Dynamic link network emulation and validation of execution datasets

Erick Petersen, Jorge López, Natalia Kushik, Maxime Labonne, Claude Poletti, Djamal Zeghlache

Dynamic link Network Emulation and Validation of Execution Datasets

Erick Petersen[1,2], Jorge López[1], Natalia Kushik[2], Maxime Labonne[1], Claude Poletti[1], and Djamal Zeghlache[2]

[1] Airbus Defence and Space, Issy-Les-Moulineaux, France
[2] Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France
{erick.petersen, jorge.lopez-c, maxime.labonne, claude.poletti}@airbus.com,
{erick_petersen, natalia.kushik, djamal.zeghlache}@telecom-sudparis.eu

Abstract. We present a network emulator for *dynamic link networks*, i.e., networks whose parameter values vary; for example, satellite communication networks where bandwidth capacity varies. We describe the design of the emulator, which allows *replicating* any network system, through the use of state-of-the-art virtualization technologies. This paper is also devoted to the verification of the datasets produced by monitoring the network emulation. We propose a model-based design for a dynamic link network emulator and discuss how to extract data for network parameters such as bandwidth, delay, etc. These data can be verified to ensure a number of desired properties. The main goal is to try to guarantee that the emulator behaves as the real physical system. We rely on model checking strategies for the dataset validation, in particular, we utilize a Satisfiability Module Theories (SMT) solver. The properties to check can include one or several network parameter values and can contain dependencies between various network instances. Experimental results showcase the pertinence of our emulator and proposed approach.

Keywords: Model-based Design, Dynamic link Networks, Emulator, Many-sorted First Order Logic, Satisfiability Modulo Theories

1 Introduction

As the demand for interactive services, multimedia and network capabilities grows in modern networks, novel software and/or hardware components should be incorporated [6]. As a consequence, the evaluation and validation process of these newly developed solutions is critical to determine whether they perform well, are reliable (and robust) before their final deployment in a real network [1]. However, thorough testing or qualifying [29] the produced software under a wide variety of network characteristics and conditions is a challenging task [9].

Currently, many of these tests are done through operational, controlled, and small-scale networks (physical testbeds) or alternatively software-based testbeds. Ideally, if available, such tests are performed on the original system in order to

replicate the conditions in which a service or protocol will be used at the highest level of fidelity [12]. Unfortunately, while system modeling is not needed, such testbeds are not always desirable or pertinent due to several reasons [33]. For example, there are difficulties in creating various network topologies, generating different traffic scenarios, and testing the implementations under specific conditions (network load or weather conditions that may affect the radio-physical links in specific network technologies such as wireless or satellite communications).

A very well-known alternative method is the use of network simulators [15]. Through network simulation, researchers can mimic the basic functions of network devices and study specific network-related issues on a single computer or high-end server. However, the adequacy of simulated systems is always in question due to the model abstraction and simplification. At the same time, not simulation but emulation for the related networks can also be a solution [17]. Network emulation provides the necessary mechanisms to reproduce the behavior of real networks at low infrastructure costs compared to physical testbeds. Emulation is also capable of achieving better realism than simulations since it allows interacting with interfaces, protocol stacks, and operating systems. Moreover, it is possible to perform continuous testing on the final implementation without having to make any changes in the solution once it is deployed in a real network. However, the emulation of dynamic link networks, i.e., networks whose link parameters change, complicates the emulation architecture. For example, certain radio-frequency links have different up/down bandwidth capacity [9], large delays (due to distant transmitters), and the links' capacities may change due to external interference, propagation conditions (weather), traffic variations (due to the shared medium), or others. Therefore, it is extremely important to have methods that allow controlling key parts of the emulation over time, such as the generation of traffic or the modification of the link property values (capacity, delay). These are required in order to build a proper emulation environment of interest which is the main focus of this work. To cope with such requirements, we herein propose a dynamic link network emulation and traffic generation which combines the functional realism and scalability of virtualization and link emulation to create virtual networks that are fast, customizable and portable.

At the same time, the tool needs to ensure the emulation of dynamic link networks exactly as expected and requested by an end user. There are a number of ways to provide such kind of assurance. On the one hand, the emulator can be permanently monitored and the captured data can be analyzed online to check if they satisfy the necessary properties. We investigated this approach previously and the interested Reader can find more details in [24]. With such an online approach, verifying complex properties can be problematic, as the verification time must be reasonable. On the other hand, it is also possible to analyze the monitored data offline. If the generated dataset does not hold expected properties, the emulator should be updated accordingly. The latter strategy is applied in the paper.

We therefore present a design and architecture of the solution that meets dynamic link emulation needs by the effective use of software technologies, such as virtualization (containers and virtual machines) and Linux kernel capabilities. Furthermore, our dynamic link network emulator provides a fast and user-friendly workflow, from the installation to the configuration of scenarios by using a formal model of the network.

We note that the emulator architecture was first presented in [25]. In this paper, we extend the previous results through the validation of the data generated by the emulator in question. For the latter, we rely on the methodology we proposed in [20]. This paper thus extends the two conference papers presented in ENASE'2022 and IC3K'2021, respectively. The main contributions of the current extension are the following: i) monitoring and data extraction within the dynamic link network emulator; ii) creation of a large list of interesting network properties that should be verified, and finally iii) experimental results on the dynamic link dataset verification.

The structure of the paper is as follows. Section 2 describes existing network emulators and simulators and their capabilities. Section 3 details the background and concepts upon which this work is based on. Section 4 presents the dynamic link emulator under design. Section 5 describes the monitoring and data collection possibilities of the emulator, which is further used in the experimental study. The dataset verification approach is presented in Section 6 while the experimental results are summarized in Section 7. Section 8 concludes the paper.

2 Related work

Several works have been devoted to the simulation and emulation of different network types, to perform experiments on novel or existing protocols and algorithms. Below, we briefly summarise some relevant existing solutions.

Ns-3 [11] is a widely used network simulator. Ns-3 simulates network devices by compiling and linking C++ modules while providing data monitoring, collection, and processing capabilities through the Data Collection Framework (DCF). Thus, it simulates the behavior of components in a user-level executable program. However, real-world network devices are highly complex (functionally speaking) or cannot be compiled and linked together with Ns-3 to form a single executable program. Therefore, Ns-3 cannot run real-world network devices but only specific ones developed for it.

Emulab [31] is a network testbed with a minimum degree of virtualization, aiming to provide application transparency and to exploit the hierarchy found in real computer networks for studying networked and distributed systems. Its architecture uses FreeBSD jail namespaces [13] to emulate virtual topologies. Monitoring and data collection can be achieved by running software on each node using a combination of .ns scripts (python for the latest version) and the Emulab web interface. Similarly, Mininet [18] enables rapid testbeds by using several virtualization features, such as virtual ethernet pairs and processes in Linux container network namespaces. It emulates hosts, switches and controllers,

which are simple shell processes that are given their network namespace and links between them. Python is used to implement all the essential functions for the monitoring and data collection process. However, both still present some limitations, including the lack of support for dynamic features such as link emulation, resource management and traffic generation.

EstiNet [35] is based on network simulation/emulation integration for different kinds of networks. Unlike previous simulators, EstiNet allows not only monitoring but also configuration and data collection through a GUI. It also supports wireless channel modeling. However, since EstiNet is a commercial solution, it cannot be easily extended. Moreover, its features are limited and depend on the EstiNet developers. Thus, the performance fidelity and the expansion to new features are reduced. OpenNet [4] also merges simulation and emulation network capabilities by connecting Mininet and Ns-3. As a result, monitoring and data collection can also be achieved through the implementation of python functions within each node. However, it also inherits the limitations of Ns-3 and Mininet. Additionally, its main focus is software-defined wireless local area networks (SD-WLAN).

More recently, the introduction of lightweight virtualization technologies (e.g., containers) has led to some few container-based emulation tools [30], [8], [27]. SDN Owl [30] is a network emulation tool to create simple SDN testbeds using few computers with Linux OSs. SDN Owl utilizes Ansible to send a set of scripts to configure each virtual component properly and can also be used to set up monitoring and data collection tasks. However, it fails to provide scalability and isolation since experiments with different types of network topologies or resource allocation are not shown. vSDNEmul [8] and ContainerNet [27] are network emulators based on Docker container virtualization, allowing autonomous and flexible creation of independent network elements, resulting in more realistic

NAME	OS	LN	GUI	EM	SC	PO	DL	AT	FD	MD	DV
Ns-3 [11]	✓	C++/Python	x	x	+++	x	✓	x	x	✓	x
Mininet [14]	✓	Python	✓	✓	+	✓	x	x	x	✓	x
Containernet [27]	✓	Python	✓	✓	++	✓	x	x	x	✓	x
OMNet++ [34]	x	C++	✓	x	+++	x	✓	x	x	✓	x
Emulab [31]	✓	C	✓	✓	+	✓	x	x	x	✓	x
OpenNet [4]	✓	C++	✓	x	+++	x	✓	x	x	✓	x
vSDNEmul [8]	✓	Python	x	✓	++	✓	x	x	x	x	x
EstiNet [35]	x	-	✓	✓	++	✓	✓	-	-	✓	x
SDN Owl [30]	✓	Python	x	✓	++	✓	x	x	x	✓	x
NetEM [10]	✓	-	x	✓	-	x	✓	x	x	x	x

Abbreviations: OS, Open Source; LN, Language; GUI, Graphical User Interface; EM, Emulation Support; SC, Scalability; PO, Portability; DL, Dynamic Links; AT, Automatic Traffic Generation; FD, Formal Description; MD, Monitoring and Data collection; DV, Dataset Verification

Table 1. Comparison of Software-based Network Testbeds

emulations. Similar to other container-based tools, data collection and monitoring tasks can be performed using the statistics collected by Docker or through services implemented in any programming language that collects the information within each container. However, these emulators can only create SDN networks. Furthermore, the network descriptions remain rather informal and do not facilitate the emulator's verification to guarantee that it properly replicates the desired network.

A feature comparison is shown in Table 1. We therefore are not aware of any work that meets all the required features to properly emulate dynamic link networks in order to qualify novel engineered solutions. Moreover, none provided a formal approach to verify that the extracted dataset (collected data from the emulator) holds the expected properties to ensure that the produced network behaves exactly as requested by an end user.

3 Background

3.1 Dynamic link networks

Examples of networks range from different types of connections or collaboration between individuals (social networks [28]), products (distribution networks [7]), computers (internet [22]) to software networks [19] where edges may represent function calls. In this work, we focus on computer networks, i.e., a set of interconnected computing devices that can exchange data and share resources with each other through links.

Computer networks [26] tend to be classified into many different types in terms of size, distance, structure, connection type or even their function. Local Area Networks (LAN) are perhaps one of the most frequently used and straightforward examples (under normal operation) of what we refer to as a *static network*, i.e., a computer network with static link parameters. A LAN consists of an interconnected group of computer devices, through a common communication path, within a single limited area. In this scenario, network adapters are typically configured (by default) to automatically negotiate the maximum transfer speed with the device they are connected to. Usually, for LAN links, those values are 1Gbps, 100 Mbps or 10 Mbps for up/down connections (full/half duplex modes), i.e., the whole LAN network operates at those constant values.

Unlike those networks, we focus on what we refer to as *dynamic link networks*, i.e., a computer network where the link parameters may change at different time instances. One example of such networks is a satellite communication network [16]. For instance, Geostationary Orbit (GEO) satellite communication systems usually have a high end-to-end latency (at normal operation) of at least 250 ms. Depending on several internal or external factors, it may vary as high as 400 ms, which has a great impact on the speed of their communication links. Additionally, medium conditions may directly have an impact on the bandwidth capacity of the link. However, we consider that the network topology does not change (as for example in ad-hoc networks) in *dynamic link networks*.

Modeling dynamic link networks We view a *static* network as a computer network where each link has a set of parameters that do not change, for example bandwidth (capacity) or delay. Differently from static networks, the parameters of the links may change in dynamic link networks; such change can be the consequence of the physical medium (e.g., in wireless / radio frequency networks) or due to logical changes (e.g., rate limiting the capacity of a given link). Therefore, static networks can be modeled as (directed) weighted graphs $(V, E, p_1, \ldots, p_k)$, where V is a set of nodes, $E \subseteq V \times V$ is a set of directed edges, and p_i is a link parameter function $p_i : E \to \mathbb{N}$, for $i \in \{1, \ldots, k\}$; without loss of generality we assume that the parameter functions map to non-negative integers (denoted by $\mathbb{N}$) or related values can be encoded with them. Similarly, dynamic link networks can be modeled as such graphs, however, p_i maps an edge to a non-empty set of integer values, i.e., $p_i : E \to 2^{\mathbb{N}} \setminus \emptyset$, where $2^{\mathbb{N}}$ denotes the power-set of $\mathbb{N}$, and represents all the possible values p_i can have. As an example, consider the dynamic link network depicted in Fig. 1, and its model $\mathcal{N} = (V, E, p_1(e), p_2(e))$, where:

$$V = \{1, 2, 3, 4\}$$

$$E = \{(1,2), (2,1), (1,3), (3,1), (1,4), (4,1), (2,4), (4,2), (3,4), (4,3)\}$$

$$p_1(e) = b((s,d)) = \begin{cases} \{4,5,6\}, & \text{if } d = 2 \\ \{2,3,4\}, & \text{otherwise} \end{cases}$$

$$p_2(e) = d((s,d)) = \begin{cases} \{1,2,3\}, & \text{if } d = 2 \\ \{9,10,1\}, & \text{otherwise} \end{cases}$$

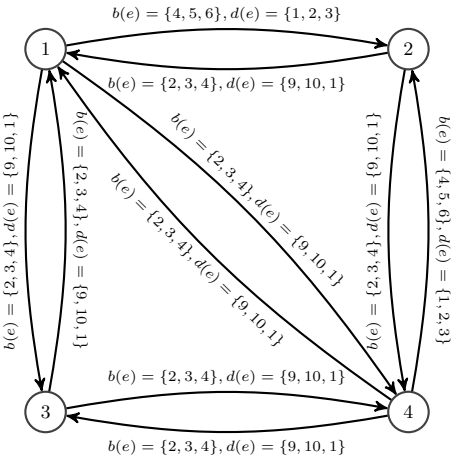


Fig. 1. Example dynamic network [25]

Semantically, this model represents a dynamic link network in which the link's available bandwidth can vary according to the function b (for *bandwidth*), and the link's delay can vary according to the function d (for *delay*). Note that a dynamic link network snapshot, at a given time instance, is a static network, and thus, both terms can be used interchangeably.

3.2 Satisfiability Modulo Theories based verification

In this subsection, we briefly describe some basic notions of SMT [3] (mostly the syntax) and how it can be used for formal verification.

SMT model, syntax, and semantics A *signature* is a tuple $\Sigma = (S, C, F, P)$, where S is a non-empty and finite set of sorts, C is a countable set of constant symbols whose sorts belong to S, F and P are countable sets of function and predicate symbols correspondingly whose arities are constructed using sorts that belong to S. Predicates and functions have an associated arity in the form $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \to \sigma$, where $n \geq 1$ and $\sigma_1, \sigma_2, \ldots, \sigma_n, \sigma \in S$.

A *Σ-term* of sort σ is either each variable x of sort (type) σ, where $\sigma \in S$, or each constant c of sort (type) σ, where $\sigma \in S$; and $f \in F$ with arity $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n \to \sigma$, is a term of sort σ, thus, for $f(t_1, \ldots, t_n)$, t_i (for $i \in \{1, \ldots, n\}$) is a Σ-term of sort σ_i.

A *Σ-atom* (Σ-atomic formula) is an expression in the form $s = t$ or $p(t_1, t_2, \ldots, t_n)$, where $=$ denotes the equality symbol, s and t are Σ-terms of the same sort, $t_1, t_2, \ldots, t_n$ are Σ-terms of sort $\sigma_1, \sigma_2, \ldots, \sigma_n \in S$, respectively, and p is a predicate of arity $\sigma_1 \times \sigma_2 \times \ldots \times \sigma_n$.

A *Σ-formula* is one of the following: (i) a Σ-atom; (ii) if ϕ is a Σ-formula, $\neg\phi$ is a Σ-formula, where $\neg$ denotes negation; (iii) if both ϕ, ψ are Σ-formulas, then, $\phi \wedge \psi$ and $\phi \vee \psi$ are Σ-formulas (likewise, the short notations $\phi \to \psi$ and $\phi \leftrightarrow \psi$ for $\neg\phi \vee \psi$ and $(\phi \wedge \psi) \vee (\neg\phi \wedge \neg\psi)$); finally, (iv) if ϕ is a Σ-formula and x is a variable of sort σ, then, $\exists x \in \sigma\ \phi$ ($x \in \sigma$ is used to indicate that x has the sort σ) is a Σ-formula (likewise, the short notation $\forall x \in \sigma\ \phi$ for $\neg\exists x \in \sigma\ \neg\phi$), where $\exists$ denotes the existential quantifier and $\forall$ denotes the universal quantifier, as usual.

We leave out the formal semantics of MSFOL formulas, their interpretations and satisfiability as we feel it can unnecessarily load the paper with unused formalism. However, we briefly discuss some aspects of MSFOL formula satisfiability. For some signatures, there exist decision procedures, which help to determine if a given formula is satisfiable. For example, consider the signature with a single sort $\mathbb{R}$, all rational number constants, functions $+, -, *$ and the predicate symbol $\leq$; SMT will interpret the constants, symbols and predicates as in the usual real ($\mathbb{R}$) arithmetic sense. The satisfiability of Σ-formulas for this theory (real arithmetic) is decidable, even for formulas with quantifiers [3, 21], i.e., for some infinite domain theories, there exist procedures[3] to decide if a given quantified formula is satisfiable. Therefore, the satisfiability for formulas as $\exists n \in \mathbb{R}\ \forall x \in \mathbb{R}\ x + n = x$ can be automatically determined (via a computer program implementing the decision procedure, i.e., an SMT solver). If a formula is satisfiable, there exists an interpretation (or model) for the formula, i.e., a set of concrete values for the variables, predicates and functions of the formula that makes this formula evaluate to TRUE.

[3] Often such procedures seek to "eliminate" the quantifiers and obtain an equivalent quantifier-free formula

Throughout this paper, we use the previously described syntax for the properties of interest (formulas). Note that for the experimental part, we use the z3 solver [5]. This solver uses the SMT-LIB language, which possesses a syntax that is very close to the described formalism (we do not detail it in this paper, however, the interested Reader may refer to [2]). For example, the formula $\exists n \in \mathbb{R}\, \forall x \in \mathbb{R}\, x + n = x$ can be expressed in SMT-LIB as follows:

```
(exists ( (n Real) ) (forall ( (x Real) ) (= (x + n) x) ) )
```

Listing 3. Example SMT-LIB code

4 Dynamic link network emulator

We hereafter present the first contribution of the work, namely, the emulation platform design and architecture. It is based on well-known state-of-the-art technologies, such as virtualization (VMs and containers) and Linux kernel features (namespaces or cgroups). When combined efficiently, these technologies provide excellent capabilities for emulating a diverse set of network topologies alongside dynamic links and interconnected network devices. The emulation platform architecture [25], shown in Figure 2, consists of several independent, flexible and configurable components. We describe each of these components in detail in the following paragraphs.

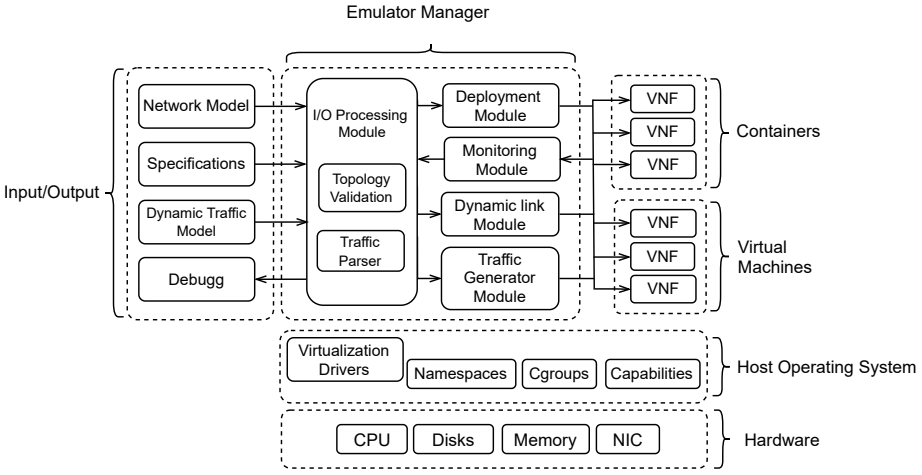


Fig. 2. Emulation Platform Architecture [25]

The **Emulator Manager** is the main component and the central processing unit. It has a single instance per physical machine and is composed of several

independent modules in charge of the management, deployment and verification of the emulator components for a given network description (input for the emulation). In addition, it is responsible for providing, within the same physical host, the containers or virtual machines required for each emulated device as well as their own emulated network specifications.

The **Input/Output Processing Module** fulfills several tasks. First, since our emulation platform relies on state-of-the-art virtualization (or container-based) solutions, it is in charge of creating and maintaining a network model that is later used by other modules to implement the necessary infrastructure elements for each emulation. To achieve this, we utilize a formal network description (specification) in terms of first-order logic formulas verified throughout the emulation by an SMT solver. The interested Reader can find more details in [24]. Indeed, the network topology can be verified using model checking strategies before its actual implementation, as well as at run-time, to ensure that certain properties of interest hold for the static network instances. Finally, the module is also in charge of parsing and verifying the file to generate dynamic traffic scenarios between the components of an emulated network as well as the debugging output of the platform. An example of a network description is given in Listing 4 [25] (for the network in Figure 1).

```
(declare-datatypes() ((Edge (mk-edge (src Int) (dst Int)))))
(declare-fun bandwidth (Edge) Int)
(declare-fun delay (Edge) Int)
;; Node storage omitted on purpose to reduce the space,
;; see edge storage
(declare-const edges (Array Int Edge))
(declare-const edges_size Int)
(assert (= (store edges 1 (mk-edge 1 2)) edges))
;; Edge storage omitted on purpose to reduce the space,
;; see first and last edge
(assert (= (store edges 10 (mk-edge 4 2)) edges))
(assert (= edges_size 10))
(assert
  (forall ((x Int))
    (=>
      (and (> x 0) (<= x edges_size))
      (and
        (=> (= (dst (select edges x)) 2)
        ;; ite not used on purpose
            (and
                (>= (bandwidth (select edges x)) 4)
                (<= (bandwidth (select edges x)) 6)
                (>= (delay (select edges x)) 1)
                (<= (delay (select edges x)) 2)
            )
        )
        (=> (not (= (dst (select edges x)) 2))
            (and
                (>= (bandwidth (select edges x)) 2)
                (<= (bandwidth (select edges x)) 4)
                (>= (delay (select edges x)) 9)
                (<= (delay (select edges x)) 10)
            )
        ) ) ) ) ) ;; closing parentheses
```

Listing 4. Example of a Network Model description (SMT-LIB) [25]

The **Deployment Module** is in charge of converting the previously generated network model into running instances of emulated network devices. In order to achieve this, the module makes use of the Pod Manager tool (`podman`) for the management and support of containers and `libvirt` for different virtualization technologies such as `KVM`, `VMware`, `LXC`, and `virtualbox`. The first step takes the input specification and creates the required nodes with their corresponding images and properties. Each emulated node is deployed by means of a VM or a container attached to its own namespace and acts according to the software or service running inside of it (as requested by the input specification). For example, if it is desired to run a virtual switch as a container, the Deployment Module creates the proper container and executes the corresponding Virtual Network Function (VNF) via a container image (e.g., Open vSwitch). Therefore, each node has an independent view of the system resources such as process IDs, user names, file systems and network interfaces while still running on the same hardware. It can also hold several individual (virtual) network interfaces, along with its associated data, including ARP caches, routing tables and independent TCP/IP stack functions. This gives excellent flexibility and capabilities to the emulator: it can execute any real software, just as real physical systems.

In the last step, the module creates the links between the nodes to complete the emulation topology. The links are emulated with Linux virtual networking devices; `TUN/TAP` devices are used to provide packet reception and transmission for user space processes (applications or services) running inside each node. They can be seen as simple Point-to-Point or Ethernet devices, which, instead of receiving (and transmitting, correspondingly) packets from a physical medium, read (and write, correspondingly) them from a user space process. `veth` (virtual Ethernet) devices are used for combining the network facilities of the Linux kernel to connect different virtual networking components together. `veth` are built as pairs of connected virtual Ethernet interfaces and can be thought of as a virtual "patch" cable. Thus, packets transmitted on one device in the pair are immediately received on the other device. When either device is down, the link state of the pair is down too.

The **Dynamic Link Module** is in charge of establishing and modifying the dynamic properties of the links (between the nodes) during the emulation's execution time. An asymmetric link between two nodes, as shown in Figure 3 [25], is emulated by a set of nesting queues; in the simplest case - two queues.

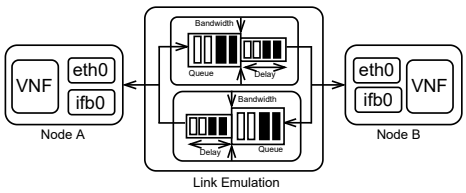


Fig. 3. Asymmetric Link emulation model [25]

In the first step, packets are queued or dropped depending on the size of the first queue. This queue is drained at a rate corresponding to the link's bandwidth. Once outside, packets are staged in a delay line for a specific time (propagation delay of the link) in the second queue and then finally injected into the network stack. This module uses the Linux Advanced traffic control `tc`, to control and set these properties by using filtering rules (classes) to map data (at the data link or the network layer) to queuing disciplines (`qdisc`) in an egress network interface. Note that since `tc` can be used only on egress, traffic Intermediate Functional Block devices (IFB) are created to allow queuing disciplines on the incoming traffic and thus use the same technique.

The **Traffic Generation Module** is in charge of converting the dynamic description of the traffic (see an example of it in Listing 2) into a timed sequence of network packets. This sequence is then introduced into the deployed nodes during the emulation. For the generation of network packets, the module uses `nmap` at each node, particularly `nping`, allowing to generate traffic with headers from different protocols. This is achieved using `virsh` commands using `libvirt` for virtual machines or by passing `execute` commands through the podman tool (for containers). It is important to note that `nping` can be replaced for any other software to generate traffic. Additionally, multiple instances of the same or different traffic generators can be executed inside each emulated node. Finally, the **Monitoring Module** retrieves and collects information from the nodes and their links (see Section 5 for more details). This information is used, for example, to verify that the emulation process is executed correctly (see Section 7).

5 Monitoring and Dataset Extraction

Monitoring is always an important element in the management of any type of network. Indeed, it can be used for planning, resource provisioning, anomaly detection, or simply to ensure their proper operation. In contrast to real networks, monitoring in simulated/emulated environments is generally only used to assess the performance of solutions under test. However, it does not consider the monitoring of the network emulator tool itself, which may also present problems. For example, bugs in the emulation framework or few available resources could lead to results that may diverge from a real scenario. With this objective, we implement a monitoring module, see Figure 4, which not only retrieves and collects information from each emulated node and its links but, also extracts a dataset to perform further verification or to be used by other modules, e.g., a learning module.

The Monitoring module, Figure 4, is organized by a control manager (CM) and several Points of Observation (POs) installed inside each emulated node to inspect its traffic and link dynamics. Each PO is responsible for generating and listening packet events, reporting those events to the CM and collecting its own parameters. The control manager (CM) has a global view of the emulated network and is responsible for coordinating the POs, collecting, computing and keeping the relevant parameters and data structures for the dataset extraction

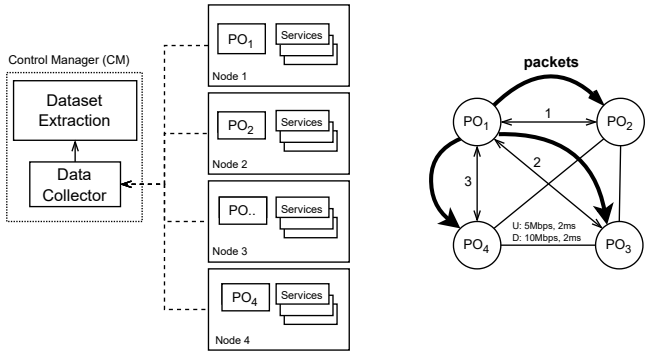


Fig. 4. Monitoring module and dataset extraction

in memory. For example, in order to monitor and extract parameters on links 1, 2 and 3 points of Observation PO_1, PO_2,PO_3,PO_4 are installed in each node. Then, PO_1 initiates packet events towards each neighboring PO (bold arrows), computes and reports to the CM its local measurements. Finally, the CM collects and stores that information and extracts the desired dataset.

Multiple link parameters are measured and added to our dataset, particularly delays, bandwidth and packet loss. Delays are measured using packet exchanges between the source and destination POs, see Figure 5; this measurement is done using the `ping` tool. For a single measure, a request packet is sent from a source to the destination node that answers with a response packet. This packet is identified by modifying the header type field. Particularly, the ICMP protocol uses the headers 8 (for request) and 0 (for reply). The reception of the response packet ensures that the communication was successful at the destination. It also allows us to extract parameters of interest, such as the round trip time (RTT) delay, i.e., the time it takes for a packet to reach its destination from a source and then reach back from destination to source. In addition, other delay types can be measured, including: (a) Transmission delay: the time taken to transmit a packet from source to the transmission medium; (b) Propagation delay: the time taken by the packet to reach the destination through the medium; (c) Queueing delay: the time a packet waits in queue, also called buffer time, before being processed by destination; (d) Processing delay: the time taken to process a packet (i.e., packet forward time); (e) Latency: the sum of all possible delays a packet can encounter during data transmission.

Packet loss is computed by sending a specific number of packets and measuring the percentage of those packets that fail to reach their destination. Bandwidth is estimated by continuously sending, over a TCP connection, data packets of known size to a specific destination node. Bandwidth capacity is the maximum amount of transmittable data over a communication channel for a specified amount of time. To estimate this value, it is necessary to know the number of bytes (data packets) sent through the channel and divide it by the time it took

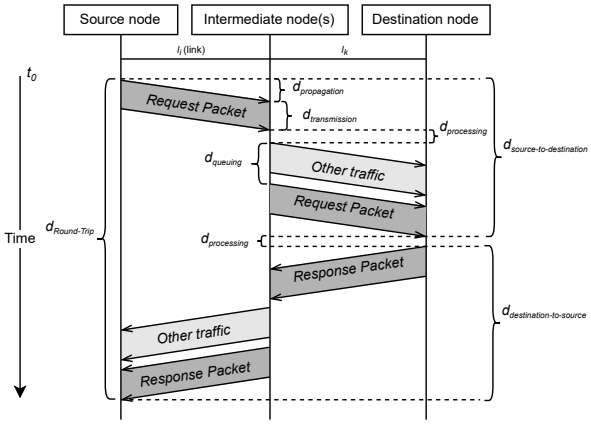


Fig. 5. Delay measure

to receive them without any loss. For that purpose, few packets are transmitted initially. Over time, the number of transmitted packets is increased until a transmission error occurs. The main idea is to send as much data as possible until an error occurs to obtain the transmission time; bandwidth and packet loss are measured using the `iperf` tool. Other parameters such as the link source node, link destination node, total number of nodes and the network density (the proportion of possible links in the network that are actually present) are also included; these parameters are calculated (taken from the network description) by our native software.

An example of a dataset is given in Table 2. It is composed of rows (measures, at a specific time instance) and columns (the parameters or features measured). For instance, let us consider the first link measure. It is composed of 10 parameters or features (columns) with different values. It is important to note that between rows, the difference in time is one second. It is the minimum value that

SRC	DST	BND	PL	MRTT	ARTT	XRTT	DRTT	TN	DEN	LB
1	2	0.00	0.00	1.67	22.30	107.29	39.62	4.00	0.83	0
1	2	0.00	10.00	1.67	28.52	125.89	50.09	4.00	0.83	0
1	2	0.00	0.00	1.69	1.75	1.96	0.09	4.00	0.83	0
2	3	0.00	0.00	1.68	3.93	22.95	6.34	4.00	0.83	0
2	3	0.95	0.00	1.65	1.71	1.80	0.04	4.00	0.83	0
2	3	0.94	10.00	1.68	1.71	1.73	0.01	4.00	0.83	0
2	3	0.95	10.00	1.67	19.20	158.71	49.32	4.00	0.83	0

Abbreviations: BND, Bandwidth; PL, Packet loss; MRTT, Min RTT; ARTT, Avg RTT; XRTT, Max RTT; DRTT, Mdev RTT; TN, Number of nodes; DEN, Network Density; LB, Label

Table 2. Dataset example

allows the proper time to execute the probes and obtain the measurements. In each row, the first 2 columns represent the link; first, the source node (SRC) with a value of 1, and second, the destination node (DST) with a value of 2. Then, the following columns represent the measured parameters assigned to this link: Bandwidth (BND) equal to 0.00 Mbps; Packet loss (PL) equal to 0.00%; Minimum Round trip time (MRTT) equal to 1.67 ms; Average Round trip time (ARTT) equal to 22.30 ms; Maximum Round trip time (XRTT) equal to 107.29 ms; Standard deviation round trip time (DRTT) equal to 39.62 ms; network total number of nodes equal to 4, and network density equal to 0.83. Lastly, a label is assigned to 0, without loss of generality. Note that this label is required to be present in our dataset as explained in the next section.

6 Dataset verification using STM solvers

6.1 Structured datasets

We consider that a *structured dataset* contains *examples* and their *expected outputs*. In our work, we assume that the expected outputs are always present. Thus, a dataset that does not require them (for example, for unsupervised machine learning where there are no expected outputs) has the same expected output for all training examples. Further, we consider only structured datasets.

The inputs are called *features* or *parameters*. A *feature vector*, denoted as $\mathbf{X}$, is an n-tuple of the different inputs, $x_1, x_2, \ldots, x_n$. The expected output for a given feature vector is called a *label*, denoted simply as y, and the possible set of outputs is respectively denoted as Y. The set of examples, called a *dataset*, consists of pairs of a feature vector and a label; each pair is called a *training example*, denoted as $(\mathbf{X}, y)$. For convenience, we represent the dataset as a matrix $D_{m\times n}$ and a vector O_m where D contains the feature vectors and O contains the expected outputs for a dataset of cardinality m. The vector representing the i-th row (training vector) is denoted as D_i, and its associated expected output as O_i. Likewise, the j-th feature (column vector) is denoted as D_j^T (D^T denotes the transpose of the matrix D). Finally, the j-th parameter of the i-th training example is denoted by the matrix element $d_{i,j}$.

6.2 Verifying properties over datasets

Note that the definition of the matrix D does not specify the type of each feature in the dataset. In general, there is no theoretical limitation over the type of these features. Nonetheless, for practical reasons, we consider that all features are real-valued. The main reason is that, otherwise, additional information would be required for each feature. Moreover, in practice, well-known libraries work with real-valued features. As usual, for those features which are not *naturally* real, an encoding must be found (for example, one hot encoding for categorical features, etc.). Thus, we consider that $d_{i,j}, o_i \in \mathbb{R}\ \forall i \in \{1,\ldots,m\}, j \in \{1,\ldots,n\}$. If a dataset is not *labeled*, then $\forall i,k \in \{1,\ldots,m\}\ o_i = o_k$.

Algorithm 1: dataset encoding [20]

Input : A dataset $D_{M\times N}$ (with N features and M training examples), and its expected output vector O_M
Output: A MSFOL formula representation of the dataset, ϕ
Step 0: Set $\phi \leftarrow$ TRUE, set *labels* $\leftarrow$ARRAY(), and set $L \leftarrow 0$;
Step 1: Set $\phi \leftarrow \phi \wedge (m, n, l \in \mathbb{Z}) \wedge (m = M) \wedge (n = N)$;
Step 2: Set $\phi \leftarrow \phi \wedge (\mathcal{D} \in \mathbb{A}_{\mathbb{Z},\mathbb{A}_{\mathbb{Z},\mathbb{R}}}) \wedge (\mathcal{O} \in \mathbb{A}_{\mathbb{Z},\mathbb{R}}) \wedge (\mathcal{L} \in \mathbb{A}_{\mathbb{Z},\mathbb{R}})$;
Step 3: **for** $i \leftarrow 0; i < M; i \leftarrow i + 1$ **do**
  Set *add* $\leftarrow$TRUE;
  for $j \leftarrow 0; i < N; j \leftarrow j + 1$ **do**
    Set $\phi \leftarrow \phi \wedge (\mathcal{D}[i][j] = d_{i,j})$;
  Set $\phi \leftarrow \phi \wedge (\mathcal{O}[i] = o_i)$;
  for $k \leftarrow 0; k < L; k \leftarrow k + 1$ **do**
    if $labels[k] = o_i$ **then**
      Set *add* $\leftarrow$FALSE;
  if *add* **then**
    Set $labels[L] \leftarrow o_i$;
    Set $\phi \leftarrow \phi \wedge (\mathcal{L}[L] = o_i)$;
    Set $L \leftarrow L + 1$;
Step 4: Set $\phi \leftarrow \phi \wedge (l = L)$ and **return** ϕ

Encoding a dataset as a MSFOL formula. Having a convenient formal description for a dataset eases the encoding of this dataset as a MSFOL formula. To encode the data as a formula, we make use of the theory of arrays[4]. We denote that an object a is of sort array with indices of type (sort) $\mathcal{T}1$ and holding objects of type $\mathcal{T}2$ as $a \in \mathbb{A}_{\mathcal{T}1,\mathcal{T}2}$. Indeed, a dataset can be encoded using Algorithm 1 [20]; the algorithm creates a formula that is satisfiable by an interpretation of arrays representing the dataset.

6.3 Formal verification of datasets

A dataset can be formally defined as an MSFOL formula ϕ_{ds} which holds the following properties: ϕ_{ds} is a conjunction of *five* main parts, that is, i) the assertion that an integer variable m is of the size of the number of training examples, a variable n is of the size of the features and a variable l is of the size of the distinct labels, ii) the assertion that $\mathcal{D}$ is a two-dimensional (integer indexed) real-valued array (of size $m \times n$) and $\mathcal{O}, \mathcal{L}$ are integer indexed real-valued arrays (of size m, and l, respectively), iii) $\mathcal{D}[i][j]$ contains the j-th feature value for the i-th training example, iv) $\mathcal{O}[i]$ contains the expected output for the i-th training example, and v) $\mathcal{L}[i]$ contains the i-th (distinct) label.

We assume that we want to verify k properties over the dataset and, furthermore, that these properties are also expressed in MSFOL. Indeed, MSFOL

[4] The theory of arrays considers basic read and write axioms [32]

allows to express many properties of interest (we showcase its expressiveness in Section 6.4). Therefore, we assume that we are given $\pi_1, \ldots, \pi_k$ MSFOL formulas to verify. These properties involve the variables in ϕ_{ds}. Additionally, we assume that these formulas should all *hold* independently over the dataset, and their conjunction is *satisfiable*. This fact imposes a restriction that $\pi_x \wedge \pi_y$ is satisfiable, for $x, y \in \{1, \ldots, k\}$; we call this set of properties the *dataset specification* σ. This means that two properties should not *contradict* each other. For example, it cannot be required that the dataset has more than 30 training examples and at the same time that it must have at most 20 $((\pi_1 \leftrightarrow (m > 30)) \wedge (\pi_2 \leftrightarrow (m \leq 20)))$. Further, the fact that the conjunction of properties must be satisfiable means that there is an interpretation that makes this formula (the conjunction) evaluate to TRUE, i.e., there exists a dataset that can satisfy this specification. Otherwise, the verification of any dataset is useless as no dataset can hold such set of properties.

The formal dataset verification problem can be reduced to the following: given a dataset formula ϕ_{ds} (created using Algorithm 1 from D and O) and a dataset specification $\sigma = \bigwedge_{l=1}^{k} \pi_l$, is $\phi_{ds} \wedge \sigma$ satisfiable? If the conjunction of these formulas is satisfiable then each of the properties must hold for the dataset, and we say that the dataset *holds* the properties $\pi_1, \ldots, \pi_k$ or that the dataset *conforms* to the specification σ. Perhaps this is quite an abstract view of the problem. For that reason, in the following subsection we provide concrete examples that should help the Reader to understand better.

6.4 Example dataset and properties

Let us consider a very small dataset as shown in Table 2. We assume that the dataset D is presented in the first part of the table and the last column O keeps the expected outputs/labels. After applying Algorithm 1 to D and O, the output (ϕ_{ds}) is:

$(m, n, l \in \mathbb{Z}) \wedge (m = 7) \wedge (n = 2) \wedge (\mathcal{D} \in \mathbb{A}_{\mathbb{Z}, \mathbb{A}_{\mathbb{Z},\mathbb{R}}}) \wedge (\mathcal{O} \in \mathbb{A}_{\mathbb{Z},\mathbb{R}}) \wedge (\mathcal{L} \in \mathbb{A}_{\mathbb{Z},\mathbb{R}}) \wedge$
$(\mathcal{D}[0][0] = 1) \wedge (\mathcal{D}[0][1] = 2) \wedge (\mathcal{D}[0][2] = 0.00) \wedge (\mathcal{D}[0][3] = 0) \wedge (\mathcal{D}[0][4] = 1.666) \wedge$
$(\mathcal{D}[0][5] = 22.297) \wedge (\mathcal{D}[0][6] = 107.294) \wedge (\mathcal{D}[0][7] = 39.622) \wedge (\mathcal{D}[0][8] = 4) \wedge$
$(\mathcal{D}[0][9] = 0.83) \wedge (O[0] = 0) \wedge (\mathcal{L}[0] = 0)$
$(\mathcal{D}[1][0] = 1) \wedge (\mathcal{D}[1][1] = 2) \wedge (\mathcal{D}[1][2] = 0.00) \wedge (\mathcal{D}[1][3] = 10) \wedge (\mathcal{D}[1][4] = 1.672) \wedge$
$(\mathcal{D}[1][5] = 28.520) \wedge (\mathcal{D}[1][6] = 125.892) \wedge (\mathcal{D}[1][7] = 50.092) \wedge (\mathcal{D}[1][8] = 4) \wedge$
$(\mathcal{D}[1][9] = 0.83)$
$(\mathcal{D}[2][0] = 1) \wedge (\mathcal{D}[2][1] = 2) \wedge (\mathcal{D}[2][2] = 0.00) \wedge (\mathcal{D}[2][3] = 0) \wedge (\mathcal{D}[2][4] = 1.687) \wedge$
$(\mathcal{D}[2][5] = 1.753) \wedge (\mathcal{D}[2][6] = 1.964) \wedge (\mathcal{D}[2][7] = 0.090) \wedge (\mathcal{D}[2][8] = 4) \wedge$
$(\mathcal{D}[2][9] = 0.83)$
$(\mathcal{D}[3][0] = 2) \wedge (\mathcal{D}[3][1] = 3) \wedge (\mathcal{D}[3][2] = 0) \wedge (\mathcal{D}[3][3] = 0) \wedge (\mathcal{D}[3][4] = 1.684) \wedge$

$(\mathcal{D}[3][5] = 3.927) \wedge (\mathcal{D}[3][6] = 22.946) \wedge (\mathcal{D}[3][7] = 6.343) \wedge (\mathcal{D}[3][8] = 4) \wedge$
$(\mathcal{D}[3][9] = 0.83)$
$(\mathcal{D}[4][0] = 2) \wedge (\mathcal{D}[4][1] = 3) \wedge (\mathcal{D}[4][2] = 0.95) \wedge (\mathcal{D}[4][3] = 0) \wedge (\mathcal{D}[4][4] = 1.653) \wedge$
$(\mathcal{D}[4][5] = 1.708) \wedge (\mathcal{D}[4][6] = 1.801) \wedge (\mathcal{D}[4][7] = 0.037) \wedge (\mathcal{D}[4][8] = 4) \wedge$
$(\mathcal{D}[4][9] = 0.83)$
$(\mathcal{D}[5][0] = 2) \wedge (\mathcal{D}[5][1] = 3) \wedge (\mathcal{D}[5][2] = 0.94) \wedge (\mathcal{D}[5][3] = 10) \wedge (\mathcal{D}[5][4] = 1.676) \wedge$
$(\mathcal{D}[5][5] = 1.707) \wedge (\mathcal{D}[5][6] = 1.727) \wedge (\mathcal{D}[5][7] = 0.014) \wedge (\mathcal{D}[5][8] = 4) \wedge$
$(\mathcal{D}[5][9] = 0.83)$
$(\mathcal{D}[6][0] = 2) \wedge (\mathcal{D}[6][1] = 3) \wedge (\mathcal{D}[6][2] = 0.95) \wedge (\mathcal{D}[6][3] = 10) \wedge (\mathcal{D}[6][4] = 1.665) \wedge$
$(\mathcal{D}[6][5] = 19.204) \wedge (\mathcal{D}[6][6] = 158.710) \wedge (\mathcal{D}[6][7] = 49.322) \wedge (\mathcal{D}[6][8] = 4) \wedge$
$(\mathcal{D}[6][9] = 0.83) \wedge (l = 1)$

We now showcase some very simple properties together with the formal verification process. Suppose that the specification consists of a single property: "the dataset must contain at least 100 training examples." This property can be expressed in MSFOL simply as $\pi_{\#} \leftrightarrow (m \geq 100)$. Notice how $\phi_{ds} \wedge \pi_{\#}$ is not satisfiable as there does not exist an interpretation that makes it evaluate to TRUE; particularly, if m is greater than 99, then the clause (in ϕ_{ds}) $m = 7$ cannot evaluate to TRUE and since this is a conjunction, $\phi_{ds} \wedge \pi_{\#}$ evaluates to FALSE. Similarly, if m is 7, then the $\pi_{\#}$ makes the conjunction evaluate to FALSE. Thus, we say that the dataset does not hold the property $\pi_{\#}$.

A slightly more complex property to verify is: "the dataset must be min-max normalized," which can be expressed in MSFOL as $\pi_{\pm} \leftrightarrow \nexists(i, j \in \mathbb{Z})((i \geq 0) \wedge (i < n) \wedge (j \geq 0) \wedge (j < m) \wedge ((\mathcal{D}[i][j] < min) \vee (\mathcal{D}[i][j] > max)))$. Certainly, min and max are defined constants (e.g., -1 and 1) an either these variables must be defined or the value must be replaced; for $min = 0$ and $max = 1000$, ϕ_{ds} holds the property $\pi_{\pm}$ (as $\phi_{ds} \wedge \pi_{\pm}$ is satisfiable).

We have showcased the flexibility of the proposed approach with somewhat standard properties to check. Nonetheless, it is interesting to point out that the approach is generic and domain-specific properties coming from expert knowledge can also be used. This is the primary motivation for the dataset verification in our work. As previously stated, we focus on guaranteeing that the behavior of the emulator fulfills the requirements of the physical system; the goal is to reduce the behavioral differences between the emulator and the real system. In general, as the properties to check can be added or removed arbitrarily, checking a given set of those for a particular dataset is possible.

7 Experimental results

This section discusses an experimental evaluation of our emulator and dataset verification approach. The main objectives of this experimental evaluation are:

i) to check the execution of the emulator w.r.t. a set of real physical properties; ii) to check the performance of the proposed approach (in terms of execution time and used space). For this reason, datasets of different sizes were extracted from our emulator; these datasets were verified over a large set of properties with various degrees of dependencies between data. Our dataset verification tool [20] makes use of the z3 theorem prover. In order to replicate our experiments, z3 versions 4.8.11 until 4.11.0 should be avoided, as they contain an incompatibility issue. At the time of this writing, we recommend version 4.8.10. However, future versions (after 4.8.11) should contain a bug fix as per our issue report (issue 6304 on z3's GitHub repository).

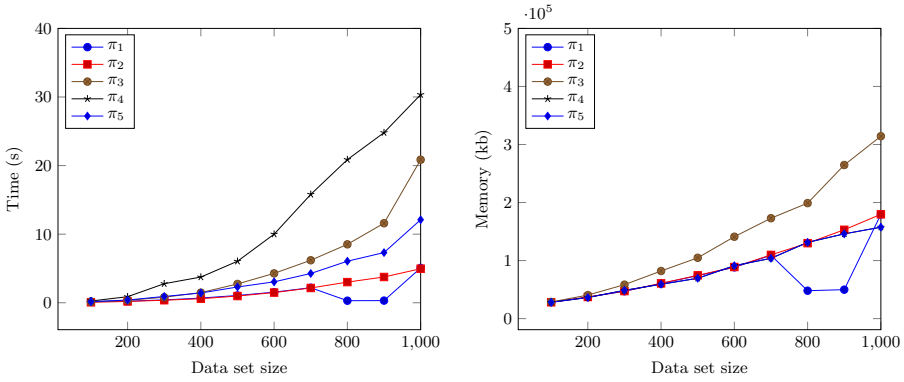
Experimental setup All experiments were executed on an Ubuntu 22.04LTS, running on an AMD Ryzen 1900X (8-core/16-thread) @ 3.8GHz, and 64GB of RAM. An extracted dataset was used by executing our emulator with the topology shown in 1. The data collection time was 20 minutes. As the monitoring interval is one second, this yields a dataset with 1200 training examples. The properties of interest are divided into three different groups. Group#1 – contains properties that verify features within one training example (line, row) but without dependencies between features. For example, the delay in all measures must not exceed a certain threshold. This verification can be done over a training example or all of them but, it is usually interesting to make it for all. Group#2 – contains properties that verify some dependencies between the network features within one training example. For instance, if in a training example, the source is s and the destination is d, then the delay of this link must not exceed a given constant. Finally, Group#3 – contains properties that reflect dependencies between training examples. For example, for a given link (a, b) the bandwidth must be greater than that on an adjacent link (b, c). The verified properties are written in the SMT-LIB language and are available in our repository [23].

Experimental results In the following figures and tables, we show the grouped properties and the performance results. Figures 6a and 6b, show the execution time and maximal required space (respectively) for properties shown in Table 3. Correspondingly, Figures 7a and 7b, show the execution time and maximal required space (respectively) for properties shown in Table 4. Likewise, Figures 8a and 8b, show the execution time and maximal required space (respectively) for properties shown in Table 5.

The time taken to verify increases with each group. This is expected and natural as the more complex properties take longer to be checked. However, note that oftentimes properties that are natural to verify for the domain of computer networks mostly fall under the groups #1 and #2. Furthermore, our offline approach is pertinent as verifying properties takes a comparable amount of time w.r.t. the emulation execution time (especially for properties with small execution time results). This is promising, especially for highly sensitive emulations whose correct functioning must be guaranteed and/or certified. The memory consumption of the verification process is confirmed to be polynomial w.r.t. to

Description	**Formula**
The delay (average round trip time) belongs to the range [min,max] (min and max are given constants, e.g., 0 and 300), for all measures (training examples) in the dataset	$\pi_1 = \nexists i \ ((i \geq 0) \wedge (i < m - 1) \wedge ((\mathcal{D}[i][AVG\_RTT] < min) \vee (\mathcal{D}[i][AVG\_RTT] > max)))$
The bandwidth capacity belongs to the range [min,max] (e.g., 0 and 50), for all measures (training examples) in the dataset	$\pi_2 = \nexists i \ ((i \geq 0) \wedge (i < m - 1) \wedge ((\mathcal{D}[i][BANDWIDTH] < min) \vee (\mathcal{D}[i][BANDWIDTH] > max)))$
The average packet loss of all measures (training examples) in the dataset must not exceed a given constant M (for example, 0.3)	$\pi_3 = (\frac{1}{m} \sum_{i=0}^{m-1} \mathcal{D}[i][PACKET\_LOSS]) \leq M$
The minimum bandwidth capacity of all measures (training examples) in the dataset must be greater or equal to a given constant μ (for example, 10)	$\pi_4 = \mathbf{min}\{\mathcal{D}[i][BANDWIDTH] \mid i \in \{1, \ldots, m-1\}\} \geq \mu$ (for readability we do not include the implementation of functions denoted in bold as **min**, however, the interested Reader may refer to our repository [23] to check the code implementations)
The maximum delay of all measures (training examples) in the dataset must not exceed a given constant $\mathcal{M}$ (for example, 300)	$\pi_5 = \mathbf{max}\{\mathcal{D}[i][BANDWIDTH] \mid i \in \{1, \ldots, m-1\}\} \leq \mathcal{M}$

Table 3. Group#1 network properties



(a) Processing time (b) Memory

Fig. 6. Results properties group #1

the dataset size. Finally, the verification yields a satisfiable verdict over our emulation in 86.66% of the verified properties (that is 130 out of 150 properties satisfied).

Description	Formula
For all measures (training examples) in the dataset, the delay (average round trip time) belongs to the range [min,max] (min and max are given constants, e.g., 0 and 200) if the source node is s and the destination node is d, for a given particular link (s, d)	$\rho_1 = \nexists i\ ((i \geq 0) \wedge (i < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d) \wedge ((\mathcal{D}[i][AVG\_RTT] < min) \vee (\mathcal{D}[i][AVG\_RTT] > max)))$
For all measures (training examples) in the dataset, the bandwidth capacity belongs to the range [min,max] (min and max are given constants, e.g., 10 and 20), if the source node is s and the destination node is d, for a given particular link (s, d)	$\rho_2 = \nexists i\ ((i \geq 0) \wedge (i < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d) \wedge ((\mathcal{D}[i][BANDWIDTH] < min) \vee (\mathcal{D}[i][BANDWIDTH] > max)))$
For all measures (training examples) in the dataset, the packet lost belongs to the range [min,max] (min and max are given constants. e.g., 0 and 0.5) if the source node is s and the destination node is d, for a given particular link (s, d)	$\rho_3 = \nexists i\ ((i \geq 0) \wedge (i < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d) \wedge ((\mathcal{D}[i][PACKET\_LOST] < min) \vee (\mathcal{D}[i][PACKET\_LOST] > max)))$
For all measures (training examples) in the dataset, the bandwidth capacity must be greater or equal than a given constant B, if the source node is s and the destination node is d, i.e., for a given particular link (s, d)	$\rho_4 = \forall i\ (((i \geq 0) \wedge (i < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d)) \implies (\mathcal{D}[i][BANDWIDTH] \geq B))$
For all measures (training examples) in the dataset, the packet loss must not exceed a given constant L if the source node is s and the destination node is d, for a given particular link (s, d)	$\rho_5 = \forall i\ (((i \geq 0) \wedge (i < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d)) \implies (\mathcal{D}[i][PACKET\_LOSS] \leq L))$

Table 4. Group#2 network properties

The violated properties are properties related to bandwidth, the reason is that our emulator allows a dynamic change in bandwidth. Thus, the measures that are taken when the bandwidth changes report a capacity of 0mbps (as sending data report a failure). This is a technical limitation and the expected behavior. The failed properties verified that the bandwidth should always be above a given constant (10mbps in our experiments). This showcases the utility of our tool as it helps reveal important details of emulated solutions.

8 Conclusion

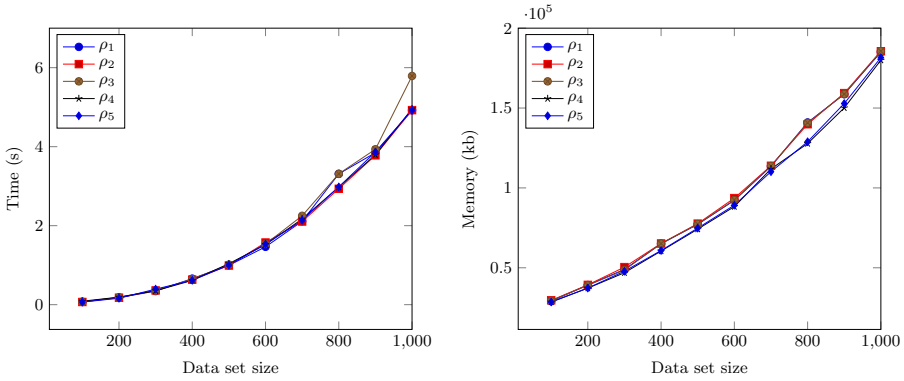
In this paper, we have showcased the design and architecture for a dynamic link network emulator. Moreover, we have presented an approach for verifying that

Description	**Formula**
For a given (valid) path $(a \to b \to c)$ the average delay does not differ more than C time units from its return path $(c \to b \to a)$	$\phi_1 = \lvert\frac{1}{m}((\textbf{cond\_sum}(AVG\_RTT, a, b) + \textbf{cond\_sum}(AVG\_RTT, b, c)) - (\textbf{cond\_sum}(AVG\_RTT, c, b) + \textbf{cond\_sum}(AVG\_RTT, b, a)))\rvert \leq C$, where $\textbf{cond\_sum}(f, s, c) = \sum\{\mathcal{D}[i][f] \lvert i \in \{1, \ldots, m-1\} \wedge \mathcal{D}[i][SRC] = s \wedge \mathcal{D}[i][DST] = d\}$
For a given (valid) path $(a \to b \to c)$ the average packet loss does not differ more than C units from its return path $(c \to b \to a)$	$\phi_2 = \lvert\frac{1}{m}((\textbf{cond\_sum}(PACKET\_LOSS, a, b) + \textbf{cond\_sum}(PACKET\_LOSS, b, c)) - (\textbf{cond\_sum}(PACKET\_LOSS, c, b) + \textbf{cond\_sum}(PACKET\_LOSS, b, a)))\rvert \leq C$
For a given (valid) path $(a \to b \to c)$ the minimum outgoing (upload) bandwidth does not exceed more than C times the minimum incoming bandwidth (download, return path) $(c \to b \to a)$	$\phi_3 = \textbf{min}\{\textbf{cond\_min}(BANDWIDTH, a, b), \textbf{cond\_min}(BANDWIDTH, b, c)) \leq C * \textbf{min}\{\textbf{cond\_min}(BANDWIDTH, c, b), \textbf{cond\_min}(BANDWIDTH, b, a)\}$, where $\textbf{cond\_min}(f, s, c) = \textbf{min}\{\mathcal{D}[i][f] \lvert i \in \{1, \ldots, m-1\} \wedge \mathcal{D}[i][SRC] = s \wedge \mathcal{D}[i][DST] = d\}$
For a given (valid) path $(a \to b \to c)$ the minimum outgoing (upload) bandwidth-delay product (the bandwidth-delay product is a common data communications metric used to measure the maximum amount of data that can be transmitted and not yet received at any time instance) does not exceed more than C times the minimum incoming bandwidth-delay product (download, return path) $(c \to b \to b)$	$\phi_4 = \textbf{min}\{\textbf{cond\_min\_prod}(a, b), \textbf{cond\_min\_prod}(b, c)) \leq C * \textbf{min}\{\textbf{cond\_min\_prod}(c, b), \textbf{cond\_min\_prod}(b, a)\}$, where where $\textbf{cond\_min\_prod}(s, c) = \textbf{min}\{\mathcal{D}[i][BANDWIDTH] * \mathcal{D}[i][AVG\_RTT] \lvert i \in \{1, \ldots, m-1\} \wedge \mathcal{D}[i][SRC] = s \wedge \mathcal{D}[i][DST] = d\}$
For all measures in the dataset, for a given link $((s, d))$ the delay (average round trip time) does not differ more than C time units from its return link $((d, s))$, for each observation in the dataset	$\phi_5 = \forall i, j(((i \geq 0) \wedge (i < m) \wedge (j \geq 0) \wedge (j < m) \wedge (\mathcal{D}[i][SRC] = s) \wedge (\mathcal{D}[i][DST] = d) \wedge (\mathcal{D}[i][SRC] = \mathcal{D}[j][DST]) \wedge (\mathcal{D}[i][DST] = \mathcal{D}[j][SRC])) \implies (\lvert\mathcal{D}[i][AVG\_RTT] - \mathcal{D}[j][AVG\_RTT]\rvert \leq C))$

Table 5. Group#3 network properties

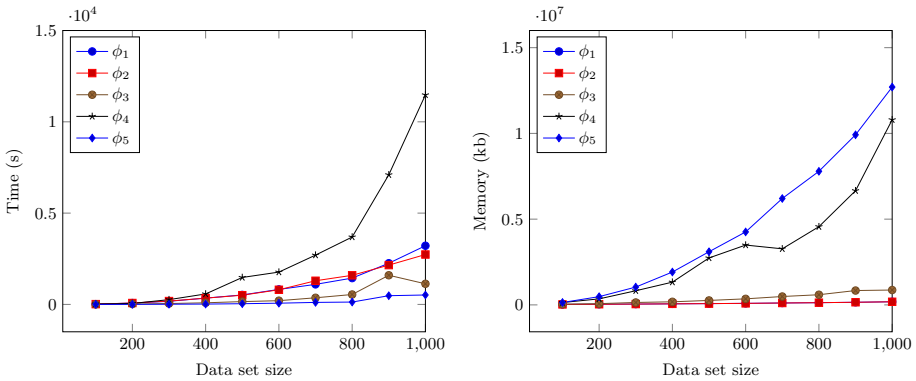
the emulation execution respects certain properties of interest; this is useful to reduce the difference between the behavior of the emulated and the real system. It is important to note that the emulator is flexible and can run any existing software; additionally, it can dynamically change the link parameter values.

In its current state, the tool proposes a solid framework for the emulation of dynamic link networks. However, certain aspects can be improved and new



(a) Processing time (b) Memory

Fig. 7. Results properties group #2



(a) Processing time (b) Memory

Fig. 8. Results properties group #3

features can be incorporated. For instance, it is desirable to reduce the monitoring time, to better guarantee that the emulator always holds the properties of interest. Nonetheless, it is technologically difficult to address this issue, which is an interesting avenue for future work. Furthermore, we plan to incorporate more features into the architecture so that it becomes more controllable and realistic. For example, we consider incorporating link state scenarios to qualify the solutions under different conditions (degraded links, weather conditions, etc.). Additionally, enhancing the verification strategies may allow performing the verification in larger time lapses or closer to runtime monitoring.

References

1. Alsmadi, I., Zarrad, A., Yassine, A.: Mutation testing to validate networks protocols. In: 2020 IEEE International Systems Conference (SysCon). pp. 1–8. IEEE, Montreal, QC, Canada (2020). https://doi.org/10.1109/SysCon47679.2020.9275875
2. Barrett, C., Stump, A., Tinelli, C.: The satisfiability modulo theories library (smtlib). SMT-LIB.org (2010)
3. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018)
4. Chan, M.C., Chen, C., Huang, J.X., Kuo, T., Yen, L.H., Tseng, C.C.: Opennet: A simulator for software-defined wireless local area network. In: 2014 IEEE Wireless Communications and Networking Conference (WCNC). pp. 3332–3336. IEEE (2014)
5. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
6. Deng, B., Jiang, C., Yao, H., Guo, S., Zhao, S.: The next generation heterogeneous satellite communication networks: Integration of resource management and deep reinforcement learning. IEEE Wireless Communications **27**(2), 105–111 (2019)
7. Fambri, G., Diaz-Londono, C., Mazza, A., Badami, M., Sihvonen, T., Weiss, R.: Techno-economic analysis of power-to-gas plants in a gas and electricity distribution network system with high renewable energy penetration. Applied Energy **312**, 118743 (2022)
8. Farias, F.N., Junior, A.d.O., da Costa, L.B., Pinheiro, B.A., Abelém, A.J.: vsdnemul: A software-defined network emulator based on container virtualization. arXiv preprint arXiv:1908.10980 (2019)
9. Gandhi, S., Singh, R.K., et al.: Design and development of dynamic satellite link emulator with experimental validation. In: 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT). pp. 1–6. IEEE (2021)
10. Hemminger, S., et al.: Network emulation with netem. In: Linux conf au. pp. 18–23 (2005)
11. Henderson, T.R., Lacage, M., Riley, G.F., Dowell, C., Kopena, J.: Network simulations with the ns-3 simulator. SIGCOMM demonstration **14**(14), 527 (2008)
12. Horneber, J., Hergenröder, A.: A survey on testbeds and experimentation environments for wireless sensor networks. IEEE Communications Surveys Tutorials **16**(4), 1820–1838 (2014). https://doi.org/10.1109/COMST.2014.2320051
13. Kamp, P.H., Watson, R.N.: Jails: Confining the omnipotent root. In: Proceedings of the 2nd International SANE Conference. vol. 43, p. 116 (2000)
14. Kaur, K., Singh, J., Ghumman, N.S.: Mininet as software defined networking testing platform. In: International Conference on Communication, Computing & Systems (ICCCS). pp. 139–42 (2014)
15. Khan, A.R., Bilal, S.M., Othman, M.: A performance comparison of open source network simulators for wireless networks. In: 2012 IEEE International Conference on Control System, Computing and Engineering. pp. 34–38. IEEE, Penang, Malaysia (2012). https://doi.org/10.1109/ICCSCE.2012.6487111
16. Kodheli, O., Lagunas, E., Maturo, N., Sharma, S.K., Shankar, B., Montoya, J.F.M., Duncan, J.C.M., Spano, D., Chatzinotas, S., Kisseleff, S., et al.: Satellite communications in the new space era: A survey and future challenges. IEEE Communications Surveys & Tutorials **23**(1), 70–109 (2020)

17. Lai, J., Tian, J., Jiang, D., Sun, J., Zhang, K.: Network emulation as a service (neaas): Towards a cloud-based network emulation platform. In: Song, H., Jiang, D. (eds.) Simulation Tools and Techniques. pp. 508–517. Springer International Publishing, Cham (2019)
18. Lantz, B., Heller, B., McKeown, N.: A network in a laptop: rapid prototyping for software-defined networks. In: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks. pp. 1–6. ACM (2010)
19. Liu, Y., Lu, H., Li, X., Zhang, Y., Xi, L., Zhao, D.: Dynamic service function chain orchestration for nfv/mec-enabled iot networks: A deep reinforcement learning approach. IEEE Internet of Things Journal **8**(9), 7450–7465 (2021). https://doi.org/10.1109/JIOT.2020.3038793
20. López, J., Labonne, M., Poletti, C.: Toward formal data set verification for building effective machine learning models. In: Cucchiara, R., Fred, A.L.N., Filipe, J. (eds.) Proceedings of the 13th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2021, Volume 1: KDIR, Online Streaming, October 25-27, 2021. pp. 249–256. SCITEPRESS (2021). https://doi.org/10.5220/0010676500003064
21. Manna, Z., Zarba, C.G.: Combining decision procedures. In: Formal Methods at the Crossroads. From Panacea to Foundational Support, pp. 381–422. Springer (2003)
22. Nurlan, Z., Zhukabayeva, T., Othman, M., Adamova, A., Zhakiyev, N.: Wireless sensor network as a mesh: Vision and challenges. IEEE Access **10**, 46–67 (2022). https://doi.org/10.1109/ACCESS.2021.3137341
23. Petersen, E., López, J., Kushik, N., Labonne, M., Poletti, C., Zeghlache, D.: Dlemudataverif: Dynamic link network emulation and validation of execution data sets. `http://gitlab.ailab.airbus.com/erick.petersen/DLEmuDataVerif.git` (2022)
24. Petersen, E., López, J., Kushik, N., Poletti, C., Zeghlache, D.: On using smt-solvers for modeling and verifying dynamic network emulators: (work in progress). In: 19th IEEE International Symposium on Network Computing and Applications, NCA 2020, Cambridge, MA, USA, November 24-27, 2020. pp. 1–3. IEEE (2020). https://doi.org/10.1109/NCA51143.2020.9306731
25. Petersen, E., López, J., Kushik, N., Poletti, C., Zeghlache, D.: Dynamic link network emulation: A model-based design. In: Kaindl, H., Mannion, M., Maciaszek, L.A. (eds.) Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2022, Online Streaming, April 25-26, 2022. pp. 536–543. SCITEPRESS (2022). https://doi.org/10.5220/0011091100003176
26. Peterson, L.L., Davie, B.S.: Computer networks: a systems approach. Elsevier (2007)
27. Peuster, M., Kampmeyer, J., Karl, H.: Containernet 2.0: A rapid prototyping platform for hybrid service function chains. In: 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft). pp. 335–337. IEEE (2018)
28. Rese, A., Görmar, L., Herbig, A.: Social networks in coworking spaces and individual coworker's creativity. Review of Managerial Science **16**(2), 391–428 (2022)
29. Shan, Q.: Testing methods of computer software. In: 2020 International Conference on Data Processing Techniques and Applications for Cyber-Physical Systems. pp. 231–237. Springer (2021)
30. Srisawai, S., Uthayopas, P.: Rapid building of software-based sdn testbed using sdn owl. In: 2018 22nd International Computer Science and Engineering Conference (ICSEC). pp. 1–4. IEEE (2018)

31. Stoller, M.H.R.R.L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale virtualization in the emulab network testbed. In: USENIX Annual Technical Conference, Boston, MA (2008)
32. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 29–37. IEEE (2001)
33. Sun, Z., Chai, W.K.: Satellite emulator for ip networking based on linux. In: 21st International Communications Satellite Systems Conference and Exhibit. p. 2393 (2003)
34. Varga, A.: Discrete event simulation system. In: Proc. of the European Simulation Multiconference (ESM'2001). pp. 1–7 (2001)
35. Wang, S.Y., Chou, C.L., Yang, C.M.: Estinet openflow network simulator and emulator. IEEE Communications Magazine **51**(9), 110–117 (2013)