can take any of the following forms:

- Simple: there are a number of metrics associated with r_i where m_{ij} is not a resource itself, for $0 \leq j \leq L$.
- Nested: Resource r_i includes another resource r_j as one of its metrics and this inclusion can go on recursively, for $0 \leq j \leq L$. The state of the outer resource is dependent on the state of the inner resource. For instance, interface I, which itself is a resource, is modeled as an ingress or egress interface of Link L. If the metric *State* of I is *down*, so is the state of L.
- Compound: Resource r_i is composed of multiple resources where the relationship between the resources is defined by the legal operations of their metrics. For instance, r_i can be defined as the average link utilization of links L_1 and L_2.

$\overrightarrow{T}$ is a vector of temporal attributes related to time such as date, interval, duration etc.:

$$\overrightarrow{T} = (\overrightarrow{t_0}, \ldots, \overrightarrow{t_K}). \tag{5}$$

For each r_i, $\overrightarrow{t_i}$ is a vector of temporal attributes defined as:

$$\overrightarrow{t_i} = (t_{i0}, \ldots, t_{iM}). \tag{6}$$

$\overrightarrow{S}$ is a vector of spatial attributes related to location such as IP address, ID, geographical address, etc.:

$$\overrightarrow{S} = (\overrightarrow{s_0}, \ldots, \overrightarrow{s_K}) \tag{7}$$

[2]Throughout the rest of this paper the term "resource" will be used in the context of the proposed policy model.

For each r_i, $\overrightarrow{s_i}$ is a vector of spatial attributes defined as:

$$\overrightarrow{s_i} = (s_{i0}, \ldots, s_{iN}). \tag{8}$$

for $\forall i, j, K, L, M, N \in \mathbb{N}$ in equations 3−8.

A policy $\overrightarrow{P}$ may be composed of multiple policies, denoted by $\overrightarrow{P} = \overrightarrow{P_1} \odot \overrightarrow{P_2} \cdots \odot \overrightarrow{P_S}$ where by definition $\overrightarrow{P}$ is enforced if $\overrightarrow{P_j}$ is enforced, for $1 \leq j \leq S$, $S \in \mathbb{N}$. The order of composition defines the proper order of enforcement. For instance, for $\overrightarrow{P}$ to be enforced, $\overrightarrow{P_1}$ needs to be enforced before $\overrightarrow{P_2}$ and so on. We define policy $\overrightarrow{P_1}$ nested if $\overrightarrow{P_1}$ is constrained on another policy, $\overrightarrow{P_2}$ (i.e, $\overrightarrow{P_2}$ as a resource constraint to $\overrightarrow{P_1}$); this means that $\overrightarrow{P_1}$ will be enforced if $\overrightarrow{P_2}$ has been already enforced.

For policy $\overrightarrow{P}$, we define policy meta-data as:

$$\overrightarrow{PMD} = (ID, DM, EX, PR, AP) \tag{9}$$

where ID is the policy identifier, DM defines the domain where $\overrightarrow{P}$ is enforced, EX is the expiration date of $\overrightarrow{P}$, PR defines the priority of $\overrightarrow{P}$, and AP is the autonomic permission that determines if the policy definer wishes to confirm the actionable policies to enforce $\overrightarrow{P}$.

Modeling Event: Next we explain how the proposed model captures events. Events can be modeled as composite data that is composed based on evaluation of one or more conditions or as primitive data [10].

Definition 1: Let ev be an event that happens if Condition c is satisfied.

Strictly speaking, in order to check if an event has happened, one needs to perform an action (i.e., *analyze*) to see if a condition is true. Hence, we model the event ev with Policy $\overrightarrow{P}$, action *analyze*, and the constraint c as defined in Equation 2. Assume we are interested to model the event *linkfailure* of Link l with State s. We define Policy $\overrightarrow{P}$ as:

$$\overrightarrow{P} = (d, e, analyze, (((l, (s))))) \tag{10}$$

following Equation 1 where d is a policy definer, e is the entity that enforces the policy (i.e, an analyzer), *analyze* is the action and l is a resource with the metric s. The temporal attribute is not given which is implicitly defined as *always* and there is no spatial attribute. So Event ev happens if $\overrightarrow{P}$ is enforced (i.e, if the result of analysis is true.)

If the event is received as primitive data, then we model the event as a resource (r_i) in the policy definition. The metrics of the event can include event type, event location and event time. For example, link failure of Link l is modeled as resource r_1 with metric ($\overrightarrow{m_1}$) given in Equation 11 where *linkFailure* is the type of the event, l is the location where the event happens and t is the time of the event.

$$\overrightarrow{m_1} = (ev, (linkFailure, l, t)) \tag{11}$$

In general we model an ECA policy $\overrightarrow{P}$ by evaluating events and conditions following Equation 10 and the resulting policies are passed as nested policies to $\overrightarrow{P}$. For example, consider an ECA policy defined as P_1 = *"on congestion, if packet drop > 5%, load-balance"*. To evaluate the event we use policy $\overrightarrow{P_2} = (d, e, analyze, (((l, (utilization > 85\%)))))$, where utilization of link $l > 85\%$ is what is defined as congestion. To evaluate the condition we use policy $\overrightarrow{P_3} = (d, analyze, (((l, (packet\_loss > 5\%)))))$. Then, $\overrightarrow{P_1}$ can be represented as a nested policy that is constrained on policies $\overrightarrow{P_2}$ and $\overrightarrow{P_3}$:

$$\overrightarrow{P_1} = (d, e, load - balance, ((\overrightarrow{P_2}, \overrightarrow{P_3}))) \tag{12}$$

In Equation 12, d is the policy definer, e is the enforcer, load-balance is the action, and the constraint includes policies $\overrightarrow{P_2}$ and $\overrightarrow{P_3}$. Unlike existing models that model only ECA policies which limits their applicability to support future management scenarios, our model represents non-ECA policies as well. We provide an example in Section V, to show how we model a utility policy. Our unified model ensures the consistency of the policies deployed across the system and provides the foundation for powerful policy analysis processes as described next.

A. INTEGRITY AND CONSISTENCY ANALYSIS (ICA)

In the proposed autonomic management system, Integrity and Consistency Analysis (ICA) is performed to ensure data quality of policies. Initial ICA is performed when intents are being transformed to declarative policies. First, the intents are formalized and checked against the corresponding data models to see if any value is missing or wrongly entered using the sanity check. This ensures the integrity of policies. Then the new declarative policies are analyzed to ensure they are consistent with the existing ones.

In a shared environment such as a network, policies are correlated. There may be overlaps between policies where a policy is already covered by another policy or one policy is in conflict with other policies either fully or partially. Given our formal definition of policy, we define following relationships between policies $\overrightarrow{P_i}$ and $\overrightarrow{P_j}$ to detect inconsistencies:

1) $\overrightarrow{P_i} \subseteq \overrightarrow{P_j}$: $\overrightarrow{P_i}$ is a subset of $\overrightarrow{P_j}$ if both specify the same action, and constraints of $\overrightarrow{P_i}$ is a subset of constraints of $\overrightarrow{P_j}$, i.e., $A_i = A_j$, $\overrightarrow{R_i} \subseteq \overrightarrow{R_j}$, $\overrightarrow{T_i} \subseteq \overrightarrow{T_j}$ and $\overrightarrow{S_i} \subseteq \overrightarrow{S_j}$, where $\overrightarrow{R_i} \subseteq \overrightarrow{R_j}$ means $\forall r_u \in \overrightarrow{R_i}, r_u \in \overrightarrow{R_j}$, $\overrightarrow{T_i} \subseteq \overrightarrow{T_j}$ means $\forall t_u \in \overrightarrow{T_i}, t_u \in \overrightarrow{T_j}$ and $\overrightarrow{S_i} \subseteq \overrightarrow{S_j}$ means $\forall s_u \in \overrightarrow{S_i}, s_u \in \overrightarrow{S_j}$.
2) $\overrightarrow{P_i} \perp \overrightarrow{P_j}$: $\overrightarrow{P_i}$ is in conflict with $\overrightarrow{P_j}$ if they cannot be enforced for the same set of constraints. In this case, their actions are orthogonal (i.e., they cannot happen at the same time) and the constraints of one policy is a subset of another policy's constraints. In other words, $A_i \perp A_j$, $\overrightarrow{R_i} \subseteq \overrightarrow{R_j}$, $\overrightarrow{T_i} \subseteq \overrightarrow{T_j}$, $\overrightarrow{S_i} \subseteq \overrightarrow{S_j}$ where $A_i \perp A_j$ means action A_i is orthogonal to action A_j.

Based on above definitions, in consistency analysis, we check for the following cases:

- **Redundancy**: Policy $\overrightarrow{P_i}$ is redundant to policy $\overrightarrow{P_j}$, if $\overrightarrow{P_i} \subseteq \overrightarrow{P_j}$. For instance, the declarative policy corresponding to intent 1 is redundant to that of intent 2 in the following example where bigger priority number means higher priority:

1) Allow http traffic between host pair (A, B) using priority 100.
2) Allow http and FTP traffic in the network using priority 200.

- **Partial Redundancy**: Policy $\overrightarrow{P_i}$ is partially redundant to policy $\overrightarrow{P_j}$, if the actions are the same for the same *subset* of constraints, i.e., $A_i = A_j$, $\overrightarrow{R_i} \subseteq \overrightarrow{R_j}$, $\overrightarrow{T_i} \subseteq \overrightarrow{T_j}$ and $\overrightarrow{S_i} \cap \overrightarrow{S_j} \neq \emptyset$, where $\overrightarrow{S_i} \cap \overrightarrow{S_j} \neq \emptyset$ means $\exists s_u \in \overrightarrow{S_i}$ and $\overrightarrow{S_j}$. For example, the declarative policies corresponding to the following intents are partially redundant:
  1) Allow http traffic between host pairs (A, B) and (C, D) using priority 100.
  2) Allow http and FTP traffic between host pairs (A, B) and (E, F) using priority 200.
- **Conflict**: conflict occurs if $\overrightarrow{P_i} \perp \overrightarrow{P_j}$. For example, the declarative policies corresponding to the following intents are in conflict with each other.
  1) Allow http traffic between host pairs (A, B) using priority 100.
  2) Drop http and FTP traffic in the network using priority 200.
- **Partial Conflict**: Policy $\overrightarrow{P_i}$ is partially in conflict with policy $\overrightarrow{P_j}$, if their actions are orthogonal for a number of common constraint attributes, i.e, $A_i \perp A_j$, $(\overrightarrow{R_i} \cup \overrightarrow{R_j}) > (\overrightarrow{R_i} \cap \overrightarrow{R_j}) \neq \emptyset$, $(\overrightarrow{T_i} \cup \overrightarrow{T_j}) > (\overrightarrow{T_i} \cap \overrightarrow{T_j}) \neq \emptyset$, $(\overrightarrow{S_i} \cup \overrightarrow{S_j}) > (\overrightarrow{S_i} \cap \overrightarrow{S_j}) \neq \emptyset$ where $(\overrightarrow{R_i} \cup \overrightarrow{R_j})$ means (r_i, r_j), $r_i \in R_i$ and $r_j \in R_j$. For example, the declarative policies corresponding to the following intents are partially in conflict with each other:
  1) Allow http traffic between host pairs (A, B), (C, D) using priority 100.
  2) Drop http and FTP traffic between host pairs (A, B), (E, F) using priority 200.

We resolve the above situations with the following resolutions:

- **Redundancy**: if Policy $\overrightarrow{P_i}$ is redundant to policy $\overrightarrow{P_j}$, only $\overrightarrow{P_j}$ will be enforced.
- **Partial Redundancy**: if Policy $\overrightarrow{P_i}$ is partially redundant to policy $\overrightarrow{P_j}$, we derive a new policy, $\overrightarrow{P_k}$, that includes all the spatial constraints within both policies, where $\overrightarrow{R_k} = \overrightarrow{R_j}$, $\overrightarrow{T_k} = \overrightarrow{T_j}$, $\overrightarrow{S_k} = \overrightarrow{S_i} \cup \overrightarrow{S_j}$ and enforce $\overrightarrow{P_k}$.
- **Conflict**: if Policy $\overrightarrow{P_i}$ is in conflict with policy $\overrightarrow{P_j}$, constraints of $\overrightarrow{P_i}$ are subset of constraints of $\overrightarrow{P_j}$, and priority of $\overrightarrow{P_i}$ is lower than $\overrightarrow{P_j}$, then $\overrightarrow{P_j}$ will be enforced. Otherwise, we break $\overrightarrow{P_j}$ into two policies where in one policy, we modify the spatial attributes of $\overrightarrow{P_j}$ to exclude the common spatial attributes with $\overrightarrow{P_i}$. In another policy, we enforce $\overrightarrow{P_j}$ for the common spatial attributes at times excluding the temporal attributes of $\overrightarrow{P_i}$. So $\overrightarrow{P_j}$ will be updated as $\overrightarrow{P_j} = (A_j, (\overrightarrow{R_j}), (\overrightarrow{T_j}), (\overrightarrow{S_j} - \overrightarrow{S_i})) \odot (A_j, (\overrightarrow{R_j}), (\overrightarrow{T_j} - \overrightarrow{T_i}), (\overrightarrow{S_i}))$ and will be enforced along with $\overrightarrow{P_i}$.
- **Partial Conflict**: if policy $\overrightarrow{P_i}$ is partially in conflict with policy P_j and priority of $\overrightarrow{P_i}$ is less than $\overrightarrow{P_j}$, we update $\overrightarrow{P_i}$ as $\overrightarrow{P_i} = (A_i, R_i - (\overrightarrow{R_i} \cap \overrightarrow{R_j}), T_i - (\overrightarrow{T_i} \cap \overrightarrow{T_j}), S_i - (\overrightarrow{S_i} \cap \overrightarrow{S_j}))$ and enforce $\overrightarrow{P_i}$ and $\overrightarrow{P_j}$.

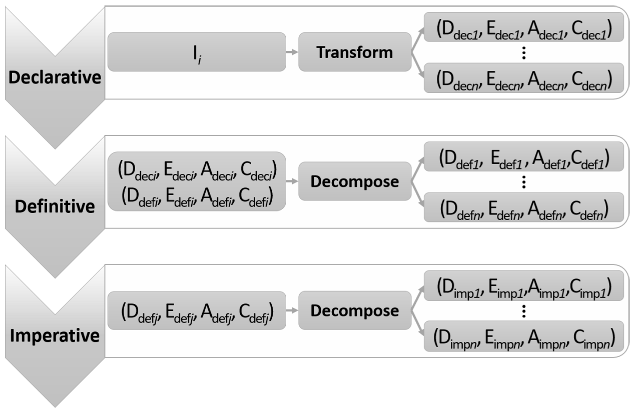

Declarative
I_i
Transform
$(D_{dec1}, E_{dec1}, A_{dec1}, C_{dec1})$
⋮
$(D_{decn}, E_{decn}, A_{decn}, C_{decn})$
Definitive
$(D_{deci}, E_{deci}, A_{deci}, C_{deci})$
$(D_{defi}, E_{defi}, A_{defi}, C_{defi})$
Decompose
$(D_{def1}, E_{def1}, A_{def1}, C_{def1})$
⋮
$(D_{defn}, E_{defn}, A_{defn}, C_{defn})$
Imperative
$(D_{defj}, E_{defj}, A_{defj}, C_{defj})$
Decompose
$(D_{imp1}, E_{imp1}, A_{imp1}, C_{imp1})$
⋮
$(D_{impn}, E_{impn}, A_{impn}, C_{impn})$

FIGURE 8. Generated policies from the refinement of intent using the three API classes.

B. POLICY TREE

Figure 8 depicts how an intent, I_i, is transformed and decomposed to multiple policies across the API classes. First the intent is transformed to one or more declarative policies formalized by Equation 1. Then each declarative policy is further decomposed to multiple definitive policies. Definitive policies may decompose to other definitive policies or to imperative policies, data models of which follow Equation 1. Since we utilize a unified policy model across abstraction layers, the mapping between policy abstraction types is performed easily which ensures consistency and synchronization among policies at various layers of abstraction. Also, the unified policy model greatly helps to manage interactions and contentions among policies of various roles such as QoS, security, fault management etc.

Based on the processing model in Figure 8, we define Policy Tree (PT) as a tree that includes the policies generated from the refinement of an intent. The root of the tree is the intent and the *pre-order traversal (root-left-middle-right)* of the tree shows the sequence in which policies are enforced. The sequence of policy enforcement follows the generic workflow presented in Section III-C.

In PT, starting from an intent, a declarative policy will be generated. It is worth mentioning that in our data model for declarative policies, the actions include high level actions within network and service management. For example, *Connect* is one of the high level actions at the declarative layer that represents connectivity, characteristics of which (i.e, QoS, type, etc.) are defined using the constraint attributes. Other examples of high level actions used in declarative policies are *Maximize* and *Minimize*. These are usually used to form utility policies that optimize network

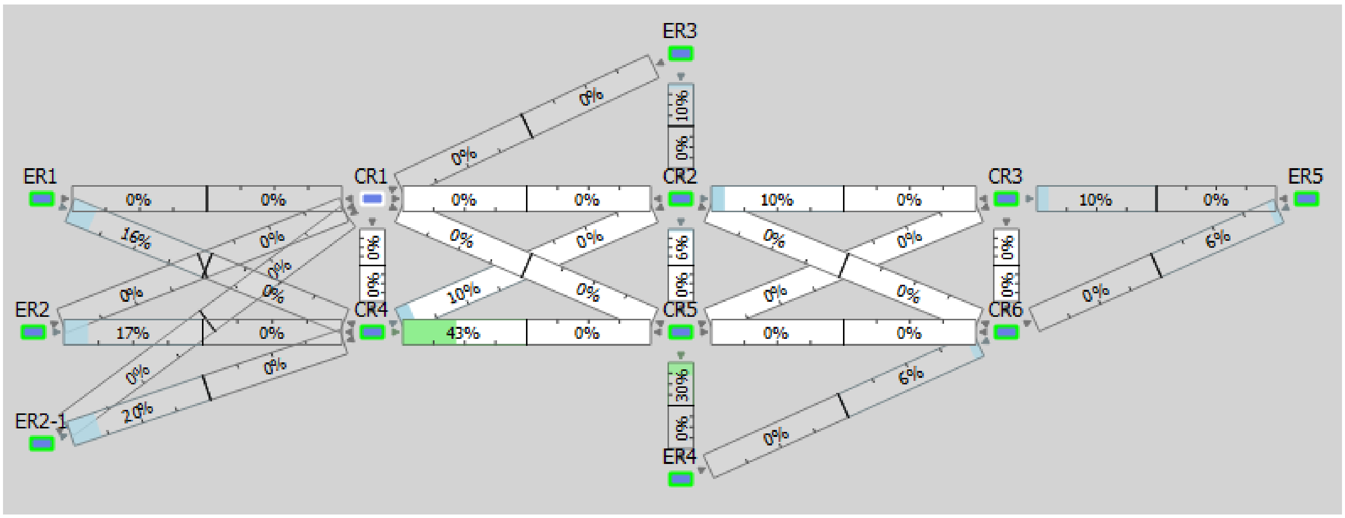


FIGURE 9. Initial link utilization state. The use-case topology includes a core network with connections to edge nodes. Core nodes are denoted by CR_i ($1 \leq i \leq 6$), and edge nodes are denoted by ER_j ($1 \leq j \leq 5$) and ER_{2-1}. Each link shows the utilization percentage (ingress and egress), computed based on traffic and link capacity. The link utilization results from 6 demands that are passing through the network, the details of which are given in Table 4.

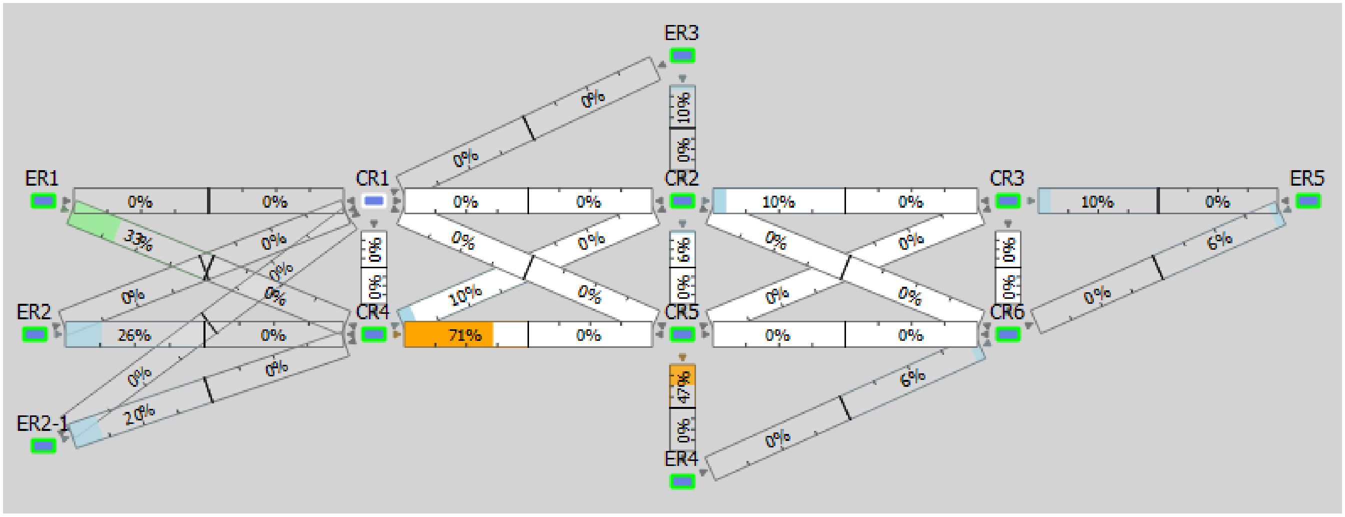


FIGURE 10. Post increase link utilization state. After demand traffic increases (presented in Table 4), two violations are detected for the intent I_7: 1) core link utilization violation ($CR_4 - CR_5 > 70\%$) and 2) edge link utilization violation ($CR_5 - ER_4 > 45\%$).

resources characterized by a set of constraints (i.e., what, where, and when to optimize.)

In the definitive layer, the declarative policy is further decomposed to more definitive policies. More specifically, the declarative policy will generate a main definitive policy targeting *Monitor* which then branches out to more definitive policies for *Monitor*. Similarly, the declarative policy will generate a main definitive policy targeting *Analyze* which then branches out to more definitive policies for *Analyze*. Based on the results from *Analyze* branch, a definitive policy will target *Plan*. This main definitive policy from *Plan* branches out to more definitive policies for *Plan*, and the same goes for *Execute*. Consequently, the definitive policies targeting *Execute* are then decomposed to imperative policies that are enforced on the resources. In the next section, we provide a use-case to demonstrate the intent refinement and the resulting policy tree.

V. USE-CASE

In this section, we present a proof of concept for intent refinement. We have implemented our proposal to demonstrate the policies and resulting policy tree, that are generated with the refinement of an intent. The use-case involves a Service Provider network, but other use-cases can be considered as well (e.g. enterprise networks). We proceed with an overview of the use-case organization and environment, followed by a detailed description of the intent refinement, that goes through the policy generation process, and ends with the results of the intent deployment in a simulated network environment. We conclude the section with a brief discussion on system generalization and performance considerations.

In section III-C, we presented a generic intent refinement workflow. Both the applications and the platform can have workflows of their own to provide specific functionalities. Each workflow essentially represents a set of sequenced jobs that generate and enforce policies following some decision process. In this use-case, we consider an application (App_1) for refinement of new intents. The application contains the logic of the MAPE (Algorithm 2) and, when required, the application invokes the M, A, P, and E from the platform. To do this, the application MAPE components make API calls to the platform, and use the functionalities available from the platform MAPE components (Algorithms 3-6). For example, the application Monitor component makes an API call to the platform Monitor workflow. This workflow returns the results back to the application for review and to determine the next job. We note that each of the presented platform workflows are internal to a specific MAPE component. Please note that for clarity, policies related to each MAPE-K component are given a prefix with the name of the component (e.g., *analyzePolicyStatus*, *analyzeValidate* etc.)

Environment and Setup: We use the Cisco WAN Automation Engine (WAE) Design simulator [93] to create a network topology for the use-case. The topology (shown in Figure 9) was extracted from a portion of a real service provider backbone network, where aggregated (elephant) traffic flows are passed through a network of six core and six edge nodes. The topology, as well as the network traffic, were generated in Cisco WAE Design. In WAE Design, a traffic flow is defined by a traffic demand that contains at least a source and destination pair, and traffic volume. We defined six demands, to generate the network traffic for the use-case, as well as a set of Segment Routing Label-Switched Paths (SR LSPs) to route the traffic related to the demands. WAE allows for adjustments to the traffic volume in order to simulate different traffic conditions, and investigate possible routing scenarios (that could be computed for example, using available, built-in WAE network optimizations). In our use-case, we show the refinement of a network optimization intent with a focus on the policy generation part. As for computing the optimization, we use one of the built-in WAE optimization functions. Last, from WAE, the topology and network metrics are also made available to the workflows. For example, link utilization is one of the metrics that we use, and WAE calculates this metric by using two other metrics: link capacity and traffic volume per link.

For the use-case, we consider three states in terms of link utilization: initial (Figure 9), post traffic increase (Figure 10), and post optimization (Figure 12). In Table 3, we present link capacity and link utilization, and in Table 4, we present details for the six demands.

Use-case: We rely on MPLS-based Segment Routing (SR) to route traffic via SR LSPs. In SR, a Segment Identifier (SID) is used to identify a segment. In MPLS-based SR, each segment is encoded as an MPLS label, and an ordered list of segments is encoded as a stack of labels. Similar to MPLS operations, the segment on top of the stack is processed and the related label is popped from the stack [94]. There are different types of SIDs that serve different purposes (e.g. prefix-SID, node-SID, adj-SID, anycast-SID, binding-SID). We use adjacency segment identifiers (adj-SID) to create the

TABLE 3. Link capacity and utilization for pertinent links.

Links and Capacities		Link Utilization (%)		
Links	Capacity (Gb/s)	Initial	Post Increase	Post Optimization
$ER_1 - CR_4$	30	16	33	33
$ER_2 - CR_4$	40	17	26	26
$CR_4 - CR_5$	30	43	**71**	**46**
$CR_4 - CR_2$	30	10	10	35
$CR_2 - CR_5$	30	6.9	6.9	19
$CR_5 - ER_4$	50	30	**47**	**39**
$CR_2 - CR_6$	40	0	0	12
$CR_6 - ER_4$	40	6	6	15

Algorithm 2: Application Logic (Refine Workflow)

```
inputs  : intent_list=[$I_7$], D: definer (D = app$_1$)
outputs : $pt$: policy tree, $P_d$: declarative policy, ($M_1$,$A_1$,$Pl_1$,$E_1$):
          MAPE platform instances, MAPE$_{out}$ = ($o_M$,$o_A$,$o_P$,$o_E$):
          outcome from Monitor, Analyze, Plan, Execute
for each intent ∈ intent_list do
    if existing_intent(intent) then
        outcome ← validate_intent(intent)
    else
        outcome ← refine_intent(intent)
    // Update knowledge with outcome
def refine_intent(intent):
    // Create Policy Tree for each new intent:
    pt = nx.DiGraph()
    // Generate & Enforce $P_{7-0}$ to transform intent to $P_7$:
    $T_1$ = Transform('intent', intent, pt)
    policy, pt ← generate(D, $T_1$, 'analyzeTransform', intent, pt)
    $P_d$ ← policy['Action'](policy)
    // Generate & Enforce $P_{7-1}$ to get monitoring data for $P_7$:
    $M_1$ = Monitor('declarative', $P_d$, pt)
    policy, pt ← generate(D, $M_1$, 'monitor', $P_d$, pt)
    $o_M$ ← policy['Action'](policy)
    // Generate & Enforce $P_{7-3}$ to analyze $P_7$:
    $A_1$ = Analyze('declarative', $P_d$, pt)
    policy, pt ← generate(D, $A_1$, 'analyze', $P_d$, pt)
    $o_A$ = policy['Action'](policy)
    if $o_A$ then
        return (True, $o_A$, pt)
    else
        // Generate & Enforce $P_{7-6}$ to plan $P_7$:
        $Pl_1$ = Plan('declarative', $P_d$, pt)
        policy, pt ← generate(D, $Pl_1$, 'plan', $P_d$, pt)
        $o_P$ ← policy['Action'](policy)
        if $o_P$ then
            // Generate & Enforce $P_{7-9}$ to deploy intent:
            $E_1$ = Execute('declarative', $P_d$, pt)
            policy, pt ← generate(D, $E_1$, 'execute', $P_d$, pt)
            $o_E$ ← policy['Action'](policy)
            if $o_E$ then
                return (True, MAPE$_{out}$, pt)
            else
                return ($o_E$)
            end if
        else
            return ($o_P$)
        end if
    end if
```

SR LSPs. An adj-SID is defined per node interface as a local label that points to a specific interface. An SR LSP may have

TABLE 4. Demand details and LSP information. Demands *D*2 and *D*3 have traffic increase. The active LSP Paths are shown for each demand before the refinement of the intent I_7. The refinement of I_7 resolves the network violations through a path change for demand *D*3.

Demand and LSP Information:					Initial and Increased traffic with initial LSP Paths:			Post Refinement:
Demand	Tag	Source	Destination	LSP Name	Initial Traffic	Increased Traffic	Active LSP Paths	Active LSP Paths
D1	DS	ER_3	ER_4	$ER_3 : ER_4$	2 Gbps	2 Gbps	$[ER_3 - CR_2 - CR_5 - ER_4]$	no change
D2	**DS**	$\mathbf{ER_1}$	$\mathbf{ER_4}$	$\mathbf{ER_1 : ER_4}$	**5 Gbps**	**10 Gbps**	$\mathbf{[ER_1 - CR_4 - CR_5 - ER_4]}$	**no change**
D3	**DT**	$\mathbf{ER_2}$	$\mathbf{ER_4}$	$\mathbf{ER_2 : ER_4}$	**4 Gbps**	**7.5 Gbps**	$\mathbf{[ER_2 - CR_4 - CR_5 - ER_4]}$	**path change**
D4	DT	ER_{2-1}	ER_4	$ER_{2-1} : ER_4$	4 Gbps	4 Gbps	$[ER_{2-1} - CR_4 - CR_5 - ER_4]$	no change
D5	DT	ER_2	ER_5	$ER_2 : ER_5$	3 Gbps	3 Gbps	$[ER_2 - CR_4 - CR_2 - CR_3 - ER_5]$	no change
D6	DT	ER_5	ER_4	$ER_5 : ER_4$	2.5 Gbps	2.5 Gbps	$[ER_5 - CR_6 - ER_4]$	no change

several path options, among which the highest priority and active one is used.

I_7 is transformed to a declarative policy P_7 using a definitive policy P_{7-0} that is generated and enforced in Algorithm 2, lines 11-13. The definitive action for P_{7-0} (*analyzeTransform*) executes the transform platform workflow (Algorithm 3) from the *Analyze* platform component. The workflow uses NLP-related methods and vocabularies[3] to translate the intent and generate policy P_7. Vocabulary samples are given in Algorithm 3, lines 15-18.

In our use-case, we assume there are 6 existing intents that specify some application supported services. These services result in demands passing through the network via SR LSPs. Details for the demands (such as delay sensitive (DS) or delay tolerant (DT)) and the LSPs are provided in Table 4. We suppose that a user ($Admin_1$) submits a new intent (I_7) to App_1 defined as: *"Minimizing maximum link utilization subject to EDGE links having upstream bound of 45% and CORE links upper bound of 70% utilization"*. During the refinement of I_7, we consider the resulting policies (P_1-P_6) from the 6 existing intents. The initial network traffic resulting from the demands does not violate I_7, so we purposefully increase traffic for two demands ($D2$, $D3$) to demonstrate the response to the violations as part of the intent refinement. The data models are expressed in Table 5. The PT is shown in Figure 11, where the policies in white nodes are enforced at the application level and the grey ones are enforced at the platform level. We will explain the PT by traversing it in a pre-order manner that demonstrates the order of policy enforcement.

The transform workflow first pre-processes the intent through tokenization, lowercase conversion, punctuation and stop words removal (lines 6-9). Item I from Table 6 shows the output after these methods are applied. Next, the transformation continues using regex,[4] lemmatization, and matching word tokens with pre-defined lookup vocabularies (lines 10 and 11). Item II from Table 6 shows the output after these methods are applied. The declarative policy P_7 is generated by mapping the output to our policy model in line 12. P_7 (shown in Listing 1) models the intent through

[3]We note that application owners should ensure adequate vocabularies exist in *Knowledge*, or add vocabularies as needed.

[4]Regular expression.

Algorithm 3: Platform Workflow (Transform)

```
   inputs   : intent, pt, vocabularies
   outputs  : $P_d$
1  Subclass Transform():
2      __init__(self, type_input, input, pt, **kwargs)
3      def analyzeTransform(self, policy):
4          if self.type_input = 'intent' then
5              nlp = NLP()
6              tokenized ← nlp.tokenize(self.input)
7              lowercase ← nlp.lowercase(tokenized)
8              no_punct ← nlp.punctuation(lowercase)
9              no_stop ← nlp.stopwords(no_punct)
10             processed ← nlp.regex(no_stop)
11             formalized ← nlp.lemmatize_and_match(processed)
12             $P_d$, pt ← generate(D, E, A, C, pt)
13             return ($P_d$)
14         end if
15 // Vocabulary samples:
16 lookup = {'minimum':min, 'maximum':max}
17 regex = [(r'(subject) ([a-zA-z]+) ([a-zA-z]+) (upstream bound)
       ([0-9]+)', '\g<2> \g<3> < \g<5>'), (r'(upper bound) ([0-9]+)', '<
       \g<2>')]
18 policy_lookup = {'Action': ['minimize', 'maximize'], 'Resource':
       ['link', 'node'], 'Metric': ['edge', 'core'], 'Utilization':['<': le]}
```

a main policy that is constrained on a second policy. When a policy is constrained on other policies, the policy can be modeled as a nested policy. To model P_7 as a nested policy we take the edge link utilization as our main policy, and the core link utilization as our second policy (a constraint to the main policy). In P_7 the policy definer is $Admin_1$, the enforcer is App_1, and the declarative action is *Minimize* constrained on two resources, Link and the second policy.

During the refinement P_7 is decomposed into more policies starting from a definitive policy P_{7-1} defined by App_1 where the definitive action is *monitor*, so that a Monitor (M_1) provides the monitoring data required for P_7. When P_{7-1} is enforced, the output is returned by a platform workflow for monitor (given in Algorithm 4). To generate the output, the workflow first extracts the resources from the declarative policy (in *monitor* method, line 5), and then determines the required monitoring data for each resource by categorizing based on the declarative policy attributes and values (lines 6-10). In our case, the resources are links, the declarative action is minimize (i.e. a utility one), and the utilization metrics are not null, thus the category for

TABLE 5. Policy data models for refining the intent I_7.

Notation	Definition
I_7	"Minimizing maximum link utilization subject to EDGE links having upstream bound of 45% and CORE links upper bound of 70% utilization"
P_{7-0}	(App_1, Transform T_1, analyzeTransform, ((I_7)))
P_7	($Admin_1$, App_1, Minimize, (((Link(Edge, (max) utilization < 45)), ($Admin_1$, App_1, Minimize, ((Link(Core, (max) utilization < 70)))))))
P_{7-1}	(App_1, Monitor M_1, monitor, ((P_7)))
P_{7-2}	(M_1, $M_{1.1}$, monitorData, (((Link,(Core, (max) utilization < 70)), (Link,(Edge, (max) utilization < 45)))))
P_{7-3}	(App_1, Analyzer A_1, analyze, ((P_7)))
P_{7-4}	(A_1, $A_{1.1}$, analyzePolicyStatus, ((P_7)))
P_{7-5}	($A_{1.1}$, $A_{1.2}$, analyzeValidate, (((Link,(Core, (max) utilization < 70)), (Link,(Edge, (max) utilization < 45)))))
P_{7-6}	(App_1, Planner Pl_1, plan, ((P_7)))
P_{7-7}	(Pl_1, $Pl_{1.1}$, planOptimize, (((Link,(Core, (max) utilization < 70)), P_1, P_2, P_3, P_4, P_5, P_6)))
P_{7-8}	(Pl_1, $Pl_{1.2}$, planOptimize, (((Link,(Edge, (max) utilization < 45)), P_1, P_2, P_3, P_4, P_5, P_6)))
P_{7-9}	(App_1, Executer E_1, execute, ((P_7)))
P_{7-10}	(E_1, $E_{1.1}$, executeConfigureLSP, (($ER_2 : ER_4$ ($ER_2 - CR_4 - CR_2 - CR_5 - ER_4$, (0.5, D_3, (DT)), $ER_2 - CR_4 - CR_2 - CR_6 - ER_4$, (0.5, D_3, (DT)))), ($ER_{2_{IP}}$)))

TABLE 6. Sample outputs from the intent to declarative policy transformation process.

I	[‘minimizing’, ‘maximum’, ‘link’, ‘utilization’, ‘subject’, ‘edge’, ‘links’, ‘upstream’, ‘bound’, ‘45’, ‘and’, ‘core’, ‘links’, ‘upper’, ‘bound’, ‘70’, ‘utilization’]
II	minimize max link utilization edge link < 45 and core link < 70 utilization

```
{
 "Definer": Admin_1,
 "Enforcer": App_1,
 "Action": "Minimize",
 "Constraint": {
     "Resource": [
         ("link", {
             "resource_type": "network",
             "type": "edge",
             "util": [max, le, 45]
         }),
         ({"Definer": Admin_1,
           "Enforcer": App_1,
           "Action": "Minimize",
           "Constraint": {
               "Resource": [
                   ("link", {
                       "resource_type": "network",
                       "type": "core",
                       "util": [max, le, 70]
                   })
               ]
           }
          }, {
             "resource_type": "policy"
         })
     ]
 }
}
```

Listing 1. Declarative Policy (P_7).

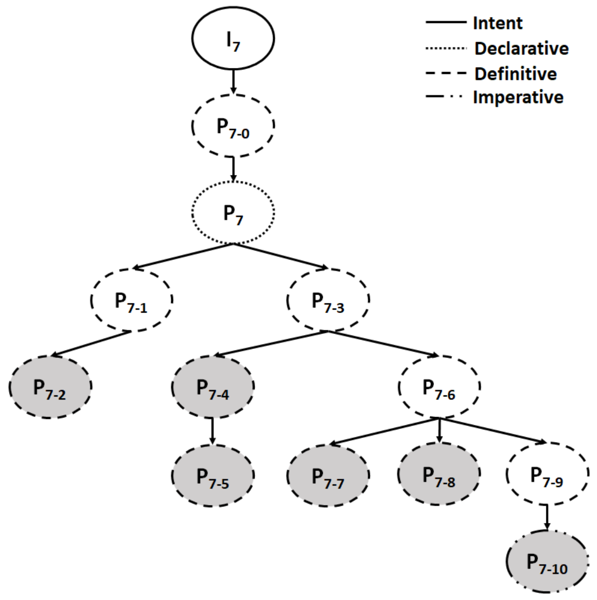


FIGURE 11. Policy tree for the intent: “Minimizing maximum link utilization subject to EDGE links having upstream bound of 45% and CORE links upper bound of 70% utilization”.

monitoring data is link utilization. P_{7-1} is decomposed to another policy P_{7-2} that is defined by M_1 and is to be applied by $M_{1.1}$, a child of Monitor M_1. Before generating P_{7-2} we verify if such a policy exists within the PT,[5] and if not we generate P_{7-2} with the definitive action as *monitorData* (lines 11-17). When P_{7-2} is enforced in line 19, the *monitorData* method (line 23) is called. This method uses the populated link utilization category and iterates over each pair of resource and metric to collect the monitoring data (lines 25-31). In our use-case, each resource is a link with a metric type that is either core or edge, hence the output is a key-value data structure separated by metric type, where each key is a link name and the value is the link utilization. This result is available to the rest of the workflows.

P_7 is further decomposed into policies for analysis: App_1 defines P_{7-3} so that an Analyzer (A_1) analyzes P_7. When P_{7-3} is enforced, the platform workflow for analysis is invoked (Algorithm 5), starting with the *analyze* method given in line 3. As there is no specific analyze workflow provided as input to this method, a default workflow is used. The workflow begins by decomposing policy P_{7-3} to another policy P_{7-4} that is defined by A_1 and is to be applied by $A_{1.1}$, a child of Analyzer A_1. P_{7-4} is generated in line 13 where the definitive action is *analyzePolicyStatus* which is a non-atomic

[5]In general, in the workflows (platform or application) before we generate a new policy, we check the policy tree to ensure no redundant policies are generated and added to the tree.

Algorithm 4: Platform Workflow (Monitor)

```
    inputs  : $P_d$, $pt$, $policy$, $U$: utility lookup e.g. $U$=[minimize]
    outputs : $link\_util$: category for link utilization, $mon\_data$:
              monitoring data, $R$: policy resource = $(r, m)$
 1  Class Monitor():
 2      __init__(self, type_input, input, pt, **kwargs)
 3      def monitor(self, policy):
 4          self.mon_data=[]; self.link_util=[]
 5          self.R ← extract_resource(policy)
 6          for each pair $(r, m) \in$ self.R do
 7              if $r$ = link and Action $\in U$ and util $\neq$ None then
 8                  self.link_util.append((r, m))
 9              end if
10          end for
11          if policy_exist('monitorData') then
12              // if the policy exists, $p_{nb}$ is the policy number
13              policy ← self.pt.nodes[$p_{nb}$]
14          else
15              // Generate & Enforce $P_{7-2}$ to get monitoring data:
16              E = Data('policy', policy, self.pt)
17              policy, self.pt ← generate(D, E, A, C, pt)
18          end if
19          result ← policy['Action'](policy)
20          return (True, result)
21  Class Data():
22      __init__(self, type_input, input, pt, **kwargs)
23      def monitorData(self, policy):
24          self.data={}; self.mon_data=[];
25          if link_util then
26              for each pair $(r, m) \in$ link_util Each do
27                  data[r] ← self.get_link_util(r, m[type])
28                  self.mon_data.append((self.data))
29                  reset data
30              end for
31          end if
32          return (self.mon_data)
```

Algorithm 5: Platform Workflow (Analyze)

```
    inputs  : $pt$, $R$, $link\_util$, $mon\_data$, $U$, $p_{nb}$: policy number from $pt$,
              $knowledge\_intent$: status and ID of intent
    outputs : $P_d$, $result$, $knowledge\_intent$
 1  Class Analyze():
 2      __init__(self, type_input, input, pt, **kwargs)
 3      def analyze(self, policy):
 4          if (kwargs) then
 5              result ← self.initiate_workflow(self.workflow, policy)
 6          else
 7              if self.type_input = 'declarative' then
 8                  if policy_exist('analyzePolicyStatus') then
 9                      policy ← self.pt.nodes[$p_{nb}$]
10                  else
11                      // Generate & Enforce $P_{7-4}$ to analyze $P_d$:
12                      E = PolicyStatus('policy', policy, self.pt)
13                      policy, self.pt ← generate(D, E, A, C, pt)
14                  end if
15                  result ← policy['Action'](policy)
16              end if
17          end if
18          // Returns False as policy status is found to be False:
19          return (False, result)
20  Subclass PolicyStatus():
21      __init__(self, type_input, input, pt, **kwargs)
22      def analyzePolicyStatus(self, policy):
23          if policy_exist('monitor') then
24              self.R←self.pt.nodes[$p_{nb}$]['Enforcer'].R
25              self.mon_data←self.pt.nodes[$p_{nb}$]['Enforcer'].mon_data
26              self.link_util←self.pt.nodes[$p_{nb}$]['Enforcer'].link_util
27          end if
28          if self.link_util then
29              if policy_exist('analyzeValidate') then
30                  policy ← self.pt.nodes[$p_{nb}$]
31              else
32                  // Generate & Enforce $P_{7-5}$ to validate:
33                  E = Validate('policy', policy, self.pt)
34                  policy, self.pt ← generate(D, E, A, C, pt)
35              end if
36              result ← policy['Action'](policy)
37          end if
38          // Set policy status to False or True:
39          if False $\in$ result then
40              return (False, result)
41          else
42              return (True, result)
43          end if
44  Subclass Validate():
45      __init__(self, type_input, input, pt, **kwargs)
46      def analyzeValidate(self, policy):
47          if link_util then
48              result ← validate_link_util(mon_data, link_util)
49              knowledge_intent[intent_id] ← result['status']
50          end if
51          return (knowledge_intent, result)
```

action that decomposes to another action to verify the status of P_7. When P_{7-4} is enforced in line 15, the *analyzePolicyStatus* method is invoked. Within this method we ensure a monitor policy exists in the PT (line 23), so the necessary monitoring data is available. The link utilization category determined by *Monitor* guides the analysis workflow to the next job, that is to validate for the link utilization policy constraint of P_7. Policy P_{7-4} is decomposed to another policy (P_{7-5}) with a definitive action *analyzeValidate* that is generated and enforced through lines 29-36. P_{7-5} validates the link utilization for each resource (line 48) to determine if P_7 is enforced. If so, no further action is required and the policy status of P_7 is a logical *True*. Otherwise, App_1 will generate and enforce policy P_{7-6} (Algorithm 2, lines 22-28) targeting Planner Pl_1 to plan how to resolve the violations.

Policy P_{7-6} is decomposed to policies P_{7-7} and P_{7-8} to plan the network optimization. The platform plan workflow uses Cisco WAE optimizations to resolve network violations. When policy P_{7-6} is enforced, the workflow begins with the *plan* method from Algorithm 6, line 3. This method generates and enforces two optimization policies (lines 6-18), one for core (P_{7-7}) and one for edge (P_{7-8}) optimization. When each policy is enforced, the *planOptimize* method that uses the Cisco WAE APIs to perform available optimizations is called. For this use-case, we use the Cisco WAE Interior Gateway Protocol (IGP) metric optimization with parameters that include utilization thresholds, nodes, interfaces, and

Algorithm 6: Platform Workflows (Plan, Execute)

```
inputs  : $P_d$, policy, pt, R
outputs : result
Class Plan():
    __init__(self, type_input, input, pt, **kwargs)
    def plan(self, policy):
        result = []
        D = policy['Enforcer']
        if policy_exist('planOptimize') then
            for pol ∈ ($p_{nb}$) do
                policy ← self.pt.nodes[pol]
                result.append(policy['Action'](policy))
            end for
        else
            // Generate & Enforce $P_{7-7}$, $P_{7-8}$ to optimize:
            for each pair (r, m) ∈ self.R do
                E = Optimize('policy', policy, pt)
                policy, pt ← generate_policy(D, E, A, C, pt)
                result.append(policy['Action'](policy))
                D = policy['Definer']
            end for
        end if
        // If successful optimization:
        return (True, result)
Class Optimize():
    __init__(self, type_input, input, pt, **kwargs)
    def planOptimize(self, policy):
        result ← metricOptimization(policy)
        return (result)
Class Execute():
    __init__(self, type_input, input, pt, **kwargs)
    def execute(self, policy):
        if policy_exist('executeConfigureLSP') then
            policy ← self.pt.nodes[$p_{nb}$]
        else
            // Generate & Enforce $P_{7-10}$ to execute configuration:
            E = ConfigureLSP('policy', policy, pt)
            policy, pt ← generate_policy(D, E, A, C, pt)
        end if
        result ← policy['Action'](policy)
        // If successful configuration:
        return (True, result)
Class ConfigureLSP():
    __init__(self, type_input, input, pt, **kwargs)
    def executeConfigureLSP(self, policy):
        // Execute configuration workflow based on $P_{7-10}$
        if (self.result_plan) then
            result ← self.initiate_workflow(self.result_plan, policy)
        return (result)
```

optimization type. Using the metric optimization we are implicitly calculating the paths by optimizing the IS-IS metric. The solution contains new LSP paths (for demand D3 only) that enable traffic load-sharing across two paths to edge node ER_4.

After Planner Pl_1 has decided on a plan, App_1 generates a definitive policy, P_{7-9} that asks an Executer E_1 to execute the plan (Algorithm 2, line 33). We provide an example of a platform execute workflow in Algorithm 6, lines 27-46. At the platform level P_{7-10} is generated (lines 30-35) to start the plan execution through $E_{1.1}$, a child of Executer E_1. Policy P_{7-10} is an imperative policy that specifies the configuration changes for the new LSP paths for demand D3. In our simulated environment we are using Cisco WAE Design to apply the optimization changes in line 45 through a WAE workflow to which we provide as input a WAE plan file. The plan file is a WAE network snapshot generated when the optimizations were run during the planning stage of the MAPE loop. The changes include updating the IGP metrics and LSP paths for demand D3. We note that the *Execute* component in practise would include some configuration management utilities that would take the above input and convert it into a set of configuration changes that need to be applied at the physical device level for instance. Hence, in a real network setting the decomposition could result with more definitive and imperative policies: for instance, policies for the configuration conversion, as well as policies for the orchestration of the configuration-specific imperative policies to deploy the resolution from the Planner. However, the simulated setting that we are using corresponds exactly to what would be done in a service provider or any other operational environment.

Last, we present the results from the intent refinement. Figure 12 shows the changes in Cisco WAE Design for the LSP paths used by demand $D3$ only. The blue-colored links represent the initial LSP path before the intent refinement, whereas the brown arrows show the updated LSP paths once P_7 is enforced. The updated LSP paths enable equal traffic load-sharing, from core node CR_2 to edge node ER_4. The applied changes result with resolved violations in both core and edge. In Figure 13, we show the traffic volume for all demands, before and after the traffic volume increased for demands $D2$ and $D3$. And, in Figure 14, we show the impact of the intent refinement upon the link utilization: once demands $D2$ and $D3$ increase, the intent becomes violated as a result of the increased link utilization in edge ($CR_5 - ER_4 > 45\%$) and in core ($CR_4 - CR_5 > 70\%$). Through the refinement, the autonomic system responds by redistributing delay tolerant demand $D3$, such that edge and core network link utilization is in accordance with the intent I_7. This can also be seen from Figure 14, where after $t = 10s$, the link utilization decreases for both links ($CR_5 - ER_4$ and $CR_4 - CR_5$).

Next, we provide a brief discussion in terms of use-case generalizability and performance, for the described system.

A. DISCUSSION

1) USE-CASE GENERALIZABILITY

It is desirable that the system supports different use-cases. The system can be generalized to accommodate more use-cases by expanding on the different components, such as vocabulary sets, sets of policies, and sets of checks and balances in each of the M, A, P, E parts. With these enrichments, we believe that the majority intents from a network user would be handled.

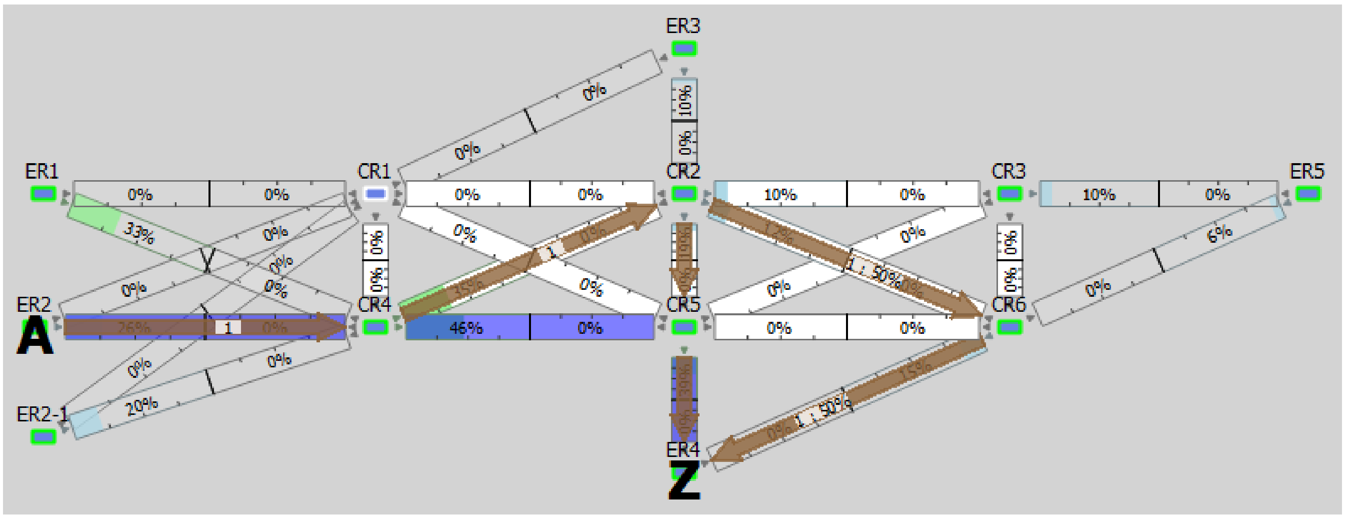


FIGURE 12. Post optimization link utilization state. No network violations exist. The links highlighted in blue represent the initial LSP path. The links in brown show the updated LSP paths with a 50% equal load-share from node CR_2 to ER_4 across two different sets of links: $CR_2 - CR_5 - ER_4$ and $CR_2 - CR_6 - ER_4$.

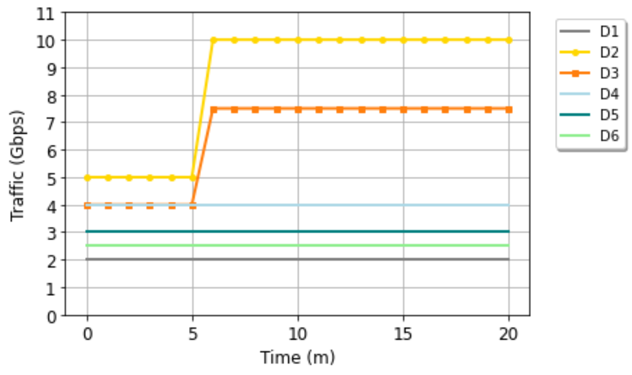


FIGURE 13. Traffic volume for all 6 demands.

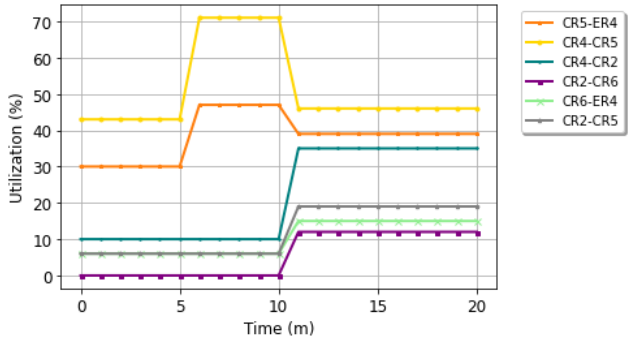


FIGURE 14. Link utilization as a result of traffic increase and the reaction of the autonomic system in accordance with the intent. After the configuration changes, link utilization decreases in both $CR_4 - CR_5$ and $CR_5 - ER_4$, and increases in the remaining links.

2) PERFORMANCE

Intent-based frameworks could potentially face long delays between the intent compilation and installation in the network. To address this, we consider having a fast loop and a slow loop. The fast loop enables a quick reaction to the problem, whereas the slow loop occurs in the background and tries to re-optimize, so we obtain the optimal solution later, which is the time-consuming part. To create a quick response in the fast loop, we consider having pre-defined policies that we can execute to take actions and enable us to react fast by deploying a temporary solution.

VI. CONCLUSION AND FUTURE WORK

We presented and implemented our proposal towards a self-driving management system for future networks, and demonstrated a proof of concept for the refinement of an intent. Our proposal addresses the following elements of a self-driving network: intent, closed-control loop, and policy-based execution of control loops. We focused on the overall process of intent refinement, and below we outline our takeaways and directions that require additional attention in future research.

The first level of intent demystification is closely coupled with the vocabulary, and Natural Language Processing (NLP). Additional work is required to determine: what is the initial vocabulary set; how can we expand the vocabulary set, and how can recent advances in ML and NLP improve our ability to demystify intents. In addition, further work on intent validation is needed in different parts of the intent refinement. For example, 1) to validate the intent itself (e.g. request, syntax, parameter values, etc.), or 2) to perform an offline execution of the intent in order to validate the intent deployment. The latter requires that we validate the imperative actions in a non-production environment, such as a network twin. Post checks can then be used to discover any unforeseen impacts of the intent deployment prior the actual deployment. Through validation we ensure that the intent deployment is both correct and safe to apply in a production environment.

A key aspect of self-driving networks is learning. In this paper, we used a heuristic approach to define the workflows that support the intent refinement. In future work, we plan to explore learning strategies to support the intent refinement and the creation of policy trees.

Last, we proposed an information model that models policy across different policy abstraction types that is capable to model goal, utility and action policies at the same time. However, the purpose of information models is to present known facts and not for discovering new knowledge [42]. Autonomic management requires both the discovery of new knowledge as well as updating the existing one. Therefore, in order to enable autonomic policy management, we plan to augment our proposed policy model with an ontology using Web Ontology Language (OWL) to manage the network as well as govern the management actions and their interactions through inference and reasoning. Furthermore, we plan to develop a language for autonomic management following our proposed architecture and policy model. We intend to use the proposed framework for various scenarios including multi-layer resource management.

GLOSSARY OF ABBREVIATIONS

MAPE-K: Monitor-Analyze-Plan-Execute-Knowledge
AC: Autonomic computing
ECA: Event-Condition-Action
PBM: Policy-Based Management
PEP: Policy Enforcement Point
PDP: Policy Decision Point
PMT: Policy Management Tool
PR: Policy Repository
MIB: Management Information Base
IBN: Intent-Based Networking
NBI: northbound interface
AM: Autonomic Manager
API: Application Programming Interface
AMS: Autonomic Management System
AMP: Autonomic Management Platform
SD-WAN: Software-defined wide-area network
UML: Unified Modeling Language
NLP: Natural Language Processing
TE: Traffic Engineering
NE: Network Engineering
SE: System Engineering
PCE: Path Computation Engine
QoS: Quality of Service
VPN: Virtual Private Network
VNF: Virtual Network Function
NFV: Network Function Virtualization
SDN: Software-Defined Networking
P: Policy
D: Policy Definer
E: Policy Enforcer
A: Policy Action
$\overrightarrow{C}$: Policy Constraint
$\overrightarrow{R}$: Policy Resource attributes
r_i: policy resource
m_i: policy resource metric
$\overrightarrow{T}$: Policy Temporal attributes
$\overrightarrow{S}$: Policy Spatial attributes
$\overrightarrow{PMD}$: Policy meta-data
ID: Policy Identifier
DM: Domain where Policy is enforced
EX: Policy Expiration date
PR: Policy Priority
AP: Policy Autonomic permission
ICA: Integrity and Consistency Analysis
PT: Policy Tree
WAE: Cisco WAN Automation Engine Design simulator
CR: Core Node
ER: Edge Node
MPLS: Multiprotocol Label Switching
LSP: Label-Switched Path
SR: Segment Routing
MPLS SR: MPLS-based Segment Routing
SID: Segment Identifier
adj-SID: adjacency segment identifier
DT: Delay Tolerant
DS: Delay Sensitive
D: Demand
App: Application
IGP: Interior Gateway Protocol
IS-IS: Intermediate System to Intermediate System
OWL: Web Ontology Language
I: Intent
pt: policy tree
U: utility action lookup
R: policy resources
(r, m): resource and metric tuple
o_M, o_A, o_P, o_E: outcomes from analyze, plan, execute
P_{dec}: declarative policy

ACKNOWLEDGMENT

The authors would like to thank Dr. Mehdi Shajari and Sayed Ehsan Etesami for discussions about this work. They would also like to thank Professor Raouf Boutaba and Professor James Won-Ki Hong for reviewing an early version of this work and providing valuable feedback.

REFERENCES

[1] C. Hare, *Simple Network Management Protocol (SNMP)*, document RFC 1098, RFC Editor, M. Fedor, M. L. Schoffstall, D. J. D. Case, and J. R. Davin, Eds., Apr. 1989. [Online]. Available: https://rfc-editor.org/rfc/rfc1098.txt
[2] T. Benson, A. Akella, and D. A. Maltz, "Unraveling the complexity of network management," in *Proc. NSDI*, 2009, pp. 335–348.
[3] H. Kim, T. Benson, A. Akella, and N. Feamster, "The evolution of network configuration: A tale of two campuses," in *Proc. ACM SIGCOMM Conf. Internet Meas. Conf.*, 2011, pp. 499–514.
[4] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani, "A survey on intent-driven networks," *IEEE Access*, vol. 8, pp. 22862–22873, 2020.
[5] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," 2017, *arXiv:1710.11583*.
[6] P. Kalmbach, J. Zerwas, P. Babarczi, A. Blenk, W. Kellerer, and S. Schmid, "Empowering self-driving networks," in *Proc. Afternoon Workshop Self-Driving Netw.*, Aug. 2018, pp. 8–14.

[7] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[8] R. Murch, *Autonomic Computing*. Indianapolis, IN, USA: IBM Press, 2004.

[9] A. Bandara, N. Damianou, E. Lupu, M. Sloman, and N. Dulay, "Policy-based management," in *Handbook of Network and System Administration*. Amsterdam, The Netherlands: Elsevier, 2008, pp. 507–563.

[10] W. Han and C. Lei, "A survey on policy languages in network and security management," *Comput. Netw.*, vol. 56, no. 1, pp. 477–489, Jan. 2012.

[11] J. O. Kephart and W. E. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Proc. 5th IEEE Int. Workshop Policies Distrib. Syst. Netw. (POLICY)*, Jun. 2004, pp. 3–12.

[12] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, vol. 26. Englewood Cliffs, NJ, USA: Prentice-Hall, 1995.

[13] D. C. Verma, "Simplifying network administration using policy-based management," *IEEE Netw.*, vol. 16, no. 2, pp. 20–26, Apr./May 2002.

[14] D. Verma, S. Calo, S. Chakraborty, E. Bertino, C. Williams, J. Tucker, and B. Rivera, "Generative policy model for autonomic management," in *Proc. IEEE SmartWorld, Ubiquitous Intell. Comput., Adv. Trusted Comput., Scalable Comput. Commun., Cloud Big Data Comput., Internet People Smart City Innov. (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2017, pp. 1–6.

[15] J. A. Wickboldt, W. P. D. Jesus, P. H. Isolani, C. B. Both, J. Rochol, and L. Z. Granville, "Software-defined networking: Management requirements and challenges," *IEEE Commun. Mag.*, vol. 53, no. 1, pp. 278–285, Jan. 2015.

[16] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—Degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, Aug. 2008, doi: 10.1145/1380584.1380585.

[17] N. Samaan and A. Karmouch, "Towards autonomic network management: An analysis of current and future research directions," *IEEE Commun. Surveys Tuts.*, vol. 11, no. 3, pp. 22–36, 3rd Quart., 2009.

[18] N. Agoulmine, S. Balasubramaniam, D. Botvich, J. Strassner, E. Lehtihet, and W. Donnelly, "Challenges for autonomic network management," in *Proc. IEEE Int. Workshop Modelling Autonomic Commun. Environ. (MACE)*, 2006.

[19] Z. Movahedi, M. Ayari, R. Langar, and G. Pujolle, "A survey of autonomic network architectures and evaluation criteria," *IEEE Commun. Surveys Tuts.*, vol. 14, no. 2, pp. 464–490, 2nd Quart., 2012.

[20] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, no. 2, pp. 223–259, Dec. 2006.

[21] M. A. Khan, S. Peters, D. Sahinel, F. D. Pozo-Pardo, and X.-T. Dang, "Understanding autonomic network management: A look into the past, a solution for the future," *Comput. Commun.*, vol. 122, pp. 93–117, Jun. 2018.

[22] R. Boutaba, M. A. Salahuddin, N. Limam, S. Ayoubi, N. Shahriar, F. Estrada-Solano, and O. M. Caicedo, "A comprehensive survey on machine learning for networking: Evolution, applications and research opportunities," *J. Internet Services Appl.*, vol. 9, no. 1, pp. 1–99, Dec. 2018.

[23] L. Fallon, J. Keeney, and R. K. Verma, "Autonomic closed control loops for management, an idea whose time has come?" in *Proc. 15th Int. Conf. Netw. Service Manage. (CNSM)*, 2019, pp. 1–5.

[24] B. Jennings, S. Van Der Meer, S. Balasubramaniam, D. Botvich, M. O. Foghlú, W. Donnelly, and J. Strassner, "Towards autonomic management of communications networks," *IEEE Commun. Mag.*, vol. 45, no. 10, pp. 112–121, Oct. 2007.

[25] H. Derbel, N. Agoulmine, and M. Salaün, "ANEMA: Autonomic network management architecture to support self-configuration and self-optimization in IP networks," *Comput. Netw.*, vol. 53, no. 3, pp. 418–430, 2009.

[26] S. Singh, I. Chana, and R. Buyya, "STAR: SLA-aware autonomic management of cloud resources," *IEEE Trans. Cloud Comput.*, vol. 8, no. 4, pp. 1040–1053, Oct. 2020.

[27] M. H. Behringer, B. E. Carpenter, T. Eckert, L. Ciavaglia, and J. C. Nobre, *A Reference Model for Autonomic Networking*, document RFC 8993, May 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc8993.txt

[28] M. Caporuscio, M. D'Angelo, V. Grassi, and R. Mirandola, "Reinforcement learning techniques for decentralized self-adaptive service assembly," in *Proc. 5th Eur. Conf. Service-Oriented Cloud Comput. (ESOCC)*, Vienna Austria. Springer, 2016, pp. 53–68.

[29] A. Rodrigues, R. D. Caldas, G. N. Rodrigues, T. Vogel, and P. Pelliccione, "A learning approach to enhance assurances for real-time self-adaptive systems," in *Proc. 13th Int. Conf. Softw. Eng. Adapt. Self-Manag. Syst.*, May 2018, pp. 206–216.

[30] N. Belhaj, D. Belaïd, and H. Mukhtar, "Framework for building self-adaptive component applications based on reinforcement learning," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jul. 2018, pp. 17–24.

[31] E. Zavala, X. Franch, J. Marco, and C. Berger, "HAFLoop: An architecture for supporting highly adaptive feedback loops in self-adaptive systems," *Future Gener. Comput. Syst.*, vol. 105, pp. 607–630, Apr. 2020.

[32] S. T. Arzo, R. Bassoli, F. Granelli, and F. H. P. Fitzek, "Multi-agent based autonomic network management architecture," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 3, pp. 3595–3618, Sep. 2021.

[33] T. B. Meriem, R. Chaparadza, B. Radier, S. Soulhi, and A. P. López, "GANA-generic autonomic networking architecture," ETSI, Sophia Antipolis, France, White Paper 16, Oct. 2016.

[34] K. Tsagkaris, M. Logothetis, V. Foteinos, G. Poulios, M. Michaloliakos, and P. Demestichas, "Customizable autonomic network management: Integrating autonomic network management and software-defined networking," *IEEE Veh. Technol. Mag.*, vol. 10, no. 1, pp. 61–68, Mar. 2015.

[35] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "SIMPLE-fying middlebox policy enforcement using SDN," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, pp. 27–38, 2013Aug. 2013, doi: 10.1145/2534169.2486022.

[36] J. Rubio-Loyola, A. Galis, A. Astorga, J. Serrat, L. Lefevre, A. Fischer, A. Paler, and H. De Meer, "Scalable service deployment on software-defined networks," *IEEE Commun. Mag.*, vol. 49, no. 12, pp. 84–93, Dec. 2011.

[37] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, "PolicyCop: An autonomic QoS policy enforcement framework for software defined networks," in *Proc. IEEE SDN Future Netw. Services (SDN4FNS)*, Nov. 2013, pp. 1–7.

[38] A. Binsahaq, T. R. Sheltami, and K. Salah, "A survey on autonomic provisioning and management of QoS in SDN networks," *IEEE Access*, vol. 7, pp. 73384–73435, 2019.

[39] Z. Zhao, E. Schiller, E. Kalogeiton, T. Braun, B. Stiller, M. T. Garip, J. Joy, M. Gerla, N. Akhtar, and I. Matta, "Autonomic communications in software-driven networks," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11, pp. 2431–2445, Nov. 2017.

[40] D. R. C. Moore, J. Strassner, E. J. Ellesson, and A. Westerinen, *Policy Core Information Model—Version 1 Specification*, document RFC 3060, Feb. 2001. [Online]. Available: https://rfc-editor.org/rfc/rfc3060.txt

[41] J. Strassner, "DEN-ng: Achieving business-driven network management," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. Manage. Solutions New Commun. World (NOMS)*, Apr. 2002, pp. 753–766.

[42] J. Strassner, J. N. de Souza, S. van der Meer, S. Davy, K. Barrett, D. Raymer, and S. Samudrala, "The design of a new policy model to support ontology-driven reasoning for autonomic networking," *J. Netw. Syst. Manage.*, vol. 17, nos. 1–2, pp. 5–32, Jun. 2009.

[43] J. van der Ham, J. Stéger, S. Laki, Y. Kryftis, V. Maglaris, and C. de Laat, "The NOVI information models," *Future Gener. Comput. Syst.*, vol. 42, pp. 64–73, Jan. 2015.

[44] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker, "Don't mind the gap: Bridging network-wide objectives and device-level configurations: Brief reflections on abstractions for network programming," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 5, pp. 104–106, Nov. 2019, doi: 10.1145/3371934.3371965.

[45] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," *ACM SIGPLAN Notices*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[46] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting diverse dynamic intent-based policies using Janus," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, Nov. 2017, pp. 296–309.

[47] D. Tuncer, M. Charalambides, G. Tangari, and G. Pavlou, "A northbound interface for software-based networks," in *Proc. 14th Int. Conf. Netw. Service Manage. (CNSM)*, 2018, pp. 99–107.

[48] Y. Han, J. Li, D. Hoang, J.-H. Yoo, and J. W.-K. Hong, "An intent-based network virtualization platform for SDN," in *Proc. 12th Int. Conf. Netw. Service Manage. (CNSM)*, Oct. 2016, pp. 353–358.

[49] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining network intents for self-driving networks," in *Proc. Afternoon Workshop Self-Driving Netw. (SelfDN)*, Aug. 2018, pp. 15–21.

[50] C. Rotsos, A. Farshad, D. King, D. Hutchison, Q. Zhou, A. J. G. Gray, C.-X. Wang, and S. McLaughlin, "ReasoNet: Inferring network policies using ontologies," in *Proc. 4th IEEE Conf. Netw. Softw. Workshops (NetSoft)*, Jun. 2018, pp. 159–167.

[51] R. Soule, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster, "Merlin: A language for managing network resources," *IEEE/ACM Trans. Netw.*, vol. 26, no. 5, pp. 2188–2201, Oct. 2018.

[52] C. C. Machado, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho, "An EC-based formalism for policy refinement in software-defined networking," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2015, pp. 496–501.

[53] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang, "PGA: Using graphs to express and automatically reconcile network policies," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, Aug. 2015, pp. 29–42.

[54] C. Janz, N. Davis, D. Hood, M. Lemay, D. Lenrow, L. Fengkai, F. Schneider, J. Strassner, and A. Veitch, "Intent NBI—Definition and principles," Open Netw. Found., Menlo Park, CA, USA, Version 2, Tech. Rep. ONF TR-523, 2015.

[55] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. (Feb. 2021). Intent-based networking—Concepts and definitions. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-03

[56] C. Li, O. Havel, W. S. Liu, A. Olariu, P. Martinez-Julia, J. C. Nobre, and D. Lopez. (Mar. 2021). Intent classification. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-intent-classification-03

[57] D. Chen, H. Yang, and K. Yao. (Feb. 2021). Network measurement intent. Internet Engineering Task Force. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-yang-nmrg-network-measurement-intent-01

[58] E. Zeydan and Y. Turk, "Recent advances in intent-based networking: A survey," in *Proc. IEEE 91st Veh. Technol. Conf. (VTC-Spring)*, May 2020, pp. 1–5.

[59] R. Cohen, K. Barabash, B. Rochwerger, L. Schour, D. Crisan, R. Birke, C. Minkenberg, M. Gusat, R. Recio, and V. Jain, "An intent-based approach for network virtualization," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage. (IM)*, May 2013, pp. 42–50.

[60] P. Sköldström, S. Junique, A. Ghafoor, A. Marsico, and D. Siracusa, "DISMI—An intent interface for application-centric transport network services," in *Proc. 19th Int. Conf. Transparent Opt. Netw. (ICTON)*, Jul. 2017, pp. 1–4.

[61] F. Callegati, W. Cerroni, C. Contoli, and F. Foresta, "Performance of intent-based virtualized network infrastructure management," in *Proc. IEEE Int. Conf. Commun. (ICC)*, May 2017, pp. 1–6.

[62] M. Kiran, E. Pouyoul, A. Mercian, B. Tierney, C. Guok, and I. Monga, "Enabling intent to configure scientific networks for high performance demands," *Future Gener. Comput. Syst.*, vol. 79, pp. 205–214, Feb. 2018.

[63] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. D. Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville, "INSpIRE: Integrated NFV-based intent refinement environment," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 186–194.

[64] Y.-W.-E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng, "Robotron: Top-down network management at Facebook scale," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 426–439.

[65] M. Pham and D. B. Hoang, "SDN applications—The intent-based northbound interface realisation for extended applications," in *Proc. IEEE NetSoft Conf. Workshops (NetSoft)*, Jun. 2016, pp. 372–377.

[66] G. Davoli, W. Cerroni, S. Tomovic, C. Buratti, C. Contoli, and F. Callegati, "Intent-based service management for heterogeneous software-defined infrastructure domains," *Int. J. Netw. Manage.*, vol. 29, no. 1, p. e2051, Jan. 2019.

[67] W. Cerroni, C. Buratti, S. Cerboni, G. Davoli, C. Contoli, F. Foresta, F. Callegati, and R. Verdone, "Intent-based management and orchestration of heterogeneous openflow/IoT SDN domains," in *Proc. IEEE Conf. Netw. Softw. (NetSoft)*, Jul. 2017, pp. 1–9.

[68] Y. Tsuzaki and Y. Okabe, "Reactive configuration updating for intent-based networking," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, 2017, pp. 97–102.

[69] J. McNamara, L. Fallon, and E. Fallon, "A mechanism for intent driven adaptive policy decision making," in *Proc. 16th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2020, pp. 1–3.

[70] B. E. Ujcich, A. Bates, and W. H. Sanders, "Provenance for intent-based networking," in *Proc. 6th IEEE Conf. Netw. Softw. (NetSoft)*, Jun. 2020, pp. 195–199.

[71] A. Leivadeas and M. Falkner, "VNF placement problem: A multi-tenant intent-based networking approach," in *Proc. 24th Conf. Innov. Clouds, Internet Netw. Workshops (ICIN)*, Mar. 2021, pp. 143–150.

[72] M. Riftadi and F. Kuipers, "P4I/O: Intent-based networking with P4," in *Proc. IEEE Conf. Netw. Softw. (NetSoft)*, Jun. 2019, pp. 438–443.

[73] J.-M. Kang, J. Lee, V. Nagendra, and S. Banerjee, "LMS: Label management service for intent-driven cloud management," in *Proc. IFIP/IEEE Symp. Integr. Netw. Service Manage. (IM)*, May 2017, pp. 177–185.

[74] P. H. Gomes, M. Buhrgard, J. Harmatos, S. K. Mohalik, D. Roeland, and J. Niemöller, "Intent-driven closed loops for autonomous networks," *J. ICT Standardization*, vol. 9, pp. 257–290, Jun. 2021.

[75] N. F. S. De Sousa, D. L. Perez, C. E. Rothenberg, and P. H. Gomes, "End-to-end service monitoring for zero-touch networks," *J. ICT Standardization*, vol. 9, no. 2, pp. 91–112, May 2021.

[76] J. D. Moffett and M. S. Sloman, "Policy hierarchies for distributed systems management," *IEEE J. Sel. Areas Commun.*, vol. 11, no. 9, pp. 1404–1414, Dec. 1993.

[77] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman, "Policy refinement: Decomposition and operationalization for dynamic domains," in *Proc. 7th Int. Conf. Netw. Service Manage.*, 2011, pp. 1–9.

[78] (Mar. 21, 2020). *Network Management Research Group Charter-irtf-nmrg-02*. [Online]. Available: https://datatracker.ietf.org/doc/charter-irtf-nmrg/02/

[79] I. F. Akyildiz, A. Kak, and S. Nie, "6G and beyond: The future of wireless communications systems," *IEEE Access*, vol. 8, pp. 133995–134030, 2020.

[80] T. Yaqoob, M. Usama, J. Qadir, and G. Tyson, "On analyzing self-driving networks: A systems thinking approach," in *Proc. Afternoon Workshop Self-Driving Netw.*, Aug. 2018, pp. 1–7.

[81] A. Clemm, M. F. Zhani, and R. Boutaba, "Network management 2030: Operations and control of network 2030 services," *J. Netw. Syst. Manage.*, vol. 28, no. 4, pp. 721–750, Oct. 2020.

[82] S. C. Madanapalli, H. H. Gharakheili, and V. Sivaraman, "Assisting delay and bandwidth sensitive applications in a self-driving network," in *Proc. Workshop Netw. Meets AI ML*, 2019, pp. 64–69.

[83] J. Zerwas, P. Kalmbach, L. Henkel, G. Rétvári, W. Kellerer, A. Blenk, and S. Schmid, "NetBOA: self-driving network benchmarking," in *Proc. Workshop Netw. Meets AI ML*, 2019, pp. 8–14.

[84] H. B. Pasandi and T. Nadeem, "Towards a learning-based framework for self-driving design of networking protocols," *IEEE Access*, vol. 9, pp. 34829–34844, 2021.

[85] M. Usama, J. Qadir, A. Raza, H. Arif, K.-L.-A. Yau, Y. Elkhatib, A. Hussain, and A. Al-Fuqaha, "Unsupervised machine learning for networking: Techniques, applications and research challenges," *IEEE Access*, vol. 7, pp. 65579–65615, 2019.

[86] S. Schneider, A. Manzoor, H. Qarawlus, R. Schellenberg, H. Karl, R. Khalili, and A. Hecker, "Self-driving network and service coordination using deep reinforcement learning," in *Proc. 16th Int. Conf. Netw. Service Manage. (CNSM)*, Nov. 2020, pp. 1–9.

[87] X. Chen, R. Proietti, C.-Y. Liu, and S. J. Ben Yoo, "Towards self-driving optical networking with reinforcement learning and knowledge transferring," in *Proc. Int. Conf. Opt. Netw. Design Modeling (ONDM)*, May 2020, pp. 1–3.

[88] T. Mai, S. Garg, H. Yao, J. Nie, G. Kaddoum, and Z. Xiong, "In-network intelligence control: Toward a self-driving networking architecture," *IEEE Netw.*, vol. 35, no. 2, pp. 53–59, Mar. 2021.

[89] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and data-driven softwarized networks: Review, opportunities, and challenges," *Proc. IEEE*, vol. 107, no. 4, pp. 711–731, Apr. 2019.

[90] H. Huang, L. Zhao, H. Huang, and S. Guo, "Machine fault detection for intelligent self-driving networks," *IEEE Commun. Mag.*, vol. 58, no. 1, pp. 40–46, Jan. 2020.

[91] D. Pendarakis, D. R. Yavatkar, and D. R. Guerin, *A Framework for Policy-Based Admission Control*, RFC 2753, Jan. 2000. [Online]. Available: https://rfc-editor.org/rfc/rfc2753.txt

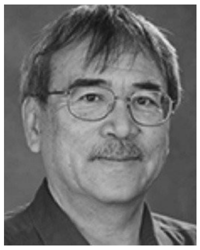
[92] T. Phan, J. Han, J.-G. Schneider, T. Ebringer, and T. Rogers, "A survey of policy-based management approaches for service oriented systems," in *Proc. 19th Austral. Conf. Softw. Eng. (ASWEC)*, Mar. 2008, pp. 392–401.

[93] Cisco Systems. (Sep. 2021). *WAN Automation Engine (WAE)*. [Online]. Available: https://www.cisco.com/c/en/us/td/docs/net_mgmt/wae/7-4-0/user_guide/cisco-wae-74-user-guide.html
[94] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, *Segment Routing Architecture*, document RFC 8402, Jul. 2018. [Online]. Available: https://rfc-editor.org/rfc/rfc8402.txt

KRISTINA DZEPAROSKA (Member, IEEE) received the B.Sc. degree in engineering from Ss. Cyril and Methodius University, Skopje, in 2015, and the M.A.Sc. degree from the Department of Electrical and Computer Engineering, University of Toronto, Canada, in 2018, where she is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. Her research interests include autonomic networks, IBN, SDN, and AI.

NASIM BEIGI-MOHAMMADI (Member, IEEE) received the Ph.D. degree in computer science from York University. She is currently a Postdoctoral Fellow with the University of Toronto. Her research interests include self-adaptive networks and service management. For more on her work, please visit (individual.utoronto.ca/nbm).

ALI TIZGHADAM (Member, IEEE) received the M.A.Sc. degree in electrical and computer engineering from the University of Tehran, in 1994, and the Ph.D. degree in electrical and computer engineering from the University of Toronto (UofT), in 2009. He is currently the Principal Technology Architect at TELUS Communications Inc. He is responsible for TELUS network softwarization strategy. He has led a team of architects and developers to design and implement TELUS Intelligent Network Analytics and Automation (TINAA) open ecosystem leveraging most recent advances in open source, big data, AI, SDN, and NFV to enable closed-loop intent-based automation and innovative service composition paving the road to realize self-driving networks dream in 5G world. In the academic side, he has designed a graduate course—Service Provider Networks—to bridge the gap between understanding of networks in academic area and service provider's domain. He is currently teaching this course with the Department of Electrical and Computer Engineering, UofT. Moreover, he is a Senior Researcher at UofT focusing on smart city applications. His research interests include span SDN, intent-based networking, end-to-end multi-layer orchestration, smart city applications, and applications of AI in networking.



ALBERTO LEON-GARCIA (Life Fellow, IEEE) is currently a Professor in electrical and computer engineering with the University of Toronto. He authored the textbooks, such as *Probability and Random Processes for Electrical Engineering* and *Communication Networks: Fundamental Concepts and Key Architecture*. He was the Founder and the CTO at AcceLight Networks, Ottawa, from 1999 to 2002. He was the Scientific Director of the NSERC Strategic Network for Smart Applications on Virtual Infrastructures (SAVI) and a Principal Investigator of the project on Connected Vehicles and Smart Transportation. SAVI designed and deployed a national testbed in Canada that converges cloud computing and software-defined networking. CVST designed and deployed an application platform for smart transportation. He is the Co-Founder and the CTO of StreamWorx.ai which offers massive-scale, real-time streaming analytics, and machine learning software for network operations and cybersecurity applications. He is a Life Fellow of the Institute of Electronics an Electrical Engineering "For contributions to multiplexing and switching of integrated services traffic."