View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repository of the Academy's Library

A Little Less Interaction, A Little More Action: A Modular Framework for Network Troubleshooting

István Pelle, Felicián Németh and András Gulyás

***Abstract*—Requirements of an ideal network troubleshooting system dictate that it should monitor the whole network at once, feed results to a knowledge-based decision making system and suggest actions to operators or correct the failure, all these automatically. Reality is quite the contrary, though: operators separated in their cubicles try to track down complex networking failures in their own way, which is generally a long sequence of manually edited parallel shell commands calling rudimentary tools. This process requires operators to be "masters of complexity" (which they often are) and continuous interaction. In this paper we aim at narrowing this huge gap between vision and reality by introducing a modular framework capable of *(i)* formalizing troubleshooting processes as the concatenation of executable functions [called troubleshooting graphs (TSGs)], *(ii)* executing these graphs via an interpreter, *(iii)* evaluating and navigating between the outputs of the functions and *(iv)* sharing troubleshooting know-hows in a formalized manner.**

***Index Terms*—Computer networks, troubleshooting, decision support.**

I. INTRODUCTION

TROUBLESHOOTING a communication network was never an easy problem. Finding causes of errors and failures, tracking down misconfigurations in the increasingly complex interconnection networks of heterogeneous networking devices is quite a challenge. What is more, the prevalence of increasingly complex software components, due to the upcoming software defined networks (SDNs), adds distributed software debugging as an additional issue to deal with. To cope with this increasing complexity, the networking research community suggests the use of knowledge-based decision support together with the standard network monitoring and diagnostic tools, and the conversion of troubleshooting into a highly automated process. Reality seems to reside very far away from this vision. Real operators tend to use the most basic diagnostic tools for monitoring the network, and rely on their own brilliance and programming skills when digging out the root causes of errors in an ad-hoc manner from the reports of these tools. Even if this approach works well in practice, it requires extremely skilled operators who can keep in mind all the details of the network under scrutiny and their continuous interaction usually is wasted on rummaging in the logs of the tools used by them.

I. Pelle and F. Németh are with Budapest University of Technology and Economics, Hungary, with HSNLab, Dept. of Telecommunications and Media Informatics. e-mail: {pelle, nemethf}@tmit.bme.hu

A. Gulyás is with Budapest University of Technology and Economics, Hungary, with HSNLab, Dept. of Telecommunications and Media Informatics and with MTA-BME Information Systems Research Group and was supported by the János Bolyai Fellowship of the Hungarian Academy of Sciences. e-mail: gulyas@tmit.bme.hu

The research leading to these results was partly supported by Ericsson and has received funding from the European Union Seventh Framework Programme under grant agreement № 619609.

As we see, the reason for this huge gap between the ideas and reality is threefold. First, there is no usable, implementation oriented formal description of the troubleshooting processes. Second, there is no platform capable of executing formally defined troubleshooting processes while giving prompt and systematic access to the outputs of the used tools. Finally, there is no existing platform that could integrate existing troubleshooting tools and decision support methodologies in a flexible manner. In lack of formalism and integrated execution platform, operators cannot share and reuse each other's troubleshooting know-hows in a structured way, thus knowledge is not accumulated but remains sporadic as operators treat every specific failure in their own ad-hoc way.

Based on these observations, our contribution will be threefold. First, we propose a formalization of troubleshooting processes in the form of troubleshooting graphs (TSGs)—complete with a description language to define them—, which let operators specify the steps of tracking down network failures in a structural manner. Once created, TSGs can make their solutions ready-to-share and re-usable. Second, we propose a modular execution framework capable of running TSGs and offering on demand fast semantic navigation among the outputs of the tools used in the troubleshooting process. Finally, we present a complete prototype system capable of defining, executing and analyzing TSGs.

The rest of our paper is structured as follows: in Section II we give a brief overview on the related work in both literature and practice. Section III lists the principles of our proposed modular troubleshooting framework, followed by the illustration of its operation over an SDN example in Section IV and an example for traditional networks in Section V. Section VI presents the fundamentals of our prototype, *Epoxide*, which is complemented with a complex illustrative case study in Section VII. Finally, we conclude the paper and give directions for future works in Section VIII.

II. STATE OF THE ART IN NETWORK TROUBLESHOOTING

From the great volume of related literature we highlight here the two main constituents of troubleshooting systems. The first is clearly the area of *network monitoring and diagnostic*

tools, of which main purpose is to seek for symptoms of specific failures. The palette is very broad here, ranging from the most basic tools (like ping, traceroute, tcpdump, netstat, nmap [1] or GNU Debugger (GDB)), through monitoring protocols (such as SNMP and RMON [1]), configuration analyzers (e.g. Splat [2]), performance measurement tools (e.g. iperf [1]) and packet analyzers (like Wireshark), to the more complex ones, such as NetFlow, HSA [3], [4] and ATPG [5]. On top of these, SDN specific tools have added a whole new segment targeting the investigation of specific parts of the architecture. Tools such as Anteater [6], OFRewind [7], NetSight [8], VeriFlow [9], NICE [10], SOFT [11], FORTNOX [12] and OFTEN [13] all fill a niche in SDN troubleshooting.

One level up, the output symptoms of these tools can be aggregated and fed into different *automatic reasoning solutions*. The first representatives of these were created as early as the second half of the 1980s [14] targeting the discovery of failures in telecommunication networks. Early on, rule-based methods were used to resolve issues by using *if–then* statements [15]. Later case-based reasoning [16] and model-based [17] methods were developed. The former utilized a collection of previous cases as a basis for failure analysis, while the latter used models of structural and functional behavior to reason about network issues. Fault-symptom graphs [18] and dependency or causality graphs [19], [20] introduced the concept of tracking failures using graphs that created connections between symptoms, detection and root causes. This concept led to the application of Bayesian networks [21], [22] where belief—in the most probable failure root cause—propagation is based on a probabilistic model.

A. What We See in Current Practice

Despite the readily available set of advanced troubleshooting tools and decision support mechanisms, operators seem to use the most rudimentary tools (like ping, traceroute, tcpdump etc.) while they completely rely on their minds as a knowledge-base. For testing this, we conducted an in-house survey querying which type of problems local administrators run into most frequently and what network troubleshooting tools they use most commonly. The results, we found were completely in accordance with those outlined in [23]. Most problems were caused by connectivity issues that arose from a variety of reasons ranging from hardware failures to configuration changes that became necessary due to security issues. Used troubleshooting tools show similarities also: mostly simple task specific tools are utilized, in certain cases combining them in a script to explore typical failures. Network information is usually stored in simple spreadsheets and proprietary monitoring or troubleshooting tools are used only when they have a low cost—or are preferably free. We found that automatic tools are less frequently used and manual troubleshooting dominates problem solving.

To get a deeper sense of the process, imagine the following scenario: an operator wants to monitor certain traffic flows in an SDN network and analyze whether these flows comply with certain criteria. By applying manual troubleshooting using multiple shells, our fictional operator has to connect to different devices—SDN switches and hosts—run software tools to extract traffic data and then filter these to obtain the flows. This process relies on the application of repetitive tasks—login and invocation of specific tools—and analyzing textual data. This process poses four main problems. *(i)* When historical data is needed, the tools cannot be closed, thus they quickly overpopulate the working environment. *(ii)* While the process is extremely flexible—as operators use tools of their choosing in the way and logic they see fit—, processing the data quickly becomes overwhelming and inefficient without computerized help. *(iii)* There is no clear way that the process—the steps to be taken—can be recorded and later reused in a flexible enough manner. *(iv)* The process is unorganized, thus operators' time is spent mostly on filtering and finding the correlation between the different outputs and keeping in mind the mapping between different shells and devices.

III. Design Principles of a Modular Framework for Network Troubleshooting

Instead of proposing a new troubleshooting tool or another decision support mechanism, we suggest here a framework[1] capable of *combining* existing (and future) special-purpose tools and reasoning methodologies in a modular fashion. Our concept builds on the observation that operators combine different troubleshooting tools to find out the root cause of a network issue. In the following sections, we go through the main notions that we use to describe such troubleshooting processes and the fundamentals of our framework capable of executing troubleshooting graphs (TSGs).

A. Nodes: Wrappers Around Troubleshooting Tools

First we define an abstraction that incorporates the basic elements of a troubleshooting process: *nodes* are wrappers around troubleshooting tools or smaller, processing functions. These are considered as black boxes hiding their internal operation from the outside (see Fig. 1). Operators have three types of interfaces for communicating with nodes. On *inputs* they execute operations (e.g. a text stream to process or clock ticks). *Configuration arguments* relay static parameters. Finally, *outputs* relay the exact output of the wrapped tool or provide extra processing before generating results.

Three stages make up the life cycle of a node. Nodes enter their *initialization* stage only once, where environment setup is performed, including resource allocation and initial configuration. At the execution stage, nodes read the data arriving on their inputs and query the wrapped process or function. Analysis or modification on the wrapped tool's output is also performed here. The node is constantly in this stage when it has been initialized but not yet been terminated. Finally, the node reaches the termination stage when it is being stopped. This stage is responsible for clearing up allocated resources and terminating wrapped processes.

[1]Our initial research and implementation of a subset of current framework functionality is discussed in [24].

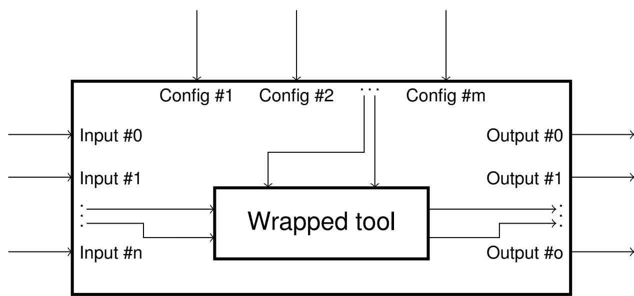


Fig. 1: A conceptual node.

B. Edges: Accessible Data Transfer

We use *edges* to describe the connections between nodes. Besides specifying the nodes to connect, the main feature required from edges is to provide accessibility for node outputs: information over the edges is observable and modifiable on demand by operators. When access is given to edges, operators can analyze troubleshooting processes on the lowest levels. By making historical data available on edges, backtracking network condition changes becomes feasible during runtime. Modifiable edges provide the additional benefit of direct operator interaction with nodes, which is helpful for instant testing purposes. This method also helps channeling—otherwise unobtainable—data into our system from the network environment.

C. The Troubleshooting Graph

By leveraging the power of wrapper nodes and accessible edges, troubleshooting processes can be formalized as TSGs: series of tools and transformations.

Besides the simple concatenation of nodes, creating branches is possible through special purpose *decision* nodes. These nodes are processing nodes capable of analyzing incoming data and matching them against a specific criteria set. Such nodes can provide generalized decision making apparatus that can combine results arriving from different nodes and implement arbitrary decision functions to analyze and evaluate them.

For a text-based representation of TSGs, we define a simple Click-inspired [25] description language. Such language fits perfectly to our concept as we look at nodes as black boxes that have inputs, outputs and configuration arguments, which the language supports by default. Port-based explicit node linking is also a feature that we make good use of. An exemplary TSG and its definition using the language is given in Section IV.

D. Execution Framework for TSGs

In order to bring the TSG concept closer to implementation, we designed our execution framework around three cornerstones. *(i) Interpretation*: since a TSG is only a formal description, the framework provides a parser to interpret the graph. *(ii) Execution*: the TSG concept describes how to connect tools to each other but it does not deal with the problem of when and how a node's life cycle is managed and how a node is notified when its inputs are updated. *(iii) Navigation options*: the benefit of handling interconnected troubleshooting tools as a graph is that it creates a natural order in the troubleshooting process. In order to better manage the complex information set contained in the graph, the framework provides different apparatuses to aid observing the execution state and navigating through the graph.

E. Recommendation System and Knowledge Sharing

The framework provides a recommendation system that is able to suggest new nodes for operators, based on their current setup, by searching for similarities in a TSG repository. Operators can upload their existing TSGs to this repository hereby promoting knowledge sharing.

IV. AN SDN EXAMPLE USING TSGS

In order to solve our running example from Section II-A, instead of manually gathering information about the flows, operators can create a TSG—such as the one shown in Fig. 2—that automates the process for them.[2] They can leverage a unified interface for accessing different OpenFlow capable entities by using nodes to interact with different controller platforms and Open vSwitch (OVS) switches. These nodes are able to collect dapatapath identifiers (DPIDs) and flow statistics information. Additionally, we created processing nodes for filtering flow statistics on a flow space[3] basis.

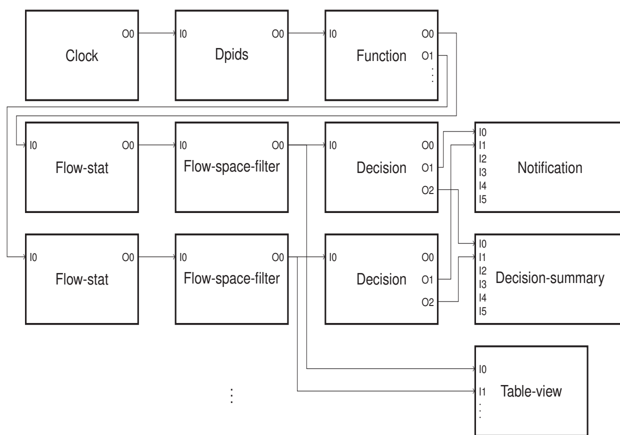
In line 1 of the example of Fig. 2, we define nodes querying DPIDs from the controller[4] on a timely basis by writing expressions that are always terminated by semicolons. Nodes can be defined by assigning an instance name to a wrapper node using the `::` operator and linked with edges to other nodes using the `->` linking operator. For simplicity, the instance name can be omitted if the node is not referenced in the code later on (see the `Dpids` node). Configuration arguments follow the node instance assignment in parentheses. Let's stop now to have a closer look at the `Dpids` node. On its single output, the node relays the DPIDs of the switches connected to a certain SDN controller, each time it receives an enabling signal on its single input. The single configuration argument specifies the location of this controller. In the initialization stage, the node sets up connection to the controller. At execution time, it accesses the controller's specific API to query the DPID information, while at the termination stage, the ongoing processes are stopped and the connection to the controller is closed.

Linking operators always have the list of node outputs on their left side and the list of inputs on their right side—as in line 7 of the example. Multiple edges can also be created between two nodes in a single expression this way. Outputs and inputs are always referred to with their zero-based indices. In case only the 0th input or output appears in a linking expression, one can omit its index, as depicted in line 1. In

[2]With an appropriate node repository, only the connections need to be correctly specified among nodes.

[3]Those flow table entries make up a flow space that match given source and destination point pairs.

[4]Nodes accessing POX, Floodlight and OpenDaylight controllers are currently available.




```
1   Clock(1) -> Dpids(<controller location>)
2   -> f :: Function(...)
3   -> Flow-stat(...)
4   -> filter1 :: Flow-space-filter(...)
5   -> d1 :: Decision(...)[1]
6   -> n :: Notification(...);
7   d1[2] -> [0]summary :: Decision-summary();
8   filter1 -> table :: Table-view()
9   --> view :: View();
10  f[1] -> Flow-stat(...)
11  -> filter2 :: Flow-space-filter(...)
12  -> d2 :: Decision(...)[1] -> [1]n;
13  d2[2] -> [1]summary --> [1]view;
14  filter2 -> [1]table;
15  ...
```

```
emacs@mininet-vm
File Edit Options Buffers Tools Help
               dpid | switch_name | l3_source | l3_destination | table_id |
                    | s1          |  10.0.0.1 | 10.0.0.4       |        0 |
| 00-00-00-00-00-02 |             |  10.0.0.1 | 10.0.0.4/32    |        0 |
U:**-  *node:display:Table-view*   All (1,0)     (Epoxide)
```

Fig. 2: An SDN example: graphical representation of a TSG querying flow statistics (top), its formal definition (middle) and the output of the `Table-view` node displaying the results (bottom). For brevity, the formal definition is shortened.

our language—unlike in Click—one can connect outputs to configuration arguments as well, which enables the flexible dynamic configuration of nodes. This uses the same syntax as the output–input linking.

After querying the list of DPIDs, the `Function` node—that can wrap any preexisting function—splits it and passes the data to further nodes that query flow statistics—by wrapping the `ovs-ofctl` tool—from their assigned switches. With the addition of flow space filtering nodes, the operator can select only the flows under scrutiny and display the results in a table format, as depicted by the bottom part of Fig. 2. By appending `Decision` nodes to the filtering nodes, the operator can make an automatic comparison of the current state of the switches to a predefined criteria set—that might come from a formal definition of the network policy or the controller itself. This could reveal synchronization issues between the devices or misdirection of the traffic caused by

Listing 1: Formal description of the TSG.

```
1   ping :: Ping(localhost, <address of the server>);
2   ifconfig :: Ifconfig(localhost);
3   arp :: Arp(localhost, nil, -n);
4   ping-decision :: Decision(...);
5   ifc-decision :: Decision(...);
6   arp-decision :: Decision(...);
7   ping -> ping-decision;
8   ifconfig -> ifc-decision;
9   arp -> arp-decision;
10  ping-decision[1] -> ifconfig;
11  ifc-decision -> Function(ifconfig-get-interfaces,
12                           'input-0)[0, 0]
13  -> [0, -2]arp;
14  ping-decision[2] -> ds :: Decision-summary();
15  ifc-decision[2] -> [1]ds;
16  arp-decision[2] -> [2]ds;
```

a software failure. The results are then displayed in a table format using a `Decision-summary` node—that interprets decisions by signaling "error" codes with visual aids—and a `Notification` node that provides desktop notifications whenever a test fails. By connecting `Gdb` nodes—wrapping the GNU Debugger—with the `Decision` nodes, the operator could remotely connect to software switches or to the controller for an in-depth analysis of the problem.[5] (Decision nodes are only briefly discussed here, see Section VI-B for more details.)

V. An Example for Traditional Networks

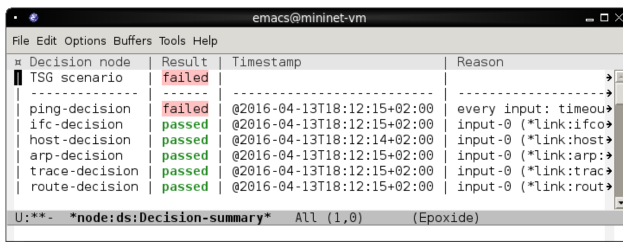
In order to locate errors in a traditional network, a set of currently used free tools can be applied. In a scenario where we want to detect different errors, we can chain these tools together, using TSGs, in such a way that Listing 1 shows. For brevity, we show only the instructions used for creating a TSG that performs a connection test and if that is unsuccessful, checks the local machine's network configuration. This exemplary TSG is not complete (e.g. routes and firewall rules are not checked[6]). We use extended wrapper nodes for the tools, so—on top of their basic functionality—we assume that `Host`, `Ifconfig` and `Arp` nodes provide means to exclude specified interfaces from their outputs, the `Traceroute` node is able to relay the last hop until which the traffic is traceable and the `Route` and `Iptables` nodes are capable of displaying only those rules that apply to certain IP addresses[7]. `Decision` nodes in the TSG evaluate the correctness of given inputs, while the `Decision-summary` node creates a human readable interface for them.

In order to perform a connection test between the current host and a server, we use a wrapper node for the `ping` tool in line 1. Lines 2–3 create tests for checking the local host's configuration using the `ifconfig` and `arp` tools. In order to automatically evaluate these, we defined three Decision

[5] A live action demo on a similar, albeit simpler, case can be watched at [26]: https://www.youtube.com/watch?v=HsiGFR0QirE

[6] Instructions for creating nodes for those tests, however, would be written using the same philosophy shown here.

[7] These assumptions are not unrealistic since wrapper nodes can extend the wrapped tool's functionality in such convenient ways.



Decision node	Result	Timestamp	Reason
TSG scenario	failed		
ping-decision	failed	@2016-04-13T18:12:15+02:00	every input: timeou
ifc-decision	passed	@2016-04-13T18:12:15+02:00	input-0 (*link:ifco
host-decision	passed	@2016-04-13T18:12:14+02:00	input-0 (*link:host
arp-decision	passed	@2016-04-13T18:12:15+02:00	input-0 (*link:arp:
trace-decision	passed	@2016-04-13T18:12:15+02:00	input-0 (*link:trac
route-decision	passed	@2016-04-13T18:12:15+02:00	input-0 (*link:rout

Fig. 3: Summarizing a troubleshooting scenario.

nodes. `ping-decision` checks whether the `ping` was successful. `ifc-decision` uses a custom function to validate the interface configuration returned by the `ifconfig` node. The `arp-decision` node performs a simple check to test whether there are entries in the Address Resolution Protocol (ARP) cache. These Decision nodes are then connected with their respective wrapper nodes in lines 7–9. In order to check the local host's configuration only when the connection test was unsuccessful, we need to connect the `ifconfig` node to the negative output of the `ping-decision` node—line 10 implements that. If there are interfaces on the host that are correctly configured, we need to check whether the host can register the layer 2 addresses from its network. We defined a `Function` node in line 11 in order to retrieve the interface names from the output of the `ifc-decision` and fed these to the `Arp` node. Finally, we defined a `Decision-summary` node in lines 14–16 to display information collected from every Decision node in a summarizing table.

We executed the extended version of this TSG in a home networking environment, where a Linux host was connected to the residential gateway via WiFi connection. A possible output of the `Decision-summary` node is shown in Fig. 3. The node gives the results of the individual decisions in the TSG as well as an overall evaluation of the current troubleshooting scenario: connection with the remote server cannot be established but the configuration of the current host seems to be fine. The most basic assumption at this point can be that the problem is caused by either the residential gateway or it is located at the Internet Service Provider's side.

These simple examples attest that by using TSGs, we can achieve a state of automation where operators can recognize failure modes at a glance by looking at the error codes, or can further delve into details by navigating through the outputs of nodes. Using the navigation options provided by the framework, the operator can walk through the troubleshooting process in an orderly fashion. Results are going to be displayed according to the workflow, laid out when setting up the process of locating the issue.

By offering proper formalization, TSGs—or their subgraphs—can be reused in similar scenarios with slight adjustments to the node configuration arguments. Besides re-usability, TSGs can act as a technique to collect troubleshooting know-hows. Once a network problem is uncovered using a TSG, it automatically becomes a guideline for discovering future similar issues. By collecting a library of these, operators can greatly decrease problem solution times and the efficiency of knowledge transfer to new operators can also increase. If TSGs are not only collected within a closed environment—i.e. within a company—but are shared with a greater audience, they can prove to be beneficial for the whole networking community. If a wide TSG library is paired with problem descriptions and solutions, new nodes and test cases can be recommended to an operator, based on previous cases described by TSGs in the library.

VI. Prototype

For proofing the concept of our framework, we created a prototype implementation, named Epoxide, using GNU Emacs as a platform. Emacs is an extensible, customizable text editor, its central concept, the buffer, is responsible for holding file contents, subprocess outputs, providing configuration interfaces, etc. By its extensible nature, Emacs offered a particularly good platform to build Epoxide upon. Emacs supports, via its advanced text manipulation and documentation functions, writing TSG definitions that we store in `.tsg` configuration files. We implemented our execution framework and wrapper nodes using Emacs's own Lisp dialect, Emacs Lisp. Nodes and their outputs are assigned to Emacs buffers for observability. Node interface and connection information is stored in buffer local variables. We note that while Emacs proved to be a good fit for our notions, the concept described in Section III can be implemented on other platforms as well.

A. Framework Functions

Epoxide provides two methods to create TSGs. The first option is building the graph by interpreting its definition written in our description language. Opening a `.tsg` file in Emacs automatically loads Epoxide and interprets the TSG stored in it. Using this method, graphs can be saved and later reused and shared. Since we added self-documenting capabilities to nodes and leveraged functionality provided by Emacs's *ElDoc* package, the framework offers hints on node interfaces during TSG definition. Syntax highlighting helps to differentiate semantic units at this stage and when Emacs's *Auto Complete* package is installed, it can provide intelligent code completion for setting up nodes. At first, the parser collects the nodes, their configuration arguments and the links between the nodes. The framework assigns unique names to node objects and outputs and maps these to buffers where nodes can write directly during their execution stage. This method can also be used to modify TSGs by way of editing the `.tsg` file and reevaluating it. A downside is that buffer contents prior to the modification get lost.

As a second option, we provide a method for incrementally building TSGs at run time using a drag and drop method. This method has the benefit that operators can monitor the output of the used tools and adjust their methodology to the results they have acquired until that point (which is fully in line with current practice). The parser creates the objects for the nodes and interconnections, and commits these to the `.tsg` file. Run-time modification of the graph—node reconfiguration and relinking—is provided by this functionality as well via an Emacs widget based interface. These two methods can be used

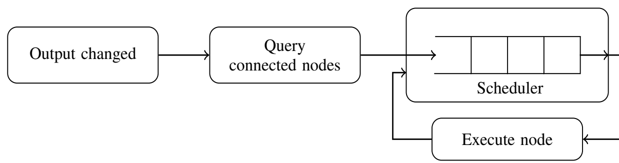


Fig. 4: Scheduling of node execution.

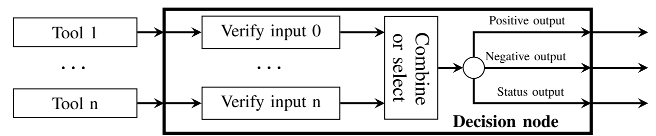


Fig. 5: Schematics of a Decision node.

in conjunction and offer flexibility while retaining the benefit of being able to store troubleshooting scenarios.

Once TSG execution is started, the framework takes care of node events, as depicted in Fig. 4. A scheduler is created—using the *timerfunctions* package—that handles a queue for registering node output changes. Each time an output has changed, the framework queries which nodes have that as their input. These destination nodes are then inserted to the end of the queue. Parallel to this, a simple scheduling mechanism is running that always takes the first item from the beginning of the queue and sends a signal to that node to enter into the execution stage. When the call returns, the scheduler moves to the next node in line and so on.

Basic navigability among the created buffers is provided by Emacs key bindings. Epoxide supplies the apparatus to move from one node buffer to its output buffers or to the next node's buffer (in forward or backward direction). To traverse a TSG, a visualization can be used also, supported by the Emacs *COGRE* package. When the *Graphviz* external software is installed, the displayed graph can be drawn using an automatic layout for better visual clarity. Semantic grouping of the created buffers is provided using the *Ibuffer* package: a dynamic list is displayed that aggregates buffers based on their types and roles in the current context. Custom grouping of buffers is also available via a special node class, a `View`, that can collect nodes or their outputs and display them together.

The current implementation of Epoxide provides a module for collecting TSG related data and supplying node recommendations. We created the instrumentation for basic case-based reasoning where currently available TSGs are considered as descriptions of previous troubleshooting cases. These TSGs are indexed and their data—together with information from the current TSG—is passed to a recommender. The recommender then can suggest nodes based on these pieces of data. We created an interface in the framework to which recommenders, developed by third parties, can connect as well. By now, we have implemented the most basic recommender that suggests the most popular nodes and displays them using Emacs's *Ido* package. Most popular nodes are computed by counting every node in all previously written TSGs and ordering them in the descending order of their cumulative count. Nodes that are already used in the current TSG, are excluded from the suggestion list.

B. Branching Nodes

When creating the apparatus that enables conditional branching in Epoxide, the most basic expectation was to *(i)* provide functionality to analyze and evaluate the output of any node and, based on the result, *(ii)* create branches in a TSG, the same way a decision would in a flow chart. Additionally, we wanted to have the ability to *(iii)* select among different inputs or to use a combination of them. This criterion was inspired by how nodes work in a Bayesian network: they receive the results of lower level nodes and calculate their outputs based on that. To satisfy these criteria, we implemented a single *Decision node*. Since nodes can be added to Epoxide in a modular fashion, this is only one possible implementation that satisfies our initial criteria. Operators are free to add their own version. A *Decision-summary node* was developed in conjunction, for summarizing the outputs of such nodes.

Fig. 5 shows the schematics of our Decision node. The node can attach to any wrapper node[8] and incoming data is first verified (as per *(i)*): compliance with a certain criteria set is determined. The node can use any function for verification that has at least one argument (the input) and can return false, when verification failed, or any string otherwise. When verification of the inputs is finished, further processing can be done using a second stage function, in accordance with *(iii)*. An operator can use a function to select an input with, for example, the *or* function: the first of the inputs that passed verification is going to be the result of this stage. Other functions can implement more complex processing, like in the aforementioned Bayesian network example where the result would be a probability value. Such second stage functions should return a string in case of a positive decision or false otherwise. We defined two node outputs to relay these return values (as per *(ii)*). The first displays information in the positive case, the second in the negative. We implemented an additional output for interoperating with Decision-summary nodes.

C. Implementing Nodes

The modular architecture of Epoxide makes it possible for anyone to implement new nodes. A node developer—who implements a node—has to create separate node definition files that contain Emacs Lisp code to provide self-documentation for nodes and implement the three life cycle stages via functions. For proper interaction with nodes, the node definition files and these functions should comply with a fixed naming scheme. Node self-documentation functions should provide information about node interfaces and could also implement validation functions to check the compliance of arguments with certain criteria. The functions implementing the separate node life cycle stages are called by the framework on specific occasions. The initialization function is evoked after the buffers belonging to the node are set up. The execution

[8]The node is also prepared to handle the asynchronism and the different output formats of the wrapper nodes.

function is called every time a change occurs on any of the node's inputs. Finally, the termination function is invoked before the framework closes the buffers associated with the node. These functions can draw on common node functions provided by the framework. Epoxide implements functions for setting up node buffers with basic data, reading inputs and configuration arguments, writing outputs and handling remote access using Emacs's *Transparent Remote Access, Multiple Protocols (TRAMP)* package.

VII. A CASE STUDY: TROUBLESHOOTING IN SERVICE FUNCTION CHAINING

One of the main goals of the UNIFY project[9] was to design a Service Function Chaining control plane architecture and implement a proof-of-concept prototype. Additionally, the project also introduces the concept of Service-Provider DevOps to combine the developer and operational workflows in carrier grade environments. DevOps results in faster deployment cycle of novel networking services. Instead of designing complex services as a whole, these services are assembled from atomic *network functions*. However, fast deployment cycles require faster testing phases and troubleshooting in the operational environment. Even when a new service is created by re-using components of previous ones, it is still going to be different enough from earlier scenarios to implicate new troubleshooting challenges. In these cases, previous knowledge is not always directly applicable. By providing an integrated platform for running troubleshooting tools and an apparatus to automatize their execution, our tool makes the formation of new troubleshooting scenarios easier, thus enabling quicker service deployment.

Epoxide has a central role in the multipurpose demonstrator showcasing major results of the project [27]. Using its dedicated and general purpose wrapper nodes, it orchestrates multiple components in a semi-automatic troubleshooting scenario. Epoxide needs to reveal a configuration error resulting in erroneous imbalance of the traffic loads of OpenFlow switches instantiated as network functions. The novel components, which Epoxide interacts with, include a flexible messaging bus (Double Decker), a tool that calculates aggregated performance metrics derived from data queried from hierarchical time series databases [Recursive Query Engine (RQE)], and a test packet generator for pinpointing errors in OpenFlow switches (AutoTPG).[10] The use of Epoxide significantly shortens the time spent on analyzing network state, compared to a manual troubleshooting scenario.

Fig. 6 highlights the main system components and the sequence of interactions. First, a monitoring component detects the resource imbalance (this takes a couple of seconds) that automatically triggers the execution of the troubleshooting process in Epoxide which executes a TSG tailored for this scenario (step 1). In step 2, Epoxide asks the Recursive Query Engine to verify the prevalence of the error. After querying historical measurement data—that takes 1-2 seconds in

[9]https://www.fp7-unify.eu

[10]Detailed description of these tools and the demonstrator can be found in [27].

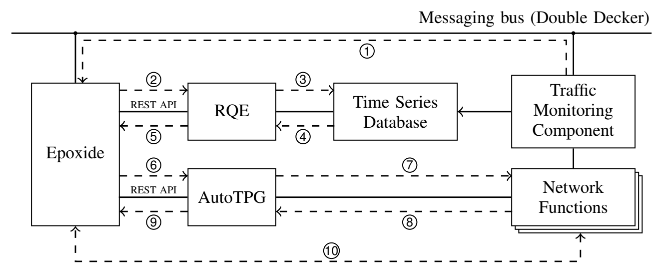


Fig. 6: Simplified view of the UNIFY demo architecture and sequence of interactions.

steps 3–4—RQE notifies Epoxide at step 5. In step 6, Epoxide starts and configures AutoTPG and asks it to test switches one by one. AutoTPG tests the correctness of the switches in steps 7–8 (this test runs in the order of minutes) and returns the result to Epoxide in step 9. Finally, Epoxide queries the flow-entries of the erroneous switch (step 10), and presents those in tabular form to the user to help further investigating the problem manually. Since Epoxide is responsible for calling other tools and performing simple analysis on their outputs, it adds very little reaction time overhead. The complete runtime of the scenario mostly comprises of the runtime of the used tools, mainly that of the AutoTPG.

The demo TSG contains a wrapper node for the messaging bus, but communicates with the other tools via `Rest-api` nodes that use JSON-formatted inputs and emit JSON-formatted outputs. We introduced formatting and JSON filtering nodes in order to assemble and parse these JSON requests and responses. Another general purpose node was also developed that is able to call any command line tool. We had good use of this node when running *ssh* commands to query and modify the configuration of a remote network function.

Epoxide exhibits properties that make it an ideal testing and troubleshooting tool for Service Function Chaining. First, the TSG language is an enabler of fast hypotheses testing and small feedback cycles because it allows connecting existing special purpose troubleshooting tools at an abstract level. Second, the test generation process can be further shortened by simply re-using parts of existing `.tsg` definitions. Finally, complex decision logics can be based on service-specific monitoring and troubleshooting tools by writing simple wrapper nodes around these tools.

VIII. CONCLUSION AND FUTURE WORK

While our modular framework proposed here is capable of flexibly combining various troubleshooting tools for tracking down networking issues, and the TSG concept enables the accumulation and sharing of troubleshooting related knowledge, the current prototype implementation should be extended in many aspects. Of course, a large library of wrapper nodes should be added to incorporate more and more troubleshooting tools. Besides this natural option, we outline here some future directions, the framework could benefit from.

Although present implementation supports the addition of new node recommenders, the currently implemented one pro-

vides only a basic functionality. We consider this a basis to implement better recommenders. Suggestions could be made more relevant by taking the environment of the nodes into consideration and suggesting node configurations as well. The process could further be supported by using a community-based repository of TSGs where TSGs can be analyzed and used for supplying better suggestions.

With the help of an appropriate failure propagation model and formal description of the network policy, we believe, TSGs containing network tests and basic evaluations can be generated with little operator intervention or totally unsupervised. By adding the possibility to create hierarchical TSGs, Epoxide could create more complex tests that are selected and configured automatically depending on results acquired from the network at real time.

REFERENCES

[1] Sloan, J. D., *Network Troubleshooting Tools*. O'Reilly, 8 2001.
[2] Abrahamson, C., Blodgett, M., Kunen, A., Mueller, N., and Parter, D., "Splat: A Network Switch/Port Configuration Management Tool," in *Proceedings of the 17th Conference on Systems Administration (LISA 2003)*, 2003.
[3] Kazemian, P., Chan, M., Zeng, H., Varghese, G., McKeown, N., and Whyte, S., "Real Time Network Policy Checking Using Header Space Analysis," in *NSDI*, 2013, pp. 99–111.
[4] Kazemian, P., Varghese, G., and McKeown, N., "Header Space Analysis: Static Checking for Networks," in *NSDI*, 2012, pp. 113–126.
[5] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N., "Automatic test packet generation," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 241–252.
[6] Mai, H., Khurshid, A., Agarwal, R., Caesar, M., Godfrey, P. B., and King, S. T., "Debugging the Data Plane with Anteater," in *Proceedings of the ACM SIGCOMM 2011 Conference*, ser. SIGCOMM '11. New York, NY, USA: ACM, 2011, pp. 290–301. [Online]. Available: http://doi.acm.org/10.1145/2018436.2018470
[7] Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A. *et al.*, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *USENIX Annual Technical Conference*, 2011.
[8] Handigol, N., Heller, B., Jeyakumar, V., Mazieres, D., and McKeown, N., "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proc. USENIX NSDI*, 2014.
[9] Khurshid, A., Zhou, W., Caesar, M., and Godfrey, P., "Veriflow: verifying network-wide invariants in real time," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 467–472, 2012.
[10] Canini, M., Venzano, D., Peresini, P., Kostic, D., Rexford, J. *et al.*, "A NICE Way to Test OpenFlow Applications," in *NSDI*, vol. 12, 2012, pp. 127–140.
[11] Kuzniar, M., Peresini, P., Canini, M., Venzano, D., and Kostic, D., "A soft way for openflow switch interoperability testing," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 265–276.
[12] Porras, P., Shin, S., Yegneswaran, V., Fong, M., Tyson, M., and Gu, G., "A security enforcement kernel for openflow networks," in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 121–126.
[13] Kuzniar, M., Canini, M., and Kostic, D., "OFTEN testing OpenFlow networks," in *Software Defined Networking (EWSDN), 2012 European Workshop on*. IEEE, 2012, pp. 54–60.
[14] Mathonet, R., Van Cotthem, H., and Vanryckeghem, L., "DANTES An Expert System for Real-Time Network Troubleshooting," in *Proceedings of the 10th International Joint Conference on Artificial Intelligence*, 1987, pp. 527–530.
[15] Hitson, B. L., "Knowledge-Based Monitoring and Control: An Approach to Understanding the Behavior TCP/IP Network Protocols," in *Symposium proceedings on Communications architectures and protocols*, vol. 18, 1988.
[16] Lewis, L., "A Case-Based Reasoning Approach to the Management of Faults in Communications Networks," in *Twelfth Annual Joint Conference of the IEEE Computer and Communications Societies*, 1993.
[17] Jakobson, G. and Weissman, M., "Real-time telecommunication network management: extending event correlation with temporal constraints," in *Proceedings of the fourth international symposium on Integrated network management IV*, 1995, pp. 290–301.
[18] Reali, G. and Monacelli, L., "Definition and performance evaluation of a fault localization technique for an NGN IMS network," in *IEEE Transactions on Network and Service Management*, 2009, p. 6(2):122136.
[19] Lu, J., Dousson, C., Radier, B., and Krief, F., "Towards an Autonomic Network Architecture for Self-healing in Telecommunications Networks," in *Mechanisms for Autonomous Management of Networks and Services*, vol. 6155, 2010, pp. 110–113.
[20] ——, "A Self-diagnosis Algorithm Based on Causal Graphs," in *The Seventh International Conference on Autonomic and Autonomous Systems*, 2011.
[21] Charniak, E., "Bayesian networks without tears: making Bayesian networks more accessible to the probabilistically unsophisticated," in *AI Magazine*, vol. 12, 1991, pp. 50–63.
[22] Khanafer, R., "Automated diagnosis for UMTS networks using Bayesian network approach," in *IEEE Transactions on Vehicular Technology*, 2008, p. 57:24512461.
[23] Zeng, H., Kazemian, P., Varghese, G., and McKeown, N., "A Survey on Network Troubleshooting," 2014.
[24] Pelle, I., Lévai, T., Németh, F., and Gulyás, A., "One tool to rule them all: A modular troubleshooting framework for SDN (and other) networks," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, 2015.
[25] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F., "The Click modular router," *ACM Transactions on Computer Systems (TOCS)*, vol. 18, no. 3, pp. 263–297, 2000.
[26] Lévai, T., Pelle, I., Németh, F., and Gulyás, A., "Epoxide: A modular prototype for sdn troubleshooting," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015, pp. 359–360.
[27] Marchetto, G. and Sisto, R. (editors), "Deliverable D4.3: Updated concept and evaluation results for SP-DevOps," in *eprint arXiv:1610.02387*, 2016.

István Pelle received M.Sc. degree in Computer Engineering at Budapest University of Technology and Economics, Budapest, Hungary in 2015. Currently he is pursuing the Ph.D. degree with Budapest University of Technology and Economics. His research interests include software defined networks and network troubleshooting.

Felicián Németh received his M.Sc. degree in Computer Science from BME in 2000. He is a research fellow at the Department of Telecommunications and Media Informatics of the same university. He was a member of national research projects and the EFIPSANS, OPENLAB, UNIFY FP7 EU projects.
His current research interests focus on Software Defined Networking, congestion control methods and autonomic computing.

András Gulyás received M.Sc. and Ph.D. degree in Informatics at Budapest University of Technology and Economics, Budapest, Hungary in 2002 and 2008 respectively. Currently he is a research fellow at the Department of Telecommunications and Media Informatics. His research interests are complex and self-organizing networks and software defined networking.