Improving Intent Correctness with Automated Testing

Paul Alcock, Ben Simms, Will Fantom, Charalampos Rotsos, Nicholas Race
School of Computing and Communications, Lancaster University
{p.alcock1, b.simms, w.fantom, c.rotsos, n.race}@lancaster.ac.uk

Abstract—**Intent-based networking (IBN) systems have become the de-facto control abstraction to drive self-service, self-healing, and self-optimized capabilities in service delivery processes. Nonetheless, the operation complexity of modern network infrastructures make network practitioners apprehensive towards adoption in production, requiring further evidence for correctness. In this paper, we argue that testing, verification and monitoring should become first-class citizens in reference IBN architecture, in order to improve the detection errors during operations. Towards this goal, we present an extension for an intent architecture that allows IBN system to validate the correctness of network configuration using realistic network emulation. Furthermore, we present an intent use-case that ensure correct operation in hybrid networks.**

Index Terms—**Network Testing, Network Emulation, CI/CD, IBN, Intent-based Networking**

I. Introduction

In recent years, the network community has undergone a major transformation to cope with increasing global demand for network connectivity. In an effort to improve the efficiency of service delivery processes, network vendors have developed automated capabilities for self-provisioning, self-healing and self-optimization. Intent-based networking (IBN) has emerged as the de facto design approach to guide the development of new high-level interfaces. Research efforts [1], [2], [3] in intent modeling have demonstrated the ability of IBN systems to deliver highly-desired management capabilities, including strong policy fulfillment guarantees, user-friendly interfaces, and policy composition with automatic conflict resolution. In parallel, standardization bodies, like the ETSI ZSM [4] and the TM Forum [5], are actively exploring the design of reference network architectures and information models for IBN systems.

IBN adoption is limited in production however, as network operators remain skeptical about the paradigm's effectiveness for day-to-day autonomic network management and the value such systems can offer for service delivery. Since the emergence of early network technologies, network management has been a human-centric process, with management processes delivered by experienced network managers, using pen and paper. Operators are hesitant to transfer control of these processes to automated systems and intelligent algorithms, fearing the impact of non-human driven decisions on network resilience.

Automation hesitancy can be explained by a number of factors. Firstly, AI/ML models, the building blocks of automation systems, remain predominantly probabilistic and thus best-effort. This becomes increasingly challenging as shifting behavioral patterns can drastically change the accuracy of a model [6]. Detecting such dataset shifts, requires the deployment of dedicated monitoring mechanisms, while detection is not instantaneous [7], [8]. Secondly, modern IBN systems exert control across several technological and administrative domains in order to deliver high-level management intents. A key element in converging control across domains is the use of data models, like YANG, and remote management interfaces, like NETCONF. Although efforts are under way to develop interoperable YANG models, such as the OpenConfig model suite [9], network operators must still utilize vendor-specific model extensions in order to take full advantage of vendor equipment features. This in turn increases model complexity and can result in errors translating models to device configurations. Finally, although programmable technologies have evolved and reached production in the last decade, several parts of the network still rely on legacy devices with support for programmability (e.g. SDH, SONET switch), which remain external to the automation control systems. IBN system must support error detection mechanisms beyond their operational scope to ensure operational correctness. Ultimately, network operators are apprehensive that IBN systems will become another network feature to continually monitor and manage.

Despite this hesitancy, automation is becoming the norm for managing systems in several computing domains. Cloud computing systems can deploy code updates in the production systems in a matter of minutes/hours with strong correctness guarantees, and minimal user involvement. Additionally, automated systems utilize cross-layer monitoring systems, which assist operators in quickly detecting and localizing the root cause of failures in production systems. Supporting these capabilities required the cloud community to make drastic changes to their operational culture, delegating several service management tasks to CI/CD pipelines. Deployment processes use multi-stage testing suites to validate code and configuration correctness, simultaneously monitoring the system to collect and analyze large amounts of runtime data. This is a stark contrast to the common "fire and forget" approach of modern network management. In this respect, IBN systems can outperform human-driven approaches, since several decisions are generated using logical process which depend on the operational state of the system (e.g. topology, resource discovery, traffic matrix).

In this paper, we argue that effective adoption of IBN systems in production networks depends on the inclusion of automated testing throughout the intent lifecycle, providing operators with strong correctness guarantees for generated configurations. Relevant tests can use a wide range of validation tools available to modern network technologies (verification, simulation, emulation) to validate several configuration aspects. To meet these requirements, this paper introduces an extension on the intent lifecycle model developed by the IETF NMRG group architecture, which integrates automated configuration testing in the intent control loop. The proposed test stage will enhance the existing validation process with the ability to verify configuration changes in the target network environment. Furthermore, we present our efforts to integrate a testing mechanism with an open-source intent manager [10]. The testing mechanism uses the open source NEAT network testing framework [11], which uses realistic device models and network emulation to validate device configuration correctness. The proposed testing platform utilizes connectivity intent requirements to automatically generate testing suites tailored to the network configuration, and can deliver integration testing between legacy and programmable devices. We believe that intent standardization processes should further this design philosophy and introduce multi-stage verification processes in reference architecture designs.

In the remainder of this paper, we discuss relevant research efforts (§ II) and discuss the opportunities for testing and verification in the intent life-cycle (§ III). Furthermore, we elaborate on the testing extension in our IBN architecture (§ IV), we describe the implementation of a traffic drain intent with test-based verification support (§ V) and conclude our paper (§ VI).

II. Background

Current standardization efforts aim to govern automated processes with a closed control-loop, enabling the network to operate without necessary human input. IBN is a fundamental component in this process as it enables the abstraction between low-level network configurations and high-level user input. However, several features of IBN remain open challenges, limiting the development of autonomous use-cases. Nonetheless, a number of practical approaches have been explored in previous work which aim to enable supporting functionality. Janus [2] demonstrates a method of configuring networks using a graph-based policy model to generate non-conflicting, low-level network policies from an intent. Using PGA [1] as a foundation, Janus extends PGAs support from access control policies to include QoS and dynamic intent-based policies, simultaneously using network resource information to inform the composition of low-level policies; maximizing the number satisfied whilst minimizing the number of path changes resulting from runtime events (such as moving endpoints).

Over the past decade, there has been an increasing interest in the domain of network verification and testing. Although they share similar goals, network verification and network testing describe distinct approaches to validating network configurations. Network verification determines if a network complies

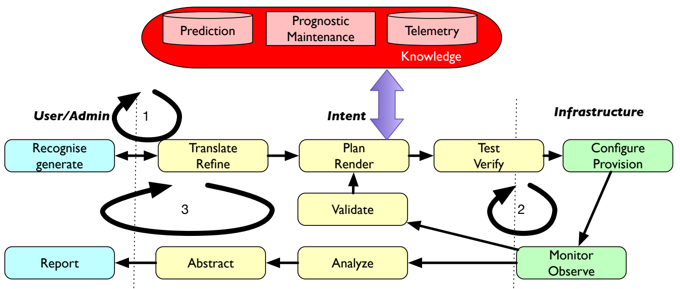


Fig. 1: A reference model of a network intent lifecycle.

with a set of invariants; properties of the network which must remain true throughout specified test cases. Formal methods, such as model checking, SAT/SMT solving and symbolic execution are commonly employed as methods of validating both online and offline configurations, usually on an extrapolated network model, rather than the live network directly. Plankton [12] is one example of a symbolic model checker, which uses a representative network model and packet equivalence classes to simulate network operations. Plankton is able to verify the symbolic operation of OSPF, BGP and static routing, and verify offline configurations. There have also been efforts to verify live network configurations in real-time. NetPlumber [13] speeds up the process of validating policy updates by analyzing incremental changes rather than periodic snapshots, using a rule dependency graph to evaluate the new change. However, verification approaches primarily consider the design of the network and cannot determine the validity of realized, dynamic execution states. Network testing provides a valuable complement to verification in directly validating the live network itself, using network packets. Pairing the two approaches allows configurations to be evaluated with a wider scope of parameters and detect issues which occur in implementation or at runtime. NetBOA [14] is an example of a black-box testing tool, which aims to identify weak-spots in network configurations for operational networks; using Bayesian Optimization to identify the most challenging traffic loads for a given network configuration. NetBOA is reliably able to determine near-optimal traffic conditions for up to 3 configuration parameters.

Further development of robust monitoring and configuration methods can enable IBN to facilitate use-cases which are dependent on the output of current standardization efforts. However, there is some work to be done in both areas to guarantee that generated configurations are suitable for complex network contexts, support suitable configuration parameters and scale to relevant topology sizes. An initial step applying these methods to intent-driven use cases is outlined in Rothenberg et al. [15], which demonstrates an intent-based, closed control-loop for converging service performance towards an inferred quality of experience objective, however this approach considers only QoS measures.

III. Intent Lifecycle

To highlight the importance of testing and verification in IBN systems, this section discusses the testing and verification challenges for IBN platforms using the IRTF intent lifecycle model [16], depicted in Figure 1. The model uses four domains to effectively deliver intents. The creation and deletion of intent is controlled by the user domain, which includes both network administrators and ISP clients. The delivery plan for an intent is defined by the intent layer, which can translate intent objects into a set of infrastructure configurations. Intent processing is supported by a knowledge layer, containing information about the state of the infrastructure, as well as, intelligent decision support services, such as anomaly detection and traffic prediction. The actual delivery of an intent is performed by the infrastructure, which offers a range of management interfaces to deploy network configuration changes on the infrastructure.

Between the four domains, three control loops are established in order to fulfil intents and to refine configuration. The first loop is established between the user and the intent layer and is responsible to capture the parameters of the intent. The opportunities for verification at this stage include initial sanity checks, ensuring the viability of the intent and breaking any semantic inconsistencies in user intent expression. In parallel, preliminary checks are performed to detect conflicts with other intents or network policies. A number of intent systems have explored the applicability of modern NLP and AI mechanisms to extract network intents from spoken language [10], [3] and to reduce semantic variability.

The second loop involves the intent and the infrastructure domain and ensures continuous intent delivery. The intent layer defines the configuration changes required to fulfil the intent requirements (e.g. create an MPLS LSP). These changes are estimated using optimization algorithms under an assumption of a single source of truth for the system and device models are used to translate intents into technology specific configurations. As a result, the new configuration is guaranteed to meet requirements, subject the assumptions of the optimization mechanism and the modeling of device behaviors, operation risks and failures. However, as the scope of an IBN system expands, so does the support for failure scenarios which reduces the effectiveness of modeling. Optimization and validation mechanisms should be coupled with network emulation [17], [11] and verification mechanisms [18] to expand testing coverage and incorporate complex elements that cannot be captured by the mathematical modeling that typically underpins these mechanisms. To reflect these requirements, we propose an extension in the IETF lifecycle model and include a ”test/verify“ stage in the lifecycle prior to the provision stage.

The final loop exists between the intent layer and the user; responsible for reporting intent state information. Reports are generated by analysis modules in the intent layer, which transform low-level monitoring information into intent-centric status updates, offering accurate, high-level notifications. Analysis at this stage, typically relies on low-level telemetry information available by default from the infrastructure (e.g. link load, VM resource utilization). Tests in this loop can be automatically generated based on intent KPIs and monitored throughout the intent lifecycle, using programmable control and even deploying special monitoring VNFs alongside the service. Further, testing opportunities are enabled using automated test generation mechanisms which can verify data plane correctness and security [19], [20]. Running multiple monitoring VNFs can incur a significant resource overhead, but modern advances in network virtualization in the form of Unikernels and containers can significantly reduce resource requirements and provide precise automation monitoring.

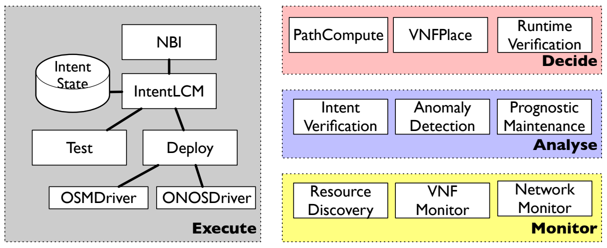


Fig. 2: Intent Layer Architecture

IV. Architecture

Our research efforts in intent testing are part of the Next Generation Converged Digital Infrastructure (NG-CDI) project [21], a research collaboration between British Telecom and four leading UK University partners. The project aims to develop completely new ways of operating network infrastructures and deliver closed-loop, self-optimizing, and self-healing operations. To support these operations, the NG-CDI project has developed an architecture and a strawman implementation of an IBN system [10]. In this section, we discuss the architectural extensions introduced in the NG-CDI intent architecture (§ IV-A), in order to validate intent fulfillment plans as part of the closed-control loop. Our validation mechanism, is based on NEAT, an open-source configuration validation too, based on network emulation.(§ IV-B). Finally, we discuss how our intent deployment process can be extended to generate testing suites tailored to the configuration changes of the infrastructure§ IV-C.

A. NG-CDI Intent Layer

Figure 2 depicts the architecture of our strawman IBN implementation. Intent management is organized in four layers, coordinated using an event-driven model. Changes in the intent life-cycle are driven by the *Monitor* layer, which contains services that extract monitoring information from the different components of the infrastructure and service components. Information is stored on a time-series database (Prometheus). An alerting service, running on top of the database, signals services in the *analyze* layer for the availability of new data, based on service-specific alert policies. Analyze services process monitoring data and can detect or predict (e.g. proactive maintenance) intent violations. Intent violations trigger the *Decide* layer to compute a new deployment plan and define a set of configuration updates. Finally, the *Execute* layer has access to low-level infrastructure management interfaces and can deploy the resulting configuration changes.

Using this multi-stage architecture, our intent layer can deliver a number of connectivity and management intents. Firstly, a "route" intent can establish paths between any set of network hosts, as well as, ensure that the traffic will traverse specific middlebox devices. Secondly, a "drain" intent, can update the forwarding policy of the network and redirects the path uses by a route intent, in order to isolate network device for maintenance. Thirdly, a "service protection" intent ensure that no configuration changes will occur during specific periods of time, in order to avoid possible service degradation during a major event (e.g. Olympic games). Finally, a *router update* intent, can automate the delivery of hardware upgrades on a network device and synchronize traffic drain operations during site engineer visits.

B. Automated Testing

NEAT [11] is an open-source network testing framework for network configuration. Network managers and developers can use NEAT to re-create realistic network topologies with custom network configurations, run asynchronous network tests and collect detailed information about test outcomes. Tests can include a wide range of network device types, including network namespaces, containers, Xen and KVM VMs, Unikernel appliances and vendor virtual appliances (Cisco, Nokia, Juniper). The architecture of NEAT is presented in Figure 3. The setup and execution of a test is enabled by a session manager module, which is responsible to boot the required VM and configure the network bridges that provide the required connectivity. Users can interact with the session manager using either a RESTful API or though a local CLI.

A network test is configured via a YAML file and an example test definition is depicted in Listing 1. A NEAT configuration file consists of two components: a list of network topologies and a list of test blocks running on top of a topology. A Topology is defined using Python scripts and an extended Mininet topology API with support for VM and container-based device types. Topologies can be supplemented with a set of files and directories that contain assets relevant to the network, for example, disk images for VMs should be included in the assets list. Optionally, a topology can be granted privileges such as access to the host Docker or LibVirt sockets. Furthermore, if topologies need more advanced configuration, such as configuration that responds to dynamic content in the topology, NEAT also provides a set of hook points for scripts to be executed. The test block defines a set of tests that should be executed on-top of a specific topology, as well as the test success criteria. NEAT offers a set of pre-packages test scenarios, including connectivity and bandwidth, and users can develop custom test scenarios, by developing bash scripts that abstract the execution of the test and the analysis of test results.

The design of the configuration system decouples test scenarios from topology specifications in order to facilitate NetDevOps methodologies. For example, this allows a VNF developer to produce a simple integration test suite for their appliance using a simple network, as well validate correctness in a real environment by sourcing topologies files from the network operator.

Listing 1: Sample test configuration for a network function test suite

```yaml
topologies:
  - name: clickos-loadbalancer
    topology: cdn-topo
    assets: ["$(pwd)/click-images"]
    libvirt: true
    post_start_script: ./install_routers.sh
blocks:
  - name: h1 connect h4
    variant: ping
    topologies: [clickos-loadbalancer]
    mutables: { sender: h1, target: h4, count: 5}
    expressions:
      - Sent == Received
```

C. Intent Testing

In order to integrate automated testing in the intent lifecycle, we have extended out *Execute* layer, and we have introduced a new Test service. The test service aims to fulfil two goals: Generate automatic test suites that validate configuration changes and execute automatically tests before the deployment of a configuration. Test generation is driven by service connectivity intents. Effectively, the intent layer translate each connectivity intent into a connectivity test, thus validating that host remains reachable after the application of a network update. The operational state of the network is derived from two configuration streams. On the one hand, we assume that any static configuration and topology information are available by the network operators, in a format which contains that latest network changes (e.g. Git repo). In parallel, the Test service, caches any configuration changes performed by active intent during the operation of the IBN. Composing the two configuration streams, can provide a realistic configuration snapshot, which can be replayed using realistic device models by NEAT. The Test service generate a NEAT YAML file, which can be executed using the NEAT CLI tools. The Test service monitors the execution of the test, and upon a failure, it requests from the intent lifecycle manager to compute a new deployment plan.

V. Use Case: Automated Intent Verification in hybrid networks

In order to demonstrate the applicability of our intent testing pipeline, we apply our testing functionality on a realistic network scenario, inspired by operational processes in the BT production network. Our experimental scenario utilizes a multi-layer topology, inspired by the 21CN BT network topology, and depicted in Figure 4. The network is separated in three zones: access, metro and core. The access and metro zones are emulated using 14 OpenFlow switches, using the Open VSwitch switch. The metro and access layers are programmable by a single ONOS instance, which uses L2 OpenFlow rules to forward traffic between Access hosts and Core routers. The Core zone consists of five Cisco 7200

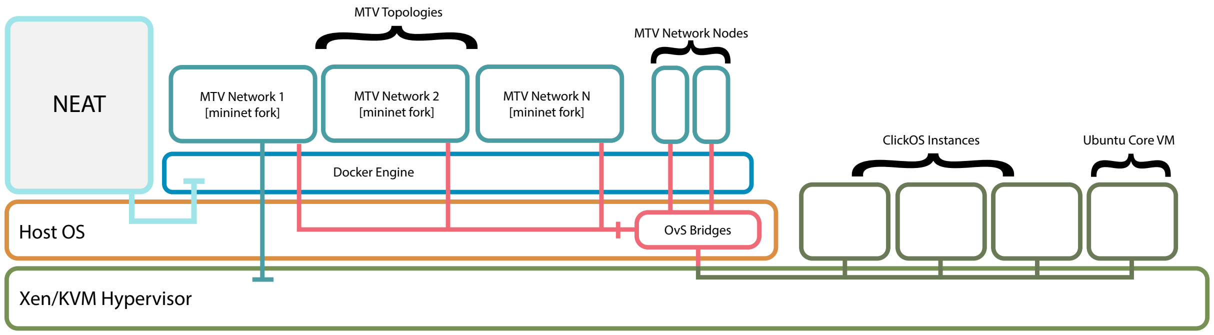


Fig. 3: NEAT architecture

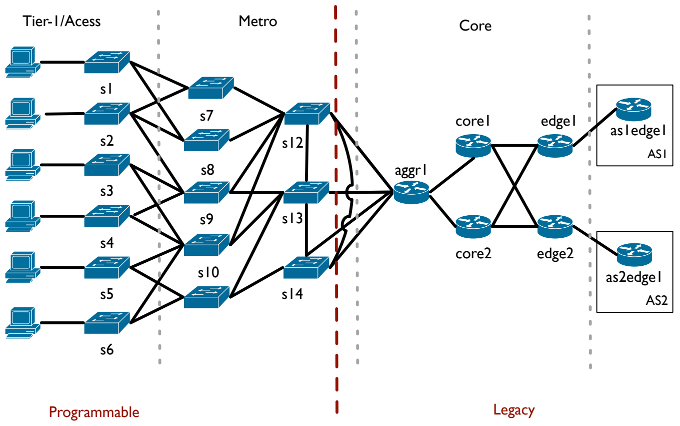


Fig. 4: A hybrid network topology, inspired by the 21CN BT topology, consisting of a range of programmable and legacy network devices.

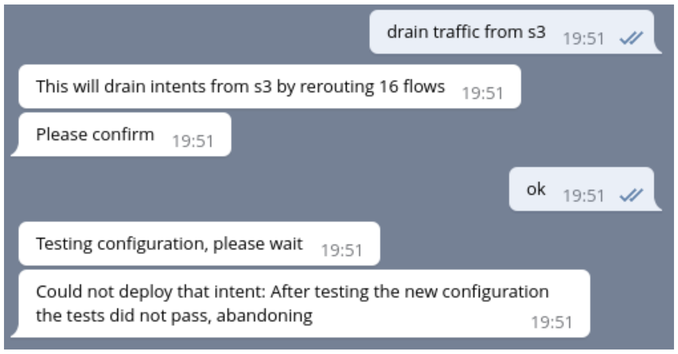
routers, emulated using the DynaMIPS platform [22], four of which are connected in a mesh topology. A fifth aggregation router uses a dedicated VLAN tag towards switches s12, s13, and s14 to connect the Metro and Core zones. The devices use handwritten IOS configuration files that route traffic from the access network subnet to adjacent ASes. Finally, the topology uses two Cisco7200 virtual routers to emulate 2 external ASes and establish BGP session with the two edge routers in our topology. The BGP configuration of the routers is minimal, full connectivity is allowed between the three ASes and path selection is achieved using a shortest path policy. The topology is used both to emulate our "operational" network, and our NEAT testing topology.

Our network design, aims to explore the ability of our intent testing pipeline to capture configuration errors in hybrid networks, that occur due to the split configuration between multiple administrative domains. To explore this testing dimension, we utilize the following configuration scenario. Our programmable part of the network is initialized with a series of connectivity intents which establish L2 paths between Access hosts and the Core aggregation switch using switch s12. After an initial cooldown period, a drain intent is sent to the intent manager for switch s12. The intent manager will compute a new set of paths between the Access nodes and the aggregation router which replace switch s12 with switches s13 and s14 and generates a set of ONOS REST API calls to install the new routes.

Before transmitting the new ONOS configuration in the production network, the intent manager runs a NEAT test to validate that connectivity is maintained by the new configuration. The NEAT test is bootstrapped with a static topology file and the configuration of the legacy devices, while the intent manager generates automatically the testing block section of the NEAT config, based on the active route intents. In order to test the correctness of the new configuration, we use ping tests between the Access nodes and the aggregation router. Each node transmits 10 ICMP Echo request packets and counts the number of successful ICMP ECHO responses received for the router within 10 seconds. The parameters of the ping test (destination host) are automatically estimated from the parameters of the initial connectivity intents. In order for the intent manager to install the new configuration to the production network, ping tests from access nodes should achieve a ping response rate of more than 50% (we allow some space for packet loss due to the best effort nature of network emulation). In case of a failed test, the intent manager will inform the user regarding the intent failure. The test outcome can also be used to collect detailed packet traces, which can be analyzed to perform an initial root cause analysis and generate precise user feedback.

We create two variation of the network scenario. A *v*alid experiment uses a static configuration for the legacy network, capable to receive and route all traffic from the programmable network. An *i*nvalid experiment uses a static configuration for the legacy network, which contains a VLAN tag misconfiguration. Specifically, the aggregation router configuration has an invalid VLAN tag configuration on the ports that connect to switches s13 and s14. Detecting such an error requires from a network engineer to closely inspect the configuration of all devices, in order to detect the source of the error, while the intent manager would assume that the intent was deployed successfully without a prior test, even if host traffic cannot reach the legacy network.

The results of our experimental scenarios are depicted in Figure 5. The hosts are able to ping each other, but tests fail when a host pings an IP beyond the programmable domain. Test results are analyzed by the intent manager and a short report is sent to the user using the Dialogflow/Telegram agent. In terms of the scalability of the testing system, the



drain traffic from s3 19:51

This will drain intents from s3 by rerouting 16 flows 19:51

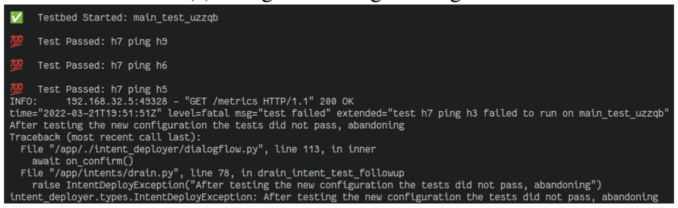
Please confirm 19:51

ok 19:51

Testing configuration, please wait 19:51

Could not deploy that intent: After testing the new configuration the tests did not pass, abandoning 19:51

(a) Telegram/Dialogflow Agent



```
✅  Testbed Started: main_test_uzzqb

💯  Test Passed: h7 ping h9

💯  Test Passed: h7 ping h6

💯  Test Passed: h7 ping h5
INFO:     192.168.32.5:49328 - "GET /metrics HTTP/1.1" 200 OK
time="2022-03-21T19:51:51Z" level=fatal msg="test failed" extended="test h7 ping h3 failed to run on main_test_uzzqb"
After testing the new configuration the tests did not pass, abandoning
Traceback (most recent call last):
  File "/app/./intent_deployer/dialogflow.py", line 113, in inner
    await on_confirm()
  File "/app/intents/drain.py", line 78, in drain_intent_test_followup
    raise IntentDeployException("After testing the new configuration the tests did not pass, abandoning")
intent_deployer.types.IntentDeployException: After testing the new configuration the tests did not pass, abandoning
```

(b) Intent Manager Logging

Fig. 5: Detecting and reporting a testing error for a “drain” intent using a NEAT connectivity test.

resulting test takes on average 33 seconds (std 1.7 sec across 10 runs) and the majority of the experimental time is spent on setting up the emulation environment (24 sec on average).

VI. Conclusion

Intent-based networking (IBN) systems have become the de facto control abstraction to drive self-service, self-healing, and self-optimized capabilities in service delivery processes. However, adoption of IBN is limited in production as network operators remain skeptical of IBNs effectiveness in day-to-day network operations. Several enabling features of IBN remain open implementation challenges as practical research has only demonstrated limited use-case for automated intent life-cycles. Without the capability to provide robust guarantees of correctness in generated configurations, network operators are apprehensive about introducing autonomous decision logic into performance critical infrastructure. This paper argued for the inclusion of network testing to support IBN use-cases and introduced a proof-of-concept tool to automatically validate intent-based configurations prior to deployment, using a network emulator to conduct testing.

In future work, we aim to expand the testing approaches to validate a variety of network features beyond reachability, enabling the system to compare network state with a more expressive intent model. Furthermore, we will explore how the intent lifecycle model can include further verification processes during run-time. Relevant verification processes can configure the underlying platform to monitor intent KPIs using programmable control and deploying special monitoring VNFs alongside the service.

Acknowledgement: The authors gratefully acknowledge the support of the Next Generation Converged Digital Infrastructure (NG-CDI) Prosperity Partnership project funded by UK’s EPSRC and British Telecom plc.

References

[1] 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,” *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, p. 29–42, aug 2015.
[2] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, “Supporting diverse dynamic intent-based policies using janus,” in *CoNEXT ’17*. ACM, 2017, p. 296–309.
[3] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, “Hey, lumi! using natural language for Intent-Based network management,” in *USENIX ATC 21*. USENIX Association, Jul. 2021, pp. 625–639.
[4] ETSI - Blog-ZSM - ETSI Blog. ETSI. [Online]. Available: https://www.etsi.org/newsroom/blogs/blog-zsm
[5] TM Forum - How to manage Digital Transformation, Agile Business Operations & Connected Digital Ecosystems. TM Forum. [Online]. Available: https://www.tmforum.org/
[6] S. Amos, “When training and test sets are different: Characterizing learning transfer,” in *Dataset Shift in Machine Learning*. The MIT Press, Dec. 2008, pp. 2–28.
[7] B. Schölkopf, D. Janzing, J. Peters, E. Sgouritsa, K. Zhang, and J. Mooij, “On causal and anticausal learning,” in *ICML’12*. Madison, WI, USA: Omnipress, 2012, p. 459–466.
[8] P. Kourouklidis, D. Kolovos, J. Noppen, and N. Matragkas, “A model-driven engineering approach for monitoring machine learning models,” in *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2021, pp. 160–164.
[9] “OpenConfig,” https://github.com/openconfig/public.
[10] M. Bezahaf, E. Davies, C. Rotsos, and N. Race, “To all intents and purposes: Towards flexible intent expression,” in *IEEE NetSoft 21*, 2021, pp. 31–37.
[11] W. Fantom, P. Alcock, B. Simms, C. Rotsos, and N. Race, “A NEAT way to test-driven network management,” in *IEEE/IFIP NOMS 22*, April 2022.
[12] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, “Plankton: Scalable network configuration verification through model checking,” pp. 953–967. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/prabhu
[13] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, “Real Time Network Policy Checking using Header Space Analysis,” p. 13.
[14] J. Zerwas, P. Kalmbach, L. Henkel, G. Rétvári, W. Kellerer, A. Blenk, and S. Schmid, “NetBOA: Self-Driving Network Benchmarking,” in *NetAI’19*. ACM, pp. 8–14.
[15] C. E. Rothenberg, D. A. Lachos Perez, N. F. Saraiva de Sousa, R. V. Rosa, R. U. Mustafa, M. T. Islam, and P. H. Gomes, “Intent-based Control Loop for DASH Video Service Assurance using ML-based Edge QoE Estimation,” in *NetSoft’20*. IEEE, pp. 353–355.
[16] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura, “Intent-based networking - concepts and definitions,” Working Draft, IETF Secretariat, Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-06, December 2021. [Online]. Available: https://www.ietf.org/archive/id/draft-irtf-nmrg-ibn-concepts-definitions-06.txt
[17] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, “Distrinet: A mininet implementation for the cloud,” *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, p. 2–9, Mar. 2021.
[18] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, “Verifying reachability in networks with mutable datapaths,” in *NSDI’17*. Boston, MA: USENIX Association, Mar. 2017, pp. 699–718.
[19] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, “Finding protocol manipulation attacks,” in *SIGCOMM ’11*. New York, NY, USA: ACM, 2011, p. 26–37.
[20] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, “Automatic test packet generation,” in *CoNEXT ’12*. New York, NY, USA: ACM, 2012, p. 241–252.
[21] “Next Generation Converged Digital Infrastructure,” http://ngcdi.com.
[22] “DynaMIPS (cisco router emulator),” https://github.com/GNS3/dynamips, 2021.