Refining Network Intents for Self-Driving Networks

Arthur Selle Jacobs
UFRGS
asjacobs@inf.ufrgs.br

Ricardo José Pfitscher
UFRGS
rjpfitscher@inf.ufrgs.br

Ronaldo Alves Ferreira
UFMS
raf@facom.ufms.br

Lisandro Zambenedetti Granville
UFRGS
granville@inf.ufrgs.br

ABSTRACT

Recent advances in artificial intelligence (AI) offer an opportunity for the adoption of self-driving networks. However, network operators or home-network users still do not have the right tools to exploit these new advancements in AI, since they have to rely on low-level languages to specify network policies. Intent-based networking (IBN) allows operators to specify high-level policies that dictate how the network should behave without worrying how they are translated into configuration commands in the network devices. However, the existing research proposals for IBN fail to exploit the knowledge and feedback from the network operator to validate or improve the translation of intents. In this paper, we introduce a novel intent-refinement process that uses machine learning and feedback from the operator to translate the operator's utterances into network configurations. Our refinement process uses a sequence-to-sequence learning model to extract intents from natural language and the feedback from the operator to improve learning. The key insight of our process is an intermediate representation that resembles natural language that is suitable to collect feedback from the operator but is structured enough to facilitate precise translations. Our prototype interacts with a network operator using natural language and translates the operator input to the intermediate representation before translating to SDN rules. Our experimental results show that our process achieves a correlation coefficient squared (*i.e.*, R-squared) of 0.99 for a dataset with 5000 entries and the operator feedback significantly improves the accuracy of our model.

CCS CONCEPTS

• **Networks** → **Network management**;

KEYWORDS

Intent-Based Networking, Self-Driving Networks, Machine Learning

1 INTRODUCTION

A *self-driving network* is an autonomous network that can predict changes and adapt to user behaviors without the intervention of an operator. Successfully implementing an autonomous network would not only ease network management but also reduce operational costs. Recent advances in artificial intelligence (AI) offer an opportunity for the adoption of self-driving networks, as machine learning models can identify patterns and learn how to respond to changes in the network. However, network operators still do not have the right tools to exploit these new developments in AI, since they still have to rely on low-level languages to specify network policies and complex interfaces to ensure that the specified policies are deployed correctly. Moreover, home-network users do not have the skills to program their networks and can benefit from a friendly management system.

Intent-based networking (IBN) allows operators to specify high-level policies that dictate how the network should behave—*e.g.*, defining goals related to quality of service, security, and performance—without worrying about the low-level details that are necessary to program the network to achieve these goals. Existing research proposals for IBN present several intent languages, frameworks, and compilers to deploy intents in network devices and middleboxes [1, 16, 17, 20]. These proposals enable composition of high-level policies [16, 17], deployment in software-defined networks (SDN) [1], and management abstractions for network operators [20]. While these are steps in the right direction, these proposals cannot extract intent information from pure natural language, requiring that network operators learn a new intent definition language in each proposal and, consequently, hindering interoperability, deployment, and management of heterogeneous networks.

Most of the existing research proposals for IBN fail to exploit the knowledge and feedback from the network operator. Highly complex and, sometimes, conflicting policies in network devices may cause network intents to derail from the desired behavior of the operator. Moreover, the adoption of programmable network technologies, such as SDN and Network Functions Virtualization (NFV) [8], introduce a new level of dynamism that results in constant changes in network conditions. Therefore, monitoring the network after deploying policies and requesting feedback from the operator are crucial for avoiding misconfigurations.

In this paper, we introduce a novel intent-refinement process that uses machine learning and feedback from the operator to translate the operator's utterances into network configurations (§2). Our process consists of three stages. First, we rely on an intelligent chatbot interface to extract the main actions and targets (*i.e., entities*) of an user intent from natural language (§2.1). We implement the chatbot interface using DialogFlow [6], which uses machine learning to identify key aspects in the user's utterances without the need for extensively covering every possible entity value. In our chatbot, examples of entities are the network endpoints, middleboxes, and temporal configurations for the policy. A natural language interface enables the deployment of our solution in distinct scenarios. For instance, a home user could use our chatbot to prioritize streaming traffic in her network during specific hours of the day.

Second, we use a neural sequence-to-sequence learning model to translate the extracted entities into a high-level structured network definition program (§2.2). The program is written in *Nile*, our new structured intent definition language (§3), which closely resembles natural language. The *Nile* program is then presented to the network operator for confirmation on the extracted behavior. For home users with no technical knowledge, the confirmation can come from a voice assistant or a graphical interface.

Finally, we compile the extracted intent program into a network policy according to the destination network (§2.3). As a proof-of-concept, we implement a service chain for specific traffic using SONATA-NFV [15](§4). However, the decoupling provided by the intent definition language allows the compilation of the intents to other existing network configurations—including policy languages, such as Janus [1], PGA [16], and Kinectic [11]—improving the reusability of our proposed solution. In this stage, we also make assertions to verify any conflicts between the extracted intent and the network configuration—e.g., an intent asking for more bandwidth than is available on the required path—and warn the operator through the chatbot interface.

In summary, our key contributions in this paper are:

(1) A novel intent-refinement process for intelligent extraction of intents from natural language that uses feedbacks from network operators to improve learning.
(2) *Nile*: a high-level, comprehensive intent definition language (§3) that resembles the English language. Nile acts as an abstraction layer for other policy mechanisms, reducing the need for operators to learn a new policy language for each different type of network.
(3) Experimental results that show significant improvements on translation accuracy with the feedback from the operator (§5).

2 REFINEMENT PROCESS

The first requirement for a self-driving network to reduce its management complexity is intelligent and seamless planning. A network operator should be able to specify network policies without worrying how they would be achieved. It would be even better if the network operator could use natural language to define the network behavior. The behavior may include customer expectations to comply with Service Level Agreements (SLAs), network functions for security, temporal behavior for accommodating large flows during peak hours, or network-wide goals like minimizing congestion or reducing traffic costs by relying on cheaper paths in the network.

With the above requirements in mind, we propose a refinement process for intent specification that can learn and adapt itself to achieve the network behavior expressed by the operator while providing a user-friendly interface for interactions with the operator. This section presents the three stages of the refinement process: entities' extraction, intent translation, and intent deployment. Figure 1 presents an overview of the refinement process with the three stages and the steps involved in translating intents described in natural language to network configurations. Note that the operator provides feedback via chatbot interface in Step 6, and Steps 2-6 are repeated until the operator confirms the correct translation of the intents.

2.1 Entities Extraction

The first step in the intent refinement process is to extract the actions and targets of the network behavior expressed in natural language by the operator. In this step, we use DialogFlow [6] to build the Entities Extractor. DialogFlow (formerly known as API.AI) is a development framework to build human-computer interactions based on natural language conversations (*i.e.*, chatbots). The framework uses machine learning to generalize example cases referred to

as *entities* and facilitate the extraction of features in the dialog. In our chatbot, the *entities* include middleboxes, SLA requirements, temporal restrictions, and endpoints targeted by the user's intent. One key advantage of using DialogFlow is the ability to deploy our chatbot across multiple platforms, including Google Assistant (present in numerous Google devices), Amazon's Alexa, or messaging apps, such as Slack and Facebook's Messenger. This feature can be helpful for a home-network user to configure her network using voice-activated assistants like Amazon's Alexa—for example, she could request parental control for her kids' devices.
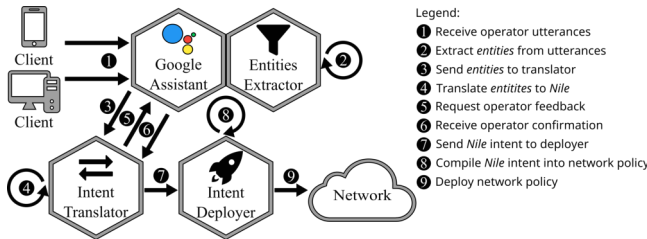

Figure 1: Intent Refinement Process.

Despite being extremely useful for user interactions, simply using a chatbot does not fulfill all the requirements for intent-based network planning. The *entities* extracted from natural languages result in key-value pairs representing the user utterances. However, these pairs do not reflect the network configuration commands. For instance, if a network operator asks a chatbot *"Please add a firewall for the backend."*, a possible extraction result, depending on how the chatbot is built and trained, would be the following *entities*: *{middleboxes: 'firewall'}, {target: 'backend'}*. Hence, after the chatbot interaction, we still need to translate the *entities* into a structured intent that can be implemented in a destination network.

2.2 Intent Translation

In DialogFlow, after the chatbot interface extracts all the required *entities* from the user utterances, the framework calls a Rest API in a backend service designated by a WebHook, which allows us to perform the heavy processing for translations. We configured a WebHook from our chatbot to our Intent Translator to receive all the extracted *entities*. These *entities* are fed to a previously trained sequence-to-sequence learning model [21], which translates *entities* to structured intents written in our *Nile* language (detailed in §3).

A neural sequence-to-sequence learning model consists of two Recurrent Neural Network (RNN) with Long Short-Term Memory (LSTM) hidden units: an *encoder* and a *decoder*. In this model, the RNN *encoder* processes the sequence of words (in our case the extracted *entities*) and generates a

thought vector, which is a numerical representation of the input sequence. The RNN *decoder* receives the thought vector as input and generates a sequence of words in the destination language (in our case *Nile*). Figure 2 shows an example of the encoding-decoding process. Note that the RNNs allow input and output sequences of different lengths.
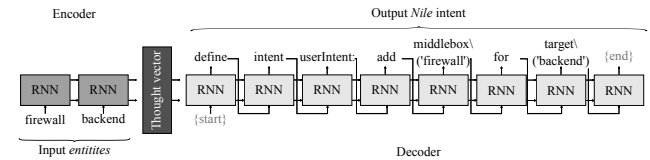

Figure 2: Sequence-to-sequence learning model.

One of the shortcomings of using neural networks for text-to-text translations is the enormous vocabulary that each language has, which requires large datasets and substantial time to train the models. However, as we are using previously extracted *entities* as input and a limited and well-defined language as output, we can overcome the shortcoming above by performing *entities' anonymization* [10]. This pre-processing consists of replacing each extracted *entity* with a token representing it and using the token representation as input for the RNN *encoder*. For example, if the Entities Extractor outputs *"firewall"* and *"backend"*, we would use *anonymization* to convert them to the tokens *'@middlebox'* and *'@target'* before starting the Intent Translation stage. After the translation, we simply run a *deanonymization* on the resulting intent program to replace the tokens with the originally extracted entities. By using anonymization, we can reduce the number of training cases needed for the model considerably, since we do not have to consider every possible *entity* value for network intents. Our preliminary tests showed a size reduction of the training dataset from 1.000.000 to 5.000 with, surprisingly, improved accuracy.

As we cannot use words directly as input for the sequence-to-sequence model, we convert each input word of the model to a unique numerical representation. The numerical representation of the anonymized *entities* are the numeric indices in a pre-built dictionary that contains all words in the model. Equation 1 presents an example of a conversion using a vocabulary with just four words that include the words *middlebox* (index 2) and *target* (index 3).

$$\left['firewall','backend'\right] \Rightarrow \left['@middlebox','@target'\right] \Rightarrow [2,3]$$
(1)

In addition to indexing the words of the input sequences, we perform Word Embedding vectorization in the first layer of the RNN *encoder* to concisely represent the indexed words as arrays of real values. This word vectorization is known

to improve the learning rates and prediction accuracy of linguistic models, as it can capture and represent the meaning of each word [14]. The array of real values, which represents the sequence of anonymized *entities* given as input to the sequence-to-sequence model, is then processed one by one by the RNN *encoder* to generate the thought vector. The RNN *decoder* then uses the encoded thought vector to predict a sequence of statements in the output language *Nile*. The structured intent definition generated by the *decoder* is then presented to the network operator for confirmation on the extracted desired behavior through the chatbot interface.

The operator may either confirm the correctness of the intent program or make adjustments if necessary. After the operator's response, the intent program and the input *entities* are included in the training database of the sequence-to-sequence model, and a new training round is initiated. In this interaction, we explicitly consider the operator's feedback during the translation, ensuring that the results improve every time the operator requests an action.

2.3 Intent Deployment

Finally, having a structured intent program verified by the operator, the Intent Deployer can compile and deploy it into a destination network, as shown in Figure 1. In this stage, we make assertions to verify any conflicts between the extracted intent and the network configuration and warn the operator through the chatbot interface. We then translate *Nile* programs into configuration commands using SONATA-NFV [15]. We currently do not deal with non-SDN networks, but we intend to develop an AI-based module that can handle different networks in future work. For example, a neural network could infer the best routes to comply with SLA requirements of the intent without the need of a pre-populated database. However, the decoupling provided by the intent definition language allows compilations to other existing network configurations, including other policy languages, such as Janus [1], PGA [16], and Kinectic [11].

Ideally, the process described in this section and presented in Figure 1 would also include an Intent Behavior Monitor module. This module would ensure that the deployed policies respect the intents extracted by the refinement process. To achieve this goal, the module could leverage a neural network to predict which parameters should be monitored. The module could then monitor the parameters and notify the operator in case of disparities between the behavior and the intent. We leave the design and implementation of this module for future work.

3 *NILE*: INTENT DEFINITION LANGUAGE

The previous section presented a lengthy process to transform natural language into device configurations. A key insight we uncovered from this translation process is the clear need for a simple, yet comprehensive, abstraction layer between lower-level policies and the natural language used by operators and home users. While low-level policy enforcers, such as SDN rules, require operators with extensive expertise and management experience to program the intended behavior of a network, natural language is hard to parse and interpret correctly and often inaccurate, creating a huge gap between the intended behavior and the network configurations. Also, translating natural language intent directly to network rules decreases portability and reusability, since each possible destination network has specific features and configuration requirements. To bridge this gap, we propose the *Nile* language as an intermediate intent representation that is close to natural language. However, *Nile* exhibits enough structure that works well as the target for the learning algorithm and allows translation to different target networks.

By introducing an intent definition language as an intermediate representation in the refinement process, we decouple the policy extraction from the policy deployment and enforcement. This decoupling, with an intermediate representation that resembles natural language and is easy to understand, allows us to use the feedback from the operator before deploying the extracted behavior. Moreover, the intent definition language acts as an abstraction layer for other policy mechanisms, reducing the need for operators to learn multiple policy languages for each different type of network. Hence, the design requirements for the intent language grammar are: *(i)* high legibility, as operators unfamiliar to the language must be able to understand and assert the correctness of the intent; *(ii)* high expressiveness, to faithfully represent the operator's intention; and *(iii)* high writability, to allow operators to make adjustments to the generated intents quickly and easily. The grammar of *Nile*, in EBNF notation [9], is in Grammar 1.

With the Nile language, we can build powerful yet simple intents. For example, an input *"Add firewall and intrusion detection from gateway to backend for client B, with latency less than 10ms and 100mbps of bandwidth, and allow HTTPS only, everyday from 09:00 to 18:00 "* can be represented as in Listings 1. Note that the Nile program includes only the specific hours defined in the intent, which means that the behavior must be repeated every day. This example illustrates how *Nile* provides a high-level abstraction for structured

---

Nile comes from Network Intent LanguagE

$\langle intent \rangle$	::=	'define intent' intent_name ':' $\langle commands \rangle$
$\langle commands \rangle$	::=	$\langle command \rangle$ { '\n' $\langle command \rangle$ }
$\langle command \rangle$	::=	($\langle middleboxes \rangle$ | $\langle qos \rangle$ | $\langle rules \rangle$)+ [$\langle optional \rangle$]
$\langle middleboxes \rangle$	::=	'add' $\langle middlebox \rangle$ { (',' | ', \n') $\langle middlebox \rangle$ }
$\langle middlebox \rangle$	::=	'middlebox(' middlebox_id ')'
$\langle qos \rangle$	::=	'with' $\langle metrics \rangle$
$\langle metrics \rangle$	::=	$\langle metric \rangle$ { (',' | ', \n') $\langle metric \rangle$}
$\langle metric \rangle$	::=	$\langle metric\_id \rangle$'(' $\langle constraint \rangle$ ',' value ')'
		| $\langle metric\_id \rangle$'(none)'
$\langle metric\_id \rangle$	::=	latency | jitter | loss | throughput
$\langle constraint \rangle$	::=	'less [or equal]' | 'more [or equal]' |
		'equal' | 'different'
$\langle rules \rangle$	::=	$\langle rule \rangle$ { '\n' $\langle rule \rangle$ }
$\langle rule \rangle$	::=	(allow | block) $\langle traffic \rangle$
$\langle optional \rangle$	::=	$\langle targets \rangle$ | $\langle locations \rangle$ | $\langle interval \rangle$
$\langle targets \rangle$	::=	'for' $\langle target \rangle$ { (',' | ', \n') $\langle target \rangle$ }
$\langle target \rangle$	::=	'client(' client_id ')' | $\langle traffic \rangle$
$\langle locations \rangle$	::=	'from' $\langle endpoint \rangle$ 'to' $\langle endpoint \rangle$
$\langle endpoint \rangle$	::=	'endpoint(' endpoint_id ')'
$\langle interval \rangle$	::=	'start' $\langle date\_time \rangle$ '\n' 'end' $\langle date\_time \rangle$
$\langle traffic \rangle$	::=	'traffic(' traffic_id ')' | 'flow(' [$\langle five\_tuple \rangle$]+ ')'
$\langle five\_tuple \rangle$	::=	'protocol:' v | 'src_port:' v | 'src_ip:' v | 'dest_port:' v | 'dest_ip:' v
$\langle date\_time \rangle$	::=	'datetime(' datetime ')' | 'date(' date ')' | 'hour(' hour ')'

Grammar 1: *Nile*.

intents. We believe this initial grammar for *Nile* is expressive enough to represent most network intents, but we do plan to expand it to incorporate new features. Note that the ids provided by the operator (*i.e.*, tokens in red in Grammar 1) must be resolved during the compilation process, as they represent information specific to each network. This feature of the language enhances its flexibility for defining intents and serving as an abstraction layer.

```
define intent qosIntent:
  from    endpoint('gateway')
  to      endpoint('database')
  for     client('B')
  add     middlebox('firewall'), middlebox('ids')
  with    latency('less', '10s'),
          throughput('more or equal', '100mbps')
  allow   traffic('https')
  start   hour('09:00')
  end     hour('18:00')
```

Listing 1: *Nile* intent example.

4 IMPLEMENTATION

We use a different Github project for each stage of the refinement process so that people can download and reuse the stages individually. We implemented the Entities Extractor as a DialogFlow chat interface and deployed it for testing in the Google Assistant. The chat interface consists of a list of entities, which are the key features to be parsed from natural language, and language intents (not related to network intents). Language intents represent possible user interactions that the chatbot creator provides for machine learning training so that DialogFlow can generalize and learn how to extract the necessary entities from future user interactions. We exported our implementation of the chat interface from DialogFlow as JSON files and uploaded them to GitHub. For reproduction purposes, the files can be imported to a new DialogFlow project and retrained.

We implemented the Intent Translator as a Python Restful API service that is called by the DialogFlow chat interface right after it extracts the entities. The service can interact with the chatbot interface to ask for additional information if necessary. Besides this interaction, the API provides a sequence-to-sequence model developed using Keras [3] that we used to train our model. We trained and computed the weights of our model with an automatically generated dataset of input entries containing examples of anonymized entities and the correspondent *Nile* program, which is also anonymized. We used different sizes of datasets in our evaluation, and the results are in §5. After generating a *Nile* intent and confirming it with the user feedback, we retrain the model by adding the intent to the training dataset. The Intent Translator project is available at GitHub. For testing purposes, we deployed the Intent Translator using Heroku [7].

Finally, we developed the Intent Deployer as a separate Python Restful API service that is called by the Intent Translator when it finishes the translation process. As a proof-of-concept, we developed a project that implements service chaining policies using SONATA-NFV [15] and deploys them in an emulated network. SONATA-NFV is an emulation platform based on Mininet [12] that deploys network functions as Docker containers. *Nile* commands for client identification, time, and traffic requirements are not implemented yet. However, we do plan to extend our implementation to include the full set of *Nile* commands and to introduce machine learning to predict the best way to compile and fulfill *Nile* programs in a destination network. This project is also available at Github.

---

5 EVALUATION

To assess the feasibility of our intent refinement process, we evaluate two main aspects: *(i)* the accuracy we can achieve with different sizes of training datasets, aiming to find the optimal ratio between dataset size (which impacts the training time significantly) and prediction accuracy; and *(ii)* the impact of the operator feedback on the accuracy of predictions over time to determine if it improves accuracy. Also, we provide a test case to demonstrate the end-to-end deployment process of intents in a destination network (i.e., from natural language to network configurations). We run our experiments on a server with 8 Intel(R) Core(TM) i7-6700 CPU at 3.40 GHz, 16GB of RAM, running Deepin Linux kernel 4.14. We generated the datasets automatically with random sets of *entities* and *Nile* intent pairs, combining a different number of middleboxes, endpoints, traffic matching rules, time, and QoS requirements in each intent. All training iterations were done with 70 epochs, batch size of 64, and a validation split of 20%. We evaluate five different sizes of training datasets: 100, 500, 1000, 2000, and 5000 entries. For each size of the training dataset, we generate a separate testing dataset, containing 20% of the number of entries from the training dataset.

To assess the first aspect, we first train the translation model and then we measure for each prediction in the testing dataset the correlation coefficient squared (*i.e.*, R-squared) between the intent predicted by the model and the expected output intent. In this case, the closer to 1 the R-squared value is, the more accurate the translation model is—i.e., less errors (*e.g.*, repeated words) and wrong instructions in the resulting *Nile* program. The measurements generated a list of R-squared values for each test case. Figure 3(a) shows the mean and 95% confidence interval for the measured R-squared values for each training dataset size. We also show in Figure 3(b) the training times for the same datasets. As expected, the larger the training dataset, the more accurate the results yielded by the translation model and the longer the training times. We can see from Figure 3(a) that we need only 5000 entries in the training set to achieve excellent results in the refinement process. We expect better results with larger datasets. However, the training process for our largest dataset was close to three hours, and larger datasets require even longer periods for training the model.

Next, we evaluate the impact of the operator feedback in the accuracy of the prediction with the same datasets. To simulate this process, we first load the neural network weights for the trained model. Having a trained model, we use 30 different test cases of *entities* and expected *Nile* intents to simulate requests from an operator. For each case,
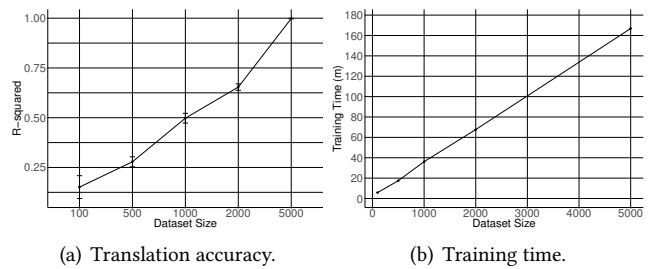

(a) Translation accuracy. (b) Training time.

Figure 3: Accuracy and training time by dataset sizes.

we first use the *entities* to predict an intent from the translation model, and measure the R-squared for that case in comparison to the expected *Nile* intent; then, we add the expected intent into the training dataset of the model, and start a new epoch of training.

Figure 4 shows the R-squared values after 0, 10, 20 and 30 feedbacks were incorporated into the training dataset. It is clear from the plot that, regardless the size of the training dataset, the accuracy improves considerably with repeated training after incorporating feedback. This behavior is particularly evident for training datasets with a smaller number of entries. For instance, we can observe that the operator feedback can improve the accuracy of the model trained with 2000 entries up to the same level as the model trained with 5000 entries without the feedback. This result means that for a much smaller dataset, which requires much less training time, we can achieve similar results. It is also worth mentioning that, in some cases, results obtained with smaller datasets were better than the results obtained with larger datasets, such as with dataset sizes 500 and 1000. This behavior is most likely because of feedback cases that repeated during the test since the training dataset were randomly generated. Hence, the model trained with a smaller dataset could predict with higher accuracy the cases he had already learned from the feedback.

Finally, one could argue that, since the training dataset with 5000 entries has close to perfect accuracy from the start, there is no need for incorporating feedback into the training process. We counter this argument by pointing out that, no matter how accurate the prediction model is, there will always be specific cases that are not covered by the training dataset, and the sequence-to-sequence model may produce erroneous results. Therefore, it is imperative that an operator confirms the generated intent to avoid misconfigurations, and corrects it so that the model can learn and reduce the frequency of these cases.

To illustrate how the end-to-end deployment process of intents works, we present a test case that concerns service
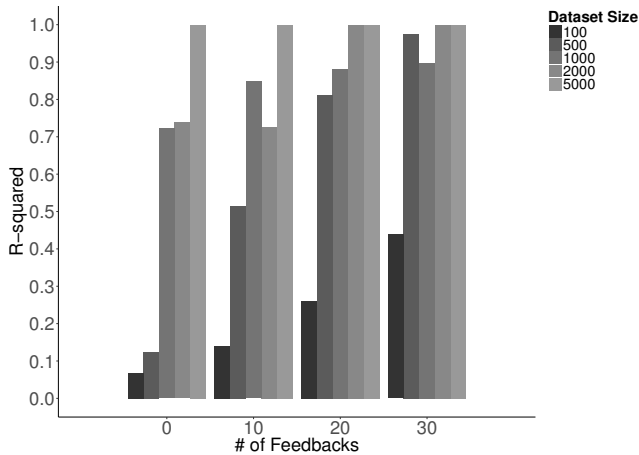
Figure 4: Accuracy improvement with feedback over time.

chaining using SONATA-NFV (see Section 4). All the scripts and reproduction artifacts of this and other test cases that we cannot show because of space limitations are available at Github. The scenario consists of a network with two Open-VSwitches connecting an Iperf client that sends 100 Mbps of UDP traffic to its server and a Stratos Web client that generates HTTP requests to a Web server. After starting both clients, we tested the user intent "Please add a firewall and an IDS from Iperf client to server," aiming to block and inspect the traffic generated from the Iperf client while ignoring the traffic from the Web client. The Entities Extractor, in DialogFlow, extracts the origin, destination and desired middleboxes of the intent, and call the Intent Translator RestAPI. The Intent Translator converts the input entities into the *Nile* intent displayed in Listing 2. Subsequently, the Intent Deployer compiles the translated *Nile* program in the SONATA-NFV commands shown in Listing 3. The middleboxes are Docker containers using pre-configured images (*i.e.*, *genic-vnf*) with the scripts required to run the network functions. We use iptables and Snort to implement the firewall and IDS, respectively. Figure 5 shows the test scenario where the red arrows represent the deployed intent.

```
define intent testIntent:
    from endpoint('iperf client')
    to endpoint('iperf server')
    add middlebox('firewall'),
        middlebox('ids')
```

Listing 2: Generated *Nile* intent.

---

https://github.com/asjacobs92/nia-experiment

```
# deploy vnfs
vim-emu compute start -d vnfs_dc -n fw \
    -i genic-vnf -c "./start_firewall.sh &" \
    --net "(id=in,ip=10.0.0.20/24),(id=out,ip
        =10.0.0.21/24)"
vim-emu compute start -d vnfs_dc -n ids \
    -i genic-vnf -c "./start_snort.sh &" \
    --net "(id=in,ip=10.0.0.30/24),(id=out,ip
        =10.0.0.31/24)"
# chain vnfs
vim-emu network add -b -src iperf-c:c-eth0 -dst fw
    :in
vim-emu network add -b -src fw:out -dst ids:in
vim-emu network add -b -src ids:out -dst iperf-s:s
    -eth0
```
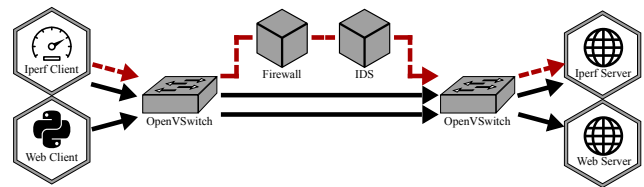
Listing 3: Generated SONATA-NFV commands.

Figure 5: End-to-end test scenario.

6 RELATED WORK

Recent works on IBN feature several intent languages, frameworks, and compilers to efficiently deploy intents in network devices and middleboxes [1, 2, 5, 11, 16, 17, 19, 20]. Most notably, PGA [16] proposes the use of a graph abstraction to compose high-level policies and deploy them in SDN networks. PGA supports Access Control List (ACL) and service-chaining policies, leveraging a graph structure to resolve conflicts. Janus [1] extends PGA to support policies with QoS requirements, mobility, and temporal dynamics. More recently, Cocoon [17] introduces a framework focused on guaranteeing correctness of SDN programs that resembles our approach, but it uses first-order logic instead of machine learning to convert high-level intents into lower level configurations. Cocoon, however, does not validate its refinements with the operator or learn the operator's intent over time. Moreover, its specification language is not as user friendly as natural language. In the industry, Robotron [20] provides a high-level intent abstraction for designing and managing the worldwide-scale network of Facebook. While these efforts present contributions for specifying and verifying network policies, they still fail to extract intent information from pure natural language, requiring that network operators learn new and complex policy definition languages. In our work, we tackle these complexity issues by leveraging the DialogFlow

chatbot interface, coupled with the neural translation process into *Nile*. By relying on *Nile* solely for adjustments and confirmation, we significantly reduce the knowledge curve of our intent solution.

Other related works focus specifically on the intent and policy refinement process. For instance, INSpIRE [18] applies a refinement process to determine which middleboxes should compose a service chain to fulfill an intent. However, this refinement process focuses solely on intents related to security middleboxes, ignoring other essential complex scenarios with other intent requirements. Machado *et al.* [13] and Craven *et al.* propose different approaches to policy refinement, leveraging Event Calculus (EC) [13] or a UML logical representation [4] as intermediate policy representations to allow the operationalization of network behavior definition. Still, these existing research proposals for IBN fail to exploit the knowledge and feedback from the network operator. Highly complex and, sometimes, conflicting policies in network devices may cause network intents to derail from the desired behavior of the operator. Hence, requesting feedback from the operator is crucial to avoid misconfigurations. We tackle these shortcomings by incorporating the operator's feedback into our neural network training dataset so that we can learn from past mistakes.

7 CONCLUSION

In this paper, we introduced a novel intent refinement process and *Nile*, a high-level intent definition language, aiming to be a step towards enabling *self-driving networks*. The proposed refinement process leverages a user-friendly chat interface and a sequence-to-sequence learning model that extracts from natural language a structured intent program, written in *Nile*. The extracted *Nile* intent acts as an abstraction layer for lower-level configuration and policy languages, which allows us to ask for feedback from the operator before compiling the structured intent into network configurations. Our evaluation of the proposed process yielded a correlation coefficient squared (*i.e.*, R-squared) of 0.99 for the intents extracted using our sequence-to-sequence model. Also, the use of feedback from the operator into the model improved the accuracy of our translation model, especially for smaller training datasets.

ACKNOWLEDGMENTS

REFERENCES

[1] A. Abhashkumar, J. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. 2017. Supporting Diverse Dynamic Intent-based Policies Using Janus. In *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '17)*. ACM, New York, NY, USA, 296–309. http://doi.acm.org/10.1145/3143361.3143380

[2] C. J. Anderson, N. Foster, A. Guha, J. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. 2014. NetKAT: Semantic Foundations for Networks. *SIGPLAN Not.* 49, 1 (Jan. 2014), 113–126. http://doi.acm.org/10.1145/2578855.2535862

[3] F. Chollet et al. 2015. Keras. https://github.com/fchollet/keras. (2015).

[4] R. Craven, J. Lobo, E. Lupu, A. Russo, and M. Sloman. 2011. Policy refinement: Decomposition and operationalization for dynamic domains. In *2011 7th International Conference on Network and Service Management.* 1–9. https://ieeexplore.ieee.org/document/6103981

[5] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. 2011. Frenetic: A Network Programming Language. *SIGPLAN Not.* 46, 9 (Sept. 2011), 279–291. http://doi.acm.org/10.1145/2034574.2034812

[6] Google Inc. 2018. Dialogflow. (Mar 2018). https://dialogflow.com/ Available at https://dialogflow.com/.

[7] Salesforce.com Inc. 2018. Heroku. (Mar 2018). https://heroku.com/ Available at https://heroku.com/.

[8] ETSI NFV ISG. 2012. *Network Functions Virtualisation.* White Paper 1. http://portal.etsi.org/NFV/NFV_White_Paper.pdf

[9] ISO/IEC 14977:1996 1996. *Information technology – Syntactic metalanguage – Extended BNF.* Standard. International Organization for Standardization, Geneva, CH. http://standards.iso.org/ittf/PubliclyAvailableStandards/

[10] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. 2017. Learning a Neural Semantic Parser from User Feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers.* 963–973. https://doi.org/10.18653/v1/P17-1089

[11] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. 2015. Kinetic: Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15).* USENIX Association, Berkeley, CA, USA, 59–72. http://dl.acm.org/citation.cfm?id=2789770.2789775

[12] B. Lantz, B. Heller, and N. McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX).* ACM, New York, NY, USA, Article 19, 6 pages. http://doi.acm.org/10.1145/1868447.1868466

[13] C. C. Machado, J. A. Wickboldt, L. Z. Granville, and A. Schaeffer-Filho. 2015. An EC-based formalism for policy refinement in software-defined networking. In *2015 IEEE Symposium on Computers and Communication (ISCC).* 496–501. https://doi.org/10.1109/ISCC.2015.7405563

[14] T. Mikolov, W. Yih, and G. Zweig. 2013. Linguistic Regularities in Continuous Space Word Representations. In *HLT-NAACL.* 746–751. http://www.aclweb.org/anthology/N13-1090

[15] M. Peuster, H. Karl, and S. van Rossem. 2016. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 148–153. https://doi.org/10.1109/NFV-SDN.2016.7919490

[16] C. Prakash, J. Lee, Y. Turner, J. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. 2015. PGA: Using Graphs to Express and Automatically Reconcile Network Policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 29–42. http://doi.acm.org/10.1145/2785956.2787506

[17] L. Ryzhyk, N. Bjørner, M. Canini, J. Jeannin, C. Schlesinger, D. B. Terry, and G. Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 683–698. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryzhyk

[18] E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville. 2017. INSpIRE: Integrated NFV-based Intent Refinement Environment. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 186–194. https://doi.org/10.23919/INM.2017.7987279

[19] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. 2014. Merlin: A Language for Provisioning Network Resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*. ACM, New York, NY, USA, 213–226. http://doi.acm.org/10.1145/2674005.2674989

[20] Y. E. Sung, X. Tie, S. H. Y. Wong, and H. Zeng. 2016. Robotron: Top-down Network Management at Facebook Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 426–439. http://doi.acm.org/10.1145/2934872.2934874

[21] I. Sutskever, O. Vinyals, and Q. V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'14)*. MIT Press, Cambridge, MA, USA, 3104–3112. http://dl.acm.org/citation.cfm?id=2969033.2969173