Descartes Network Infrastructures (DNI) Manual

Meta-models, Transformations, Examples

Piotr Rygielski, Samuel Kounev
{piotr.rygielski, samuel.kounev}@uni-wuerzburg.de

Technical Report

University of Würzburg

Version 0.3

November 2014

About this Document

Goals

The goal of this document is to present the DNI Meta-Model, its instances, sub-meta-models, and model transformations in more details than in an usual conference or journal paper. We describe the Meta-models and the transformations in details, give examples of systems modeled using the meta-models, and present the transformations in a step-by-step manner.

Version of the DNI Package

The Manual in its current version (0.3) describes the DNI package (i.e.: models, meta-models and the transformations) version 2.3.

Executive Summary

This document is updated on a regular basis. In the table below, we summary the recent changes.

Document Version	Date	DNI Version	Introduced changes
0.3	Nov 2014	2.3	Polishing the DNI Manual, improving diagrams, adding examples.
0.2	Sep 2014	2.3	Initial version of the DNI Manual is published online. Introducing changes in the routing (classical vs. flow-based).
0.1	Jul 2014	2.2	Improvements and optimization for SimQPN transformation. Introducing miniDNI meta-model and its related transformations. First sketch of the DNI Manual.
—	Feb 2014	2.0	Improvements for OMNeT++. Transformation to SimQPN.
—	Jun 2013	1.0	Finalized first version of the DNI meta-model. Transformation to OMNeT++.
—	Jan 2013	0.1	Initial version of the DNI meta-model

Contents

1 Introduction 1

2 Meta-Models 3
- 2.1 DNI . . . 3
  - 2.1.1 Network Configuration . . . 3
  - 2.1.2 Network Structure . . . 5
  - 2.1.3 Network Traffic . . . 6
- 2.2 Example of a DNI Model: SBUS . . . 7
  - 2.2.1 Notation . . . 7
  - 2.2.2 SBUS System . . . 7
  - 2.2.3 Model of the Network Structure . . . 8
  - 2.2.4 Model of the Network Traffic . . . 10
  - 2.2.5 Model of the Network Configuration . . . 10
- 2.3 miniDNI . . . 11
- 2.4 Example of a miniDNI Model: SBUS . . . 11

3 Transformations 15
- 3.1 Map of the Transformations . . . 15
  - 3.1.1 General Remarks . . . 15
- 3.2 DNI-to-OMNeT++ . . . 17
  - 3.2.1 Pre-transformation Operations . . . 17
  - 3.2.2 Transformation Rules . . . 17
  - 3.2.3 Post-transformation Operations . . . 22
- 3.3 DNI-to-miniDNI . . . 23
  - 3.3.1 Pre-transformation Operations . . . 23
  - 3.3.2 Transformation Rules . . . 23
  - 3.3.3 Post-transformation Operations . . . 25
- 3.4 DNI-to-QPN . . . 26
  - 3.4.1 Pre-transformation Operations . . . 26
  - 3.4.2 Transformation Rules . . . 26
  - 3.4.3 Post-transformation Operations . . . 34
- 3.5 miniDNI-to-QPN . . . 36
  - 3.5.1 Pre-transformation Operations . . . 36
  - 3.5.2 Transformation Rules . . . 36
  - 3.5.3 Post-transformation Operations . . . 38
- 3.6 Routing-format-conversion . . . 39

4 Conclusion 45

Bibliography 47

Chapter 1

Introduction

Approach

The modeling approach we propose is based on a new meta-model for modeling network infrastructures in virtualized data centers. This meta-model, which we refer to as Descartes Network Infrastructure (DNI) meta-model, is part of our broader work in the context of the *Descartes Modeling Language* (DML) [BHK13], an architecture-level modeling language for dynamic IT systems and services.

Our approach assumes that instances of the DNI Meta-model are automatically transformed to predictive stochastic models (e.g., product-form queueing networks or stochastic simulation models) by means of model-to-model transformations. Thus, our modeling approach does not require explicit knowledge and expertise in stochastic modeling and analysis. The DNI Meta-model has been designed to support describing the most relevant performance influencing factors that occur in practice while abstracting fine-granular low level protocol details. Our approach is designed to support the implementation of different transformations of the high-level DNI models to underlying predictive stochastic models (by abstracting environment-specific details, transformations to multiple predictive models are possible), thereby providing flexibility in the trade-off between the overhead and accuracy of the analysis.

Chapter 2

Meta-Models

2.1 DNI

The DNI meta-model (Descartes Networking Infrastructures) covers three main parts of every data center network infrastructure: structure, traffic and configuration. It is implemented in Ecore using the Eclipse Modeling Framework (EMF). An initial preliminary version of the DNI Meta-model was presented as a work-in-progress paper in [RZK13]; since then, the meta-model has evolved significantly and therefore we present a brief overview of its major parts in the following.

The root element of the DNI Meta-model (*NetworkInfrastructure*) connects three main parts: network structure, traffic and configuration (see Fig. 2.1). To analyze the performance of any network infrastructure, one must know how the network is physically built (*NetworkStructure*), how it is configured (*NetworkConfiguration*) and how it is used (*NetworkTraffic*). Every numeric value in the model is modeled as a *Dependency* (Fig. 2.3). The *Dependency* represents a *Variable* (constant or random) or a *Function*. Additionally, each *Dependency* can be accompanied with a *Unit*. Examples of a *Dependency* can be the following descriptions of a parameter: *"exponentially distributed with mean value of* $100ms$*"*, or just *"5Mbps"*.

2.1.1 Network Configuration

The network-configuration meta-model is presented in Figure 2.4. Currently, the *NetworkConfiguration* contains information about routes, protocols and protocols stacks. In the model, we describe a snapshot of the current routes in the system; we

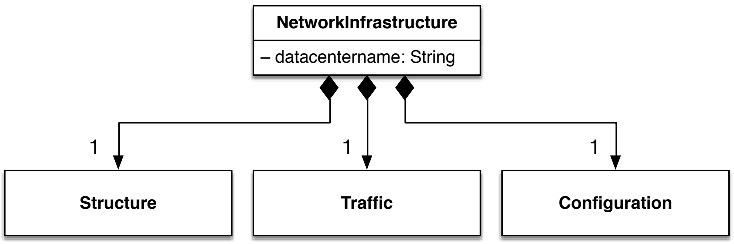

Figure 2.1: Root of the DNI meta-model.

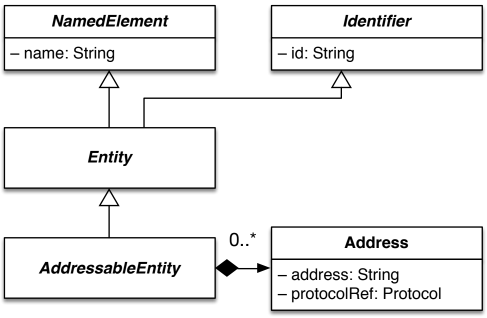

Figure 2.2: Common classes used in the DNI meta-model.

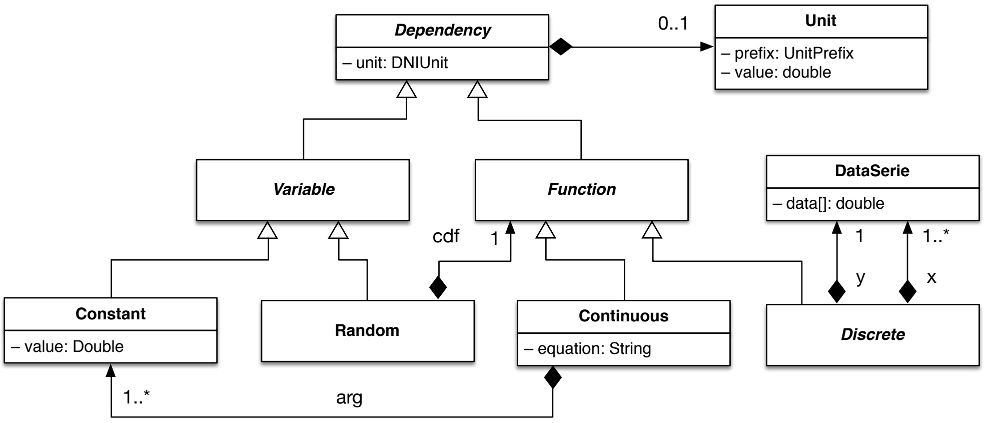

Figure 2.3: In the DNI meta-model, Dependencies are used to model numeric values and functions.

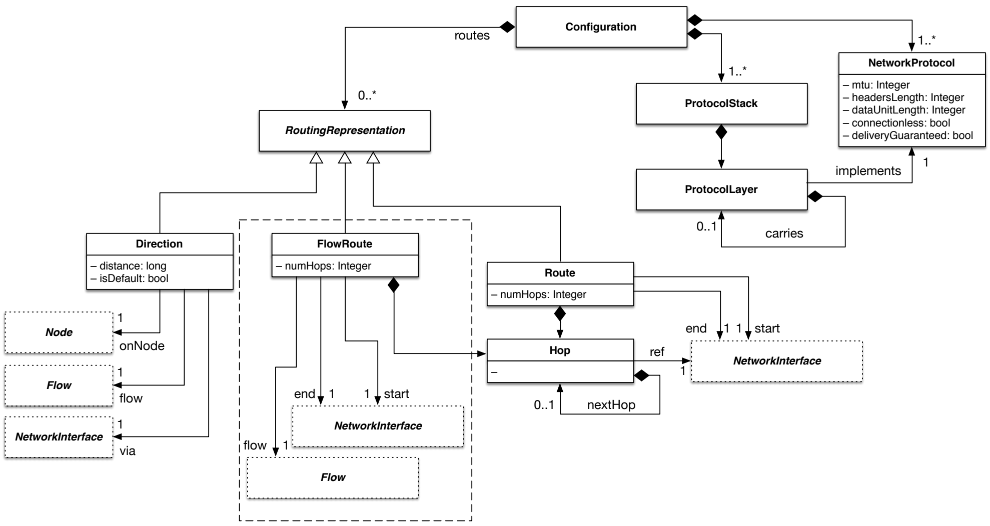

Figure 2.4: Meta-model of Network configuration. Entities in the dashed box represent a intermediary step for transforming *Routes* info *Directions*. Dotted entities represent the entities form the otehrs parts of the DNI meta-model (Infrastructure or Traffic).

do not explicitly consider dynamic routing as this would require detailed information about routing algorithms which is abstracted here. In the meta-model, a *Route* consists of *Hop*s and each *Hop* references a *NetworkInterface*. The term *route* is an abstraction; we do not store information about dynamic routing protocols. A *ProtocolStack* is an ordered set of *ProtocolLayer*s, where each *ProtocolLayer* references a single *NetworkProtocol*. The *NetworkProtocol* itself is described by a generic set of parameters such as, for example, overheads introduces by the data unit headers.

2.1.2 Network Structure

The part of the meta-model representing the network structure is depicted in Figure 2.5. The *NetworkStructure* is a graph consisting of *Node*s and *Link*s connected through *NetworkInterface*s. All nodes, links and interfaces can be either physical or virtual; each virtual network element is hosted on a *PhysicalNode*. The performance-relevant influencing factors of every element in the *NetworkStructure* are described using **Performance* entities, both for physical and virtual elements. We distinguish end nodes (e.g., virtual machine, server) and intermediate nodes (e.g., switch, router), because their performance descriptions are different, e.g., end nodes do not utilize information about forwarding performance.

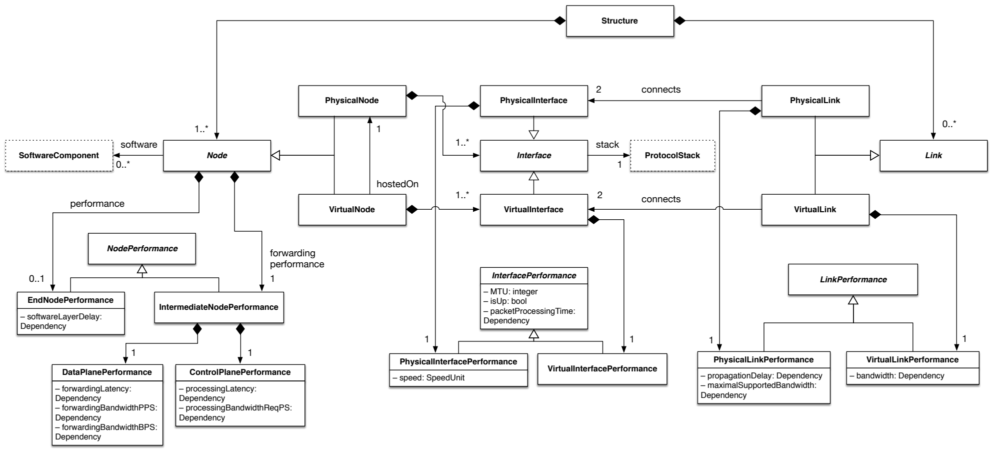

Figure 2.5: Meta-model of Network structure.

2.1.3 Network Traffic

In a data center, most of the network traffic is generated by deployed applications. This includes also the hypervisors (applications) which can trigger, e.g., VM migrations (traffic). As depicted in Figure 2.6, the DNI meta-model, network traffic is generated by *TrafficSource*s that originate from *SoftwareComponent*s. Software components are deployed on end nodes. Each *TrafficSource* generates traffic *Flow*s that have exactly one source and possibly multiple destinations. The *Flow* destinations are located in *Node*s and can be uniquely identified by a set of protocol-level addresses. Each *TrafficSource* can generate a specified set of flows. The information about the precise transmission time of a flow is modeled in the workload model (*GenericWorkload*). Each flow can be described by means of various flow descriptions; in this paper, we use a *GenericFlow* description capturing the amount of transferred data. The meta-model can be extended to support other traffic models that can be found in the literature, e.g., [FHH02, KMF04].

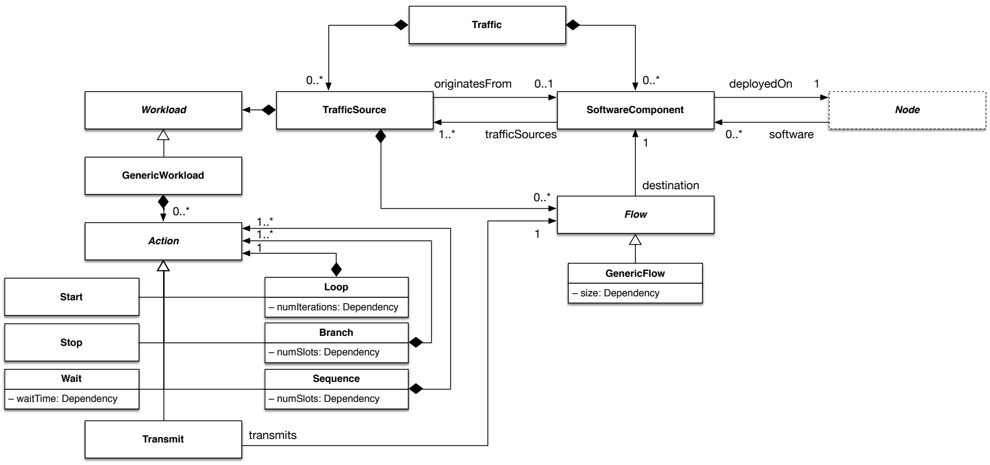

Figure 2.6: Meta-model of Network traffic.

2.2 Example of a DNI Model: SBUS

An instance of the DNI Meta-model can be called a DNI model. A DNI model represents a concrete infrastructure and setup of a real system that is being modeled. To ease the understanding of the DNI meta-model, we propose an example that models the SBUS scenario [Ing09].

2.2.1 Notation

To present DNI models we use diagram similar to the Object diagram known from UML. Due to size of the diagrams, we connect the parts of a single diagrams using a connector 2.7. The connector is represented by a circle with a name that eases to map it to the objects that it is connecting. Additionally, in the diagrams we use colors to make the reading of the diagram easier. *Node* objects are cloured gray, *traffic*-related objects and connectors are light-blue, and finally, the objects related to *performance descriptions* are marked in green.

The entities marked with dotted lines represent entities form other parts of the given meta-model. We do not present the whole meta-model in one diagram to not decrease the readability. The dashed frames are used to highlight selected parts of a model.

2.2.2 SBUS System

The SBUS system is a traffic monitoring application based on results from the Transport Information Monitoring Environment (TIME) project [BBE$^+$08] at the University of Cambridge. The system consists of multiple distributed components and is based on the SBUS/PIRATES (short SBUS) middleware [Ing09].

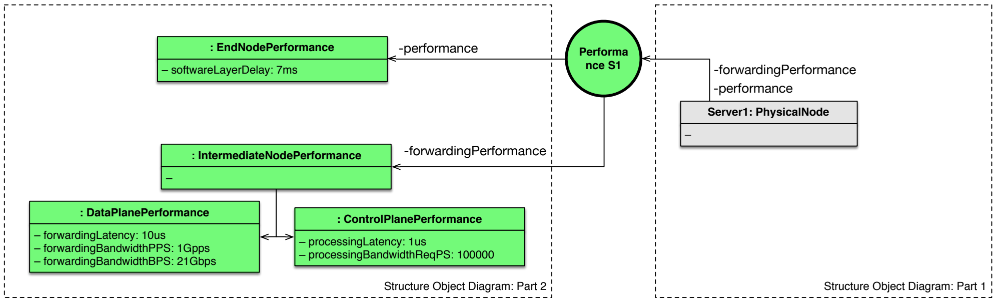
: EndNodePerformance
– softwareLayerDelay: 7ms
-performance
Performance S1
-forwardingPerformance
-performance
Server1: PhysicalNode
–
: IntermediateNodePerformance
–
-forwardingPerformance
: DataPlanePerformance
– forwardingLatency: 10us
– forwardingBandwidthPPS: 1Gpps
– forwardingBandwidthBPS: 21Gbps
: ControlPlanePerformance
– processingLatency: 1us
– processingBandwidthReqPS: 100000
Structure Object Diagram: Part 2
Structure Object Diagram: Part 1

Figure 2.7: The representation of the connector of two parts of a diagram. Here the performance descriptions of the *Server1* are located in a separate diagram (Structure Object Diagram: Part 2) and connected to the other diagram (Structure Object Diagram: Part 1) using the *Performance S1* connector.

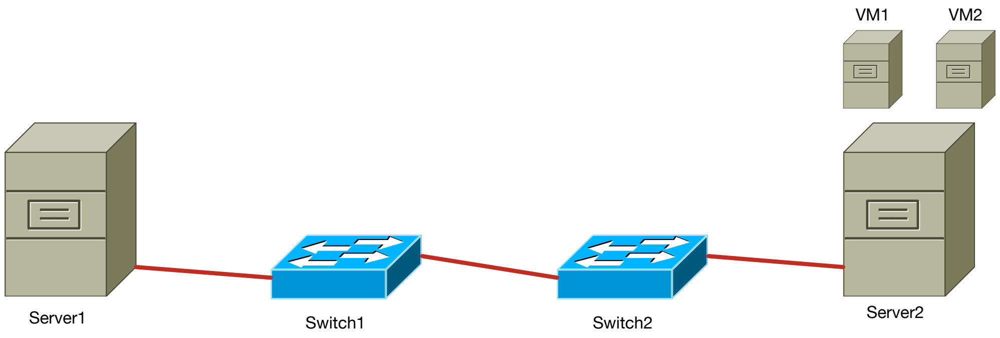

Figure 2.8: The overview on the network structure.

In the case study, we consider two kinds of components: cameras and license plate recognition (LPR). The cameras are located in the city and take pictures of cars that are speeding or entering a paid zone. Each camera is connected to an SBUS component that sends the picture together with a time stamp to the predefined LPR components. The LPR components are deployed in a data center due to their high consumption of computing resources. LPRs receive the pictures emitted by cameras and run a recognition algorithm to identify the license plate numbers of the observed vehicles.

2.2.3 Model of the Network Structure

The modeled system consists of two physical servers: *Server1* and *Server2*. *Server2* is virtualized and hosts two virtual machines *VM1* and *VM2*. The two physical servers are connected with 1G Ethernet data center network. There are two switches located on the network path between *Server1* and *Server2—Switch1* and *Switch1*. The simplified schematic overview of the network structure is presented in Figure 2.8.

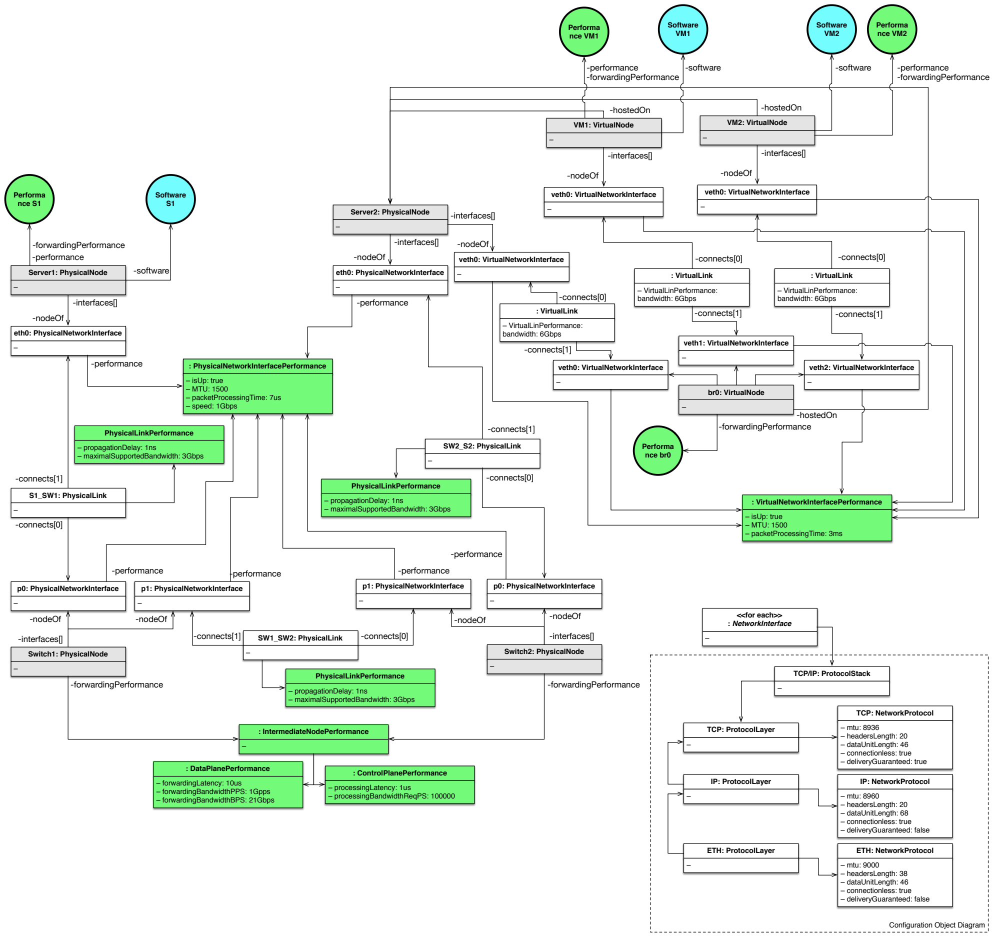

Figure 2.9: The DNI model representing the structure of the network used in the SBUS scenario. For brevity, we presented the relation between each NetworkInterface and Protocol stack separately. The entities enclosed in the dashed frame come from the configuration part of the DNI meta-model.

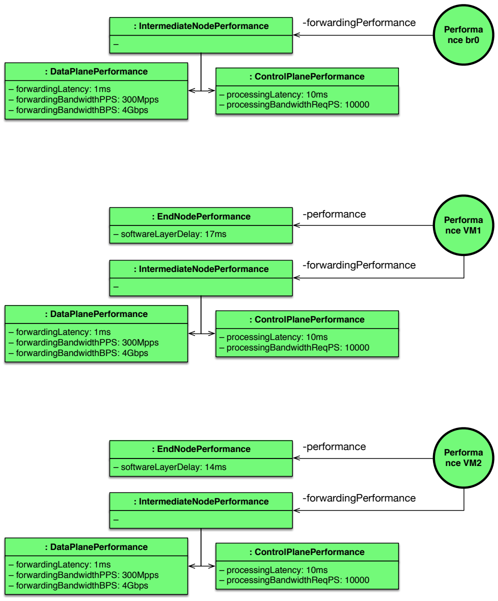
: IntermediateNodePerformance
–
-forwardingPerformance
Performance br0
: DataPlanePerformance
– forwardingLatency: 1ms
– forwardingBandwidthPPS: 300Mpps
– forwardingBandwidthBPS: 4Gbps
: ControlPlanePerformance
– processingLatency: 10ms
– processingBandwidthReqPS: 10000
: EndNodePerformance
– softwareLayerDelay: 17ms
-performance
Performance VM1
: IntermediateNodePerformance
–
-forwardingPerformance
: DataPlanePerformance
– forwardingLatency: 1ms
– forwardingBandwidthPPS: 300Mpps
– forwardingBandwidthBPS: 4Gbps
: ControlPlanePerformance
– processingLatency: 10ms
– processingBandwidthReqPS: 10000
: EndNodePerformance
– softwareLayerDelay: 14ms
-performance
Performance VM2
: IntermediateNodePerformance
–
-forwardingPerformance
: DataPlanePerformance
– forwardingLatency: 1ms
– forwardingBandwidthPPS: 300Mpps
– forwardingBandwidthBPS: 4Gbps
: ControlPlanePerformance
– processingLatency: 10ms
– processingBandwidthReqPS: 10000

Figure 2.10: The continuation of the DNI model presented in Fig. 2.9. Performance descriptions for nodes br0, VM1, and VM2.

2.2.4 Model of the Network Traffic

The overview of the network traffic model is presented in Figure. 2.11, whereas the model in Figure. 2.12. In the example, there are four software components involved: Camera1, Camera2, LPR1 (license plates recognition), and LPR2. The cameras generate traffic by sending pictures to the LPRs. Each camera generates a picture of a given size in regular time periods and sends it to the respective LPR component using the network.

2.2.5 Model of the Network Configuration

The model of the network configuration is presented in Figure 2.13. It consist of two protocol stacks: TCP+IP+Ethernet for node-to-node communication, and IP+Ethernet for inter-switch communication. The two protocol stacks reference the protocols that are specified using the NetworkProtocol classes.

Configuration of the network includes also data about the routes. In Figure 2.13, we present a single route with it list of hops. The other routes in the model are defined analogously.

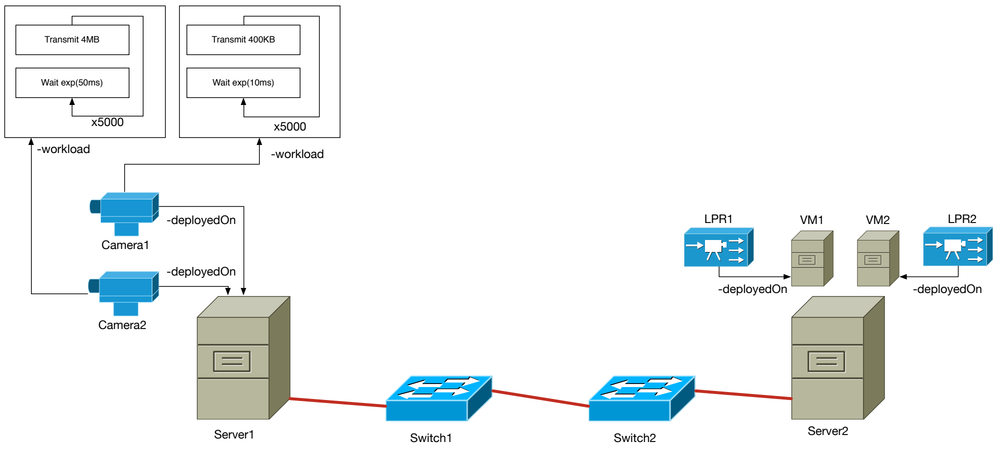

Figure 2.11: The overview on the network traffic. Software components (Camera, LPR) are deployed on nodes and generate traffic according to their workloads.

2.3 miniDNI

Despite the high level of abstraction of DNI, it still requires many parameters to be provided as input. To reduce the amount of input data, we provide a smaller version of the DNI meta-model; we call it *miniDNI*. The entities included in the *miniDNI* are depicted in Figure 2.14.

In the *miniDNI* meta-model, we abstract the following information. First, we abandon the virtual entities (links and nodes) and provide only one generic representation of them. Second, we remove the *NetworkInterfaces*; the information included in a *NetworkInterface* is now merged into the *Link*. Third, we simplify the descriptions of the traffic sources and workflows. In *miniDNI*, a workflow specifies only a size of a message and the number of messages per second without defining what a message actually is. Finally, we abstract out all information about protocols used in the network. From the *DNI*'s *NetworkConfiguration*, we keep only simplified information about routes in the network. The routes are flow-based, which means that there is at least one route defined for every traffic source (in contrary to the classical approach, where routes are defined for all source-destination pairs).

2.4 Example of a miniDNI Model: SBUS

The model presented in Figure 2.15 represents the same network, as the model described in Section 2.2.

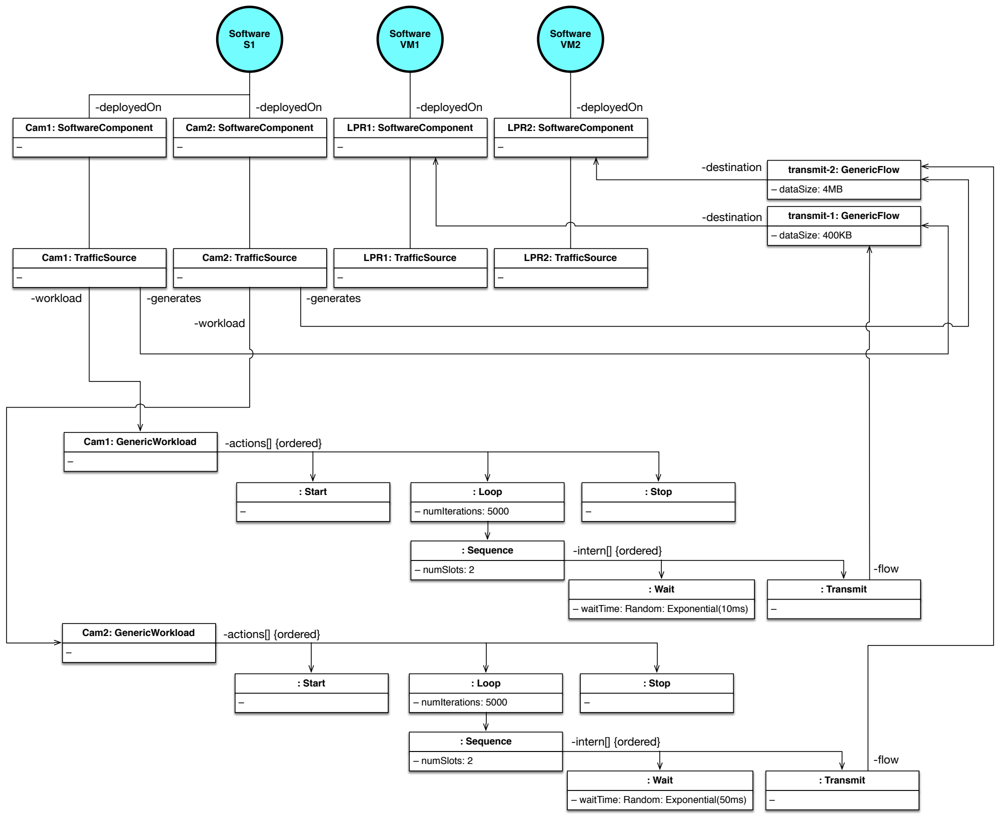

Figure 2.12: The DNI model representing the traffic in the network used in the SBUS scenario.

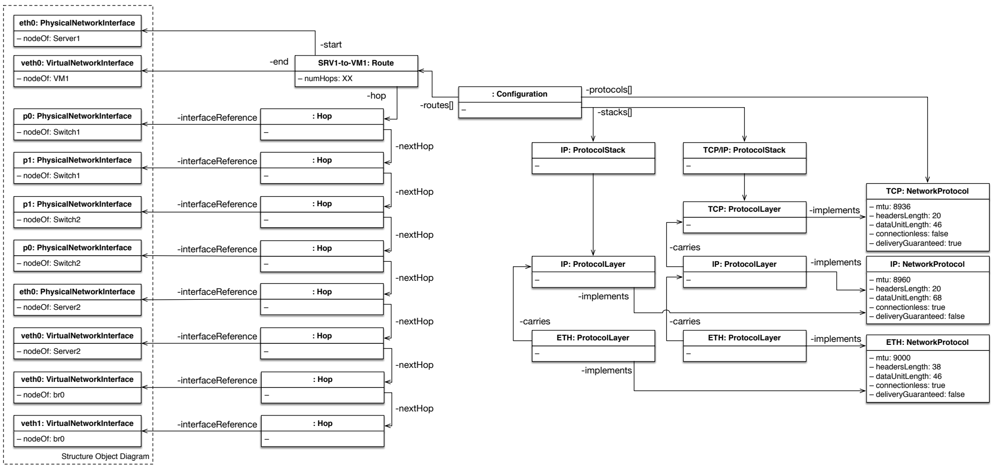

Figure 2.13: The DNI model representing the configuration of the network used in the SBUS scenario.

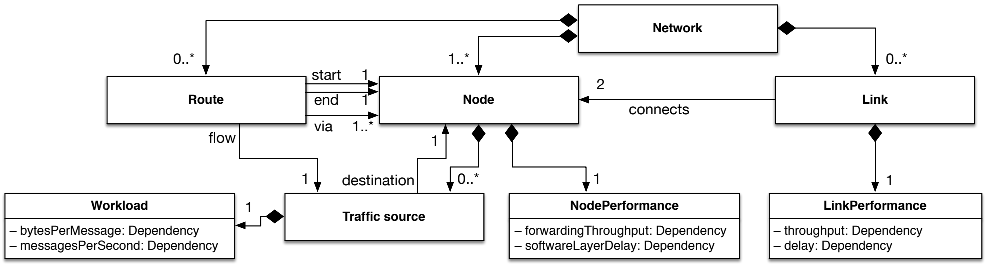

Figure 2.14: *miniDNI* meta-model

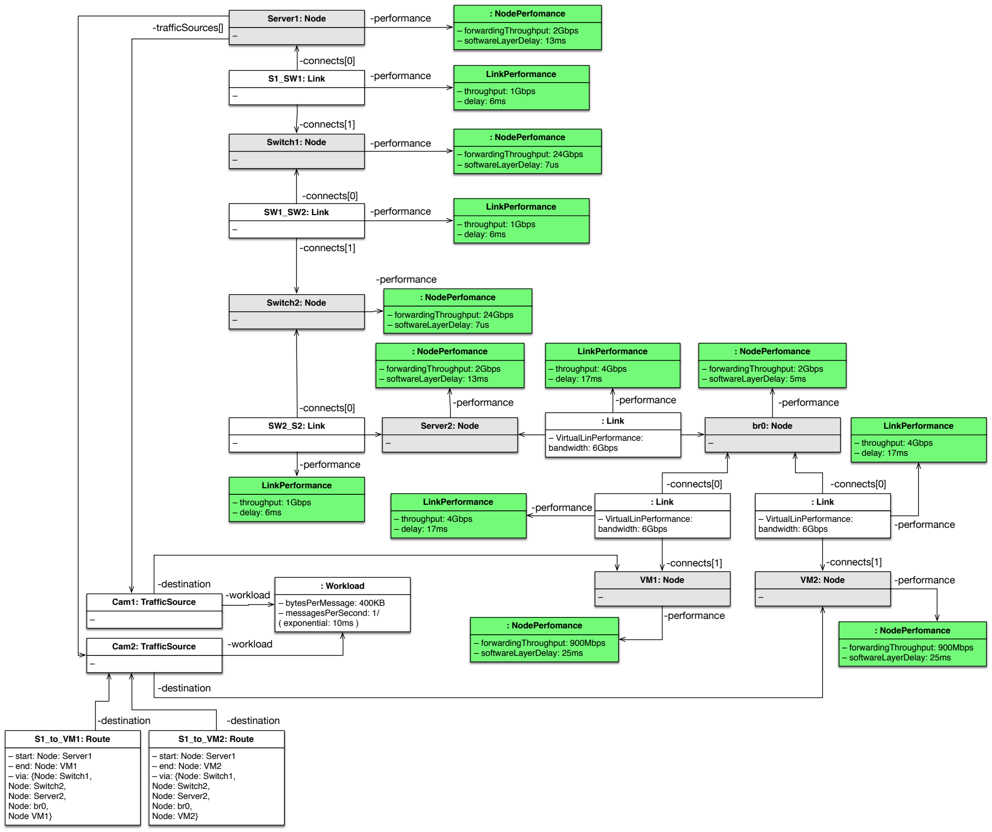

Figure 2.15: The miniDNI model representing the network used in the SBUS scenario.

Chapter 3

Transformations

3.1 Map of the Transformations

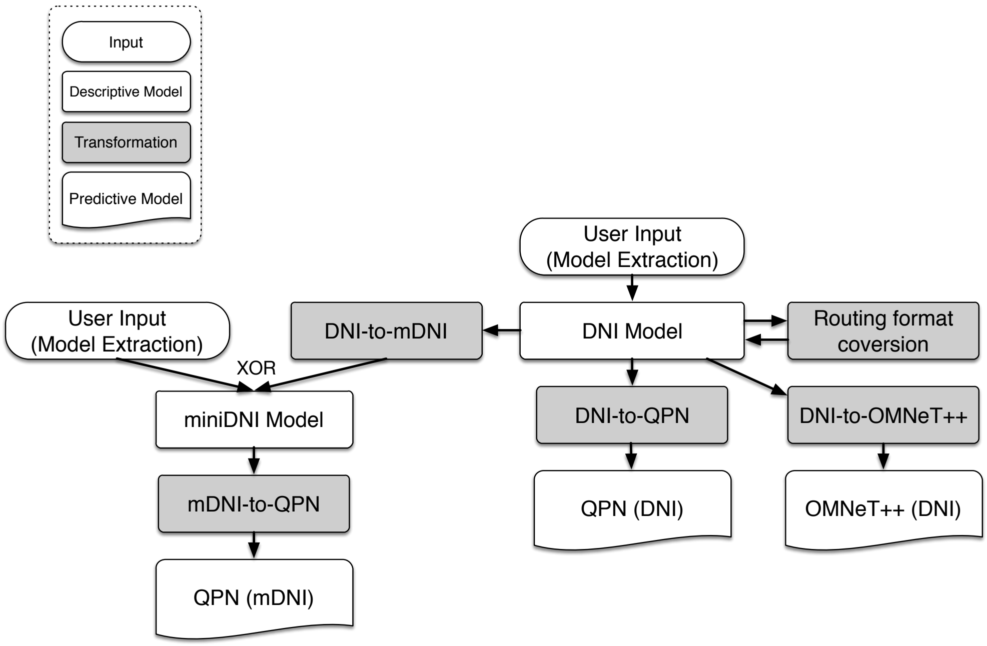

Figure 3.1: Models and model transformations

3.1.1 General Remarks

We use the Epsilon Transformation Language (ETL) [KPP08] for transforming the models into other models. According to, [DK14] „ETL has been designed as a hybrid language that implements a task-specific rule definition and execution scheme but also inherits the imperative features of EOL to handle complex transformations where this is deemed necessary."

The transformations are executed in the following order: pre-blocks, rules, post-blocks. The pre-blocks are used for preparation tasks, post-blocks for final linking after the transformation is ready and the rules transform the given source object into the destination object. In the descriptions of the transformations, we refer to the pre and post transformation blocks because not every transformation can be handled by the transformation rules.

3.2 DNI-to-OMNeT++

3.2.1 Pre-transformation Operations

In the *DNI-to-OMNeT++* (short *DNI-to-OMNeT*) transformation, the pre-blocks contains the predefined names of object modules that should be applied to respective DNI entities in case a direct mapping in ambiguous.

3.2.2 Transformation Rules

The rules execute in the order of appearing.

Rule DNI2ned

In this rule, a global Ned file is added to the OMNeT model, global variables (for OMNeT scope) are declared, ands the necessary OMNeT imports are added to enable the use of OMNeT modules form the INET library.

Rule NetworkInfrastructure2Network

In this rule, we add and configure the OMNeT Network to the NedFile. Additionally, the IPNetworkConfigurator module is added to the OMNeT Network.

Rule PhysicalEndNode2Network

This rule is executed for all DNI Nodes that **does host** other nodes and have the performance description of type EndNodePerformanceSpecification.

For every PhysicalNode a new OMNeT (sub-)Network is generated. That network contains the following OMNeT Modules: VMM bridge (of type as defined globally—currently EtherSwitch), the VMM host (e.g., dom0 in Xen) of type StandardHost, set of StandardHosts representing the VirtualNodes that are hosted on the given PhysicalNode.

In OMNeT one does not specify the performance of a NetworkInterface (except of specifying its type that depends on the protocol of the second layer). The performance is encoded in communication channels whereas the contention that happens in interfaces is modeled internally in the respective OMNeT module that represents the NetworkInterface. The resulting sub-network with single VirtualNode, a VMM bridge and VMM host is presented graphically in Fig. 3.2. The respective ned code is listed in Listing 3.1.

```
network type_relate6
{
    @display("bgb=246,167");
    gates:
        inout ethg[];

    types:
        channel IDCHANNEL__KD7zsBmlEeSnOoFTf1SkPw extends DNIThruputMeteringChannel
        {
            //arificial 1 gig channel
```

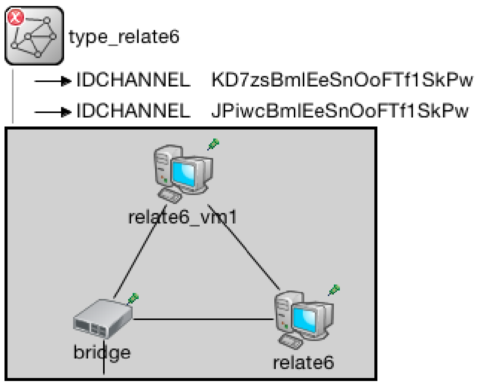
type_relate6
IDCHANNEL KD7zsBmlEeSnOoFTf1SkPw
IDCHANNEL JPiwcBmlEeSnOoFTf1SkPw
relate6_vm1
bridge
relate6

Figure 3.2: OMNeT representation of a hypervisor (PhysicalEndNode that hosts VirtualNodes).

```
            datarate = 1.0Gbps;
            thruputDisplayFormat = "b";
            delay = 1.0E-7s;
        }

        channel IDCHANNEL__JPjwcBmlEeSnOoFTf1SkPw extends DNIThruputMeteringChannel
        {
            //relate6 iface veth0<--->relate6_vm1 iface veth0
            datarate = 10000.0Mbps;
            thruputDisplayFormat = "b";
            delay = 9.99999993922529E-9s;
        }

    submodules:
        bridge: EtherSwitch {}
        relate6: StandardHost {}
        relate6_vm1: StandardHost {}
    connections allowunconnected:
        bridge.ethg++ <--> ethg++;
        bridge.ethg++ <--> IDCHANNEL__KD7zsBmlEeSnOoFTf1SkPw <--> relate6.ethg++;
        bridge.ethg++ <--> IDCHANNEL__KD7zsBmlEeSnOoFTf1SkPw <--> relate6_vm1.ethg++;
        relate6.ethg++ <--> IDCHANNEL__JPjwcBmlEeSnOoFTf1SkPw <--> relate6_vm1.ethg++;

}
```

Listing 3.1: Ned languange representation of a PhysicalEndNode hosting VirtualNodes

Rule PhysicalLink2Channel

This rule is executed for all DNI PhysicalLinks. In OMNeT, we create a new Channel (meta class for a connection) for every link to be able to specify its performance-related parameters. The Channel description is added to the OMNeT Network.

Rule VirtualLink2Channel

This rule is executed for all DNI VirtualLinks. In OMNeT, we create a new Channel (meta class for a connection) for every link to be able to specify its performance-related parameters. The created channel is added to a respective sub-network that represents a given hypervisor (create using the rule PhysicalEndNode2Network).

Rule PhysicalEndNode2Submodule

This rule is executed for all DNI Nodes that **do not host** other nodes and have the performance description of type EndNodePerformanceSpecification. In that case, the StandardHost module is created for the given PhysicalNode.

Rule PhysicalIntermediateNode2Submodule

This rule is executed for all DNI Nodes that have the performance description of type IntermediateNodePerformanceSpecification or the type of the performance description cannot be derived.

Based on the NetworkProtocol used by the NetworkInterfaces of that Node, an EtherSwitch (protocol has 2 layers), Router (protocol has 3 layers), or StandardHost (protocol has >3 layers) module is created. In case the performance description is of unknown type, an EtherSwitch is generated to provide the basic connectivity.

Rule VirtualEndNode2Submodule

This rule in its current form annotates the VMs that are hosted in a hypervisor. This rule will be removed in future versions of the transformations as it was replaced by more complex rule PhysicalEndNode2Network.

Rule VirtualIntermediateNodeNode2Submodule

This rule creates a new EtherSwitch (or Router, or StandardHost respectively, based on the NetworkProtocol used by the NetworkInterfaces) inside the sub-network representing a hypervisor.

Rule Link2ChannelConnectionItem

One all nodes of the OMNeT network are ready (sub-networks, StandardHosts, Routers, or EtherSwitches), the nodes can be connected. Proper connections between interfaces are added in the *connections* section of every OMNeT network. The names of the interfaces are not transformed from DNI to OMNeT! We use the default names (e.g., *ethg*) with a numerical postfix, because the names have semantics that are already encoded inside the INET library (e.g., the StandartHost must have interfaces named ethg). If custom names of interfaces are required, then the code of modules provided in the INET library must be adapted manually.

For every DNI Link, a ChannelConnectionItem is created. The ChannelConnectionItem instantiates respecite Channel generated before by the Physical-/Virtual-Link2Channel rule.

Rule PhysicalLink2ChannelConnectionItem (extends Link2ChannelConnectionItem)

The rule adds the ChannelConnectionitem (created in super-rule Link2ChannelConnectionItem) to the connections section of the OMNeT NedFile.

Rule VirtualLink2ChannelConnectionItem (extends Link2ChannelConnectionItem)

The rule adds the ChannelConnectionitem (created in super-rule Link2ChannelConnectionItem) to the connections section of the proper sub-network in the OMNeT NedFile.

Rules DNI2IniFile + DNI2IniGroup + DNI2RoutingFile

All the rules of the *DNI-to-OMNeT* presented up to that point are responsible for generating the NedFile of the OMNeT simulation. The configuration and the parameters of the simulation are stored in configuration files: omnetpp.ini (configuration and parameters) and ipconfig.xml (routing and addressing).

The three rules generate the missing configuration files. Technically the rules call specified operations of the transformation language, because the procedure involves creation of multiple detailed objects what would be difficult to implement in a rule. Moreover, the meta-model that specifies these configuration files is simplified.

To transform the remainder of the model, the following operations are executed:

1. For all receiving nodes, receiver applications are added. We use INET model of *TCPSinkApp* for this purpose. We set the respective configuration of the omnetpp.ini file by using code as shown in Listing 3.2.

2. For all generator nodes, sending applications are added. We use INET model of *DNITCPSessionApp* that is a modified version of the *TCPSessionApp* available in INET. The modifications concern only the change of format that is passed to the *.sendScript* parameter—we add a new *.specialSendScript* that is much shorter than the original representation, for example: `"send 1265000 every 0.01 for 5000 times;"`, which means send 1265000 bytes every 0.01 seconds; repeat 5000 times. The original *.sendScript* representation of this example would be presented as a long list of pairs (time + data). Example: `"0.0 1265000; 0.01 1265000; 0.02 1265000; ... 49.99 1265000;`. Exemplary configuration of a sending application is presented in Listing 3.3.

3. To produce the *.specialSendScript*, every workload of each traffic source is converted to a respective time series (pairs of tuples: time, data). For LoopActions, one can benefit of the compact textual representation of the *.specialSendScript* parameter.

4. For every NetworkInterface, the protocol addresses are extracted and stored in the ipconfig.xml file. An exemplary content of the ipconfig.xml file that describes interfaces addressing is shown in Listing 3.4.

5. For every Route, the list of hops is extracted and stored in the ipconfig.xml file. An exemplary content of the ipconfig.xml file that describes routing is shown in Listing 3.5.

```
###################################
##### config of node relate4_vm1
###################################
**.relate4_vm1.numTcpApps = 1
##### RCV APP. App id 0
**.relate4_vm1.tcpApp[0].typename = "TCPSinkApp"
**.relate4_vm1.tcpApp[0].localPort = 1000
```

Listing 3.2: Fragment of the omnetpp.ini file. Description of a receiver application.

```
###################################
##### config of node relate2
###################################
**.relate2.numTcpApps = 2
##### SEND APP. App id 0
**.relate2.tcpApp[0].typename = "DNITCPSessionApp"
**.relate2.tcpApp[0].casual_name = "TS_Cam2to4vm1 ==> lpr411 on relate4.relate4_vm1"
**.relate2.tcpApp[0].connectAddress = "relate4.relate4_vm1"
**.relate2.tcpApp[0].connectPort = 1000
**.relate2.tcpApp[0].tOpen = 0s
**.relate2.tcpApp[0].tSend = 0s
**.relate2.tcpApp[0].sendBytes = 0B
**.relate2.tcpApp[0].tClose = 0s
**.relate2.tcpApp[0].specialSendScript = "send 1265000 every 0.1 for 5000 times;"

##### SEND APP. App id 1
**.relate2.tcpApp[1].typename = "DNITCPSessionApp"
**.relate2.tcpApp[1].casual_name = "TS_Cam2to5vm1 ==> lpr511 on relate5.relate5_vm1"
**.relate2.tcpApp[1].connectAddress = "relate5.relate5_vm1"
**.relate2.tcpApp[1].connectPort = 1000
**.relate2.tcpApp[1].tOpen = 0s
**.relate2.tcpApp[1].tSend = 0s
**.relate2.tcpApp[1].sendBytes = 0B
**.relate2.tcpApp[1].tClose = 0s
**.relate2.tcpApp[1].specialSendScript = "send 1265000 every 0.1 for 5000 times;"
```

Listing 3.3: Fragment of the omnetpp.ini file. Description of a node that has two sender applications.

```
<interface hosts='*HL4' names='eth0' address='10.0.1.4' netmask='255.255.255.0' metric='20'/>
<interface hosts='*HL5' names='eth0' address='10.0.1.5' netmask='255.255.255.0' metric='20'/>
<interface hosts='*HR0' names='eth0' address='10.0.100.0' netmask='255.255.255.0' metric='20'/>
<interface hosts='*HR1' names='eth0' address='10.0.100.1' netmask='255.255.255.0' metric='20'/>
```

Listing 3.4: Fragment of the ipconfig.xml file. Description of addresses of interfaces.

```
<route
  hosts='relate2'
  destination='10.0.1.6'
  netmask='255.255.255.0'
  gateway = '*'
  interface='eth0'
  metric='2'/>
<route
  hosts='relate6'
  destination='10.0.1.2'
  netmask='255.255.255.0'
  gateway = '*'
  interface='eth0'
  metric='2'/>
```

Listing 3.5: Fragment of the ipconfig.xml file. Description of routes.

Rule NetworkInfrastructure2Network

1. Determine the DNI Flows that are used in the DNI Workloads for the given DNI TrafficSource.
2. For each Flow, estimate the amount of data and the total wait time using the following procedure:
   a) For the workload graph (represented as a set of Actions connected with edges that represent the order), apply the GraphFold procedure [RS10] to calculate the total time of the execution of the DNI Workload.
   b) For the workload graph, sum the amount of data that is produced by the given DNI TrafficSource
3. Divide the amount of the transmitted data by the duration of the workload (in second).
4. For the miniDNI TrafficSource, set the values of the parameters: bytesPerMessage to the average amount of data sent in a second, and the messagesPerSecond to 1;
5. Set the source and destination node of the flows exactly as in the DNI model.
6. Add the generate TrafficSource entity to the root element of the miniDNI.

3.2.3 Post-transformation Operations

In the *DNI-to-OMNeT* transformation two post-blocks are used. First is responsible for renaming the entities (character „-” is not allowed in OMNeT and is replaced by „_”). Second post-block adds the OMNeT Network to the NedFile entity and adds the respective configuration files (routing and omnetpp.ini).

3.3 DNI-to-miniDNI

3.3.1 Pre-transformation Operations

In the *DNI-to-miniDNI* transformation, no the pre-blocks are used.

3.3.2 Transformation Rules

The rules execute in the order of appearing.

Rule NetworkInfrastructure2Network

This is a root-rule that creates the Network entity and assigns the data center name to it.

Rule Node2Node

Every DNI Node is translated to a respective miniDNI Node. In this rule, the Node object is crated and the default values describing the performance are assigned.

The performance of the Node is calculated as follows:

- softwareLayerDelay of miniDNI = softwareLayerDelay of the EndNodePerformance in DNI if the DNI node is EndNode
- softwareLayerDelay of miniDNI = forwardingLatency of the DataPlanePerformance in DNI if the DNI node is IntermediateNode
- forwardingThroughput of miniDNI = minimum of the following values:
  - forwardingBandwidthBPS of DataPlanePerformance
  - deriverd bandwidth in bytesPerSecond for the average data unit size (e.g., MTU for Ethernet frame) and the forwardingBandwidthPPS parameter of the DataPlanePerformance. Example: for average MTU = 9000 bytes, and forwardingBandwidthPPS: 300Mpps, the forwardingThroughput equals 270000 MBps (megabytes per second) = 21 Tbps (tera bits per second).
  - deriverd bandwidth in bytesPerSecond for the average data unit size (e.g., MTU for Ethernet frame) and the forwardingLatency parameter of the DataPlanePerformance. Example: for average MTU = 9000 bytes, and forwardingLatency: 1ms, the forwardingThroughput equals: 9000 * 1s/forwardingLatency = 9 MBps (megabytes per second) = 72 Mbps (mega bits per second).

Rule PhysicalNode2Node (extends Node2Node)

In this rule adds the generated Node to the root element of the miniDNI. Except the performance description, no other assignments are made.

Rule VirtualNode2Node (extends Node2Node)

In this rule adds the generated Node to the root element of the miniDNI. Except the performance description, no other assignments are made. The miniDNI model does not distinguish the physical and the virtual nature of a DNI Node.

Rule Link2Link

To transform the DNI Link into the miniDNI Link, the following procedure is executed.

1. We take the DNI Link and the NetworkInterfaces that it connects.
2. The nodes of the DNI NetworkInterfaces are determined.
3. The miniDNI Link is set to connect the nodes.
4. The LinkPerformance entity is calculated as sum of the: propagation delay of the DNI Link, the packetProcessingTime parameter of the first NetworkInterface, and the packetProcessingTime of the second NetworkInterface.

Rule Physical2Link (extends Link2Link)

In this rule adds the generated Link to the root element of the miniDNI. Except the performance description, no other assignments are made. The miniDNI model does not distinguish the physical and the virtual nature of a DNI Link.

Rule Virtual2Link (extends Link2Link)

In this rule adds the generated Link to the root element of the miniDNI. Except the performance description, no other assignments are made. The miniDNI model does not distinguish the physical and the virtual nature of a DNI Link.

Rule TrafficSource2TrafficSource

For each DNI TrafficSource the following procedure is applied:

1. Determine the DNI Flows that are used in the DNI Workloads for the given DNI TrafficSource.
2. For each Flow, estimate the amount of data and the total wait time using the following procedure:
   a) For the workload graph (represented as a set of Actions connected with edges that represent the order), apply the GraphFold procedure [RS10] to calculate the total time of the execution of the DNI Workload.
   b) For the workload graph, sum the amount of data that is produced by the given DNI TrafficSource

3. Divide the amount of the transmitted data by the duration of the workload (in second).
4. For the miniDNI TrafficSource, set the values of the parameters: bytesPerMessage to the average amount of data sent in a second, and the messagesPerSecond to 1;
5. Set the source and destination node of the flows exactly as in the DNI model.
6. Add the generate TrafficSource entity to the root element of the miniDNI.

Rule FlowRoute2Route

This rule transforms a single DNI route in the FlowRoute format (see Fig. 2.4) to a single miniDNI Route. In the DNI, each FlowRoute is defined as a list of interfaces that need to be traversed to get form source node to the destination. In the miniDNI, each route is a ordered set of nodes to be traversed. The transformation procedure traverses the DNI route and selects the node that belongs to a given interface referenced in each Hop element. The List of nodes is stored and the duplicate nodes that appear one-by-one are removed as this redundancy is not needed; on a given node, for each flow, there is one receive (rx) and one transmit (tx) interface that both belong to the same node. This causes that there may appear two nodes in a row.

3.3.3 Post-transformation Operations

In the *DNI-to-miniDNI* transformation, no the post-blocks are used.

3.4 DNI-to-QPN

3.4.1 Pre-transformation Operations

In the *DNI-to-QPN* transformation, the pre-block is used for declaration of global variables that need to be accessible for the whole transformation, e.g., mapping of DNI-flows to QPN-colors, defining global QPN-colors (ether-color).

3.4.2 Transformation Rules

The rules execute in the order of appearing.

Rule DNI2QPN

This is a root-rule that creates an empty QpmeDocument and initializes containers for queues, places, colors, transitions.

Rule DNI2QPN-findColors

This is a preconfiguration-rule that reads the DNI model and searches for *Flow* instances. For every instance found, a QPMEColor is generated and saved. The mapping of DNI-flows to QPN-colors is stored in global variables. Additionally, the unique global color called *ether* is generated and stored.

Rule IntermediateNode2Place

In this rule, a SubnetPlace is generated for every instance of the IntermediateNode. The actions are executed in the following order.

1. The constant Input and Output places are added to the Subnet. The ether color is assigned to the places.
2. The following transitions are added to the subnet: input, output, switching. The input transition is connected to the input place and the output transition to the output place.
3. For each NetworkInterface of the IntermediateNode, a pair of QueueingPlaces is created (tx and rx).
4. We gather all colors that represent the flows traversing the Node.
5. For each traversing color a color reference is added to the places: all tx-, and rx-places, input, output, the subnet.
6. The rx and tx places with no color references are removed (no traffic flows through the interfaces).
7. We add mode to the input transition that consumes one token of color *ether* and deletes it.

8. We add mode to the input transition that consumes one token of every traversing color and forwards it to the respective rx-place. The selection of the place (=network interface) is taken directly from the routing information.
9. For every traversing color, for every rx place, for every tx place a mode is created in the *switching* transition if the traversing color is routed through the given pair of rx/tx interfaces. The exemplary modes of the *switching* transition are presented in Figure 3.4.
10. For every traversing color a new mode is added to the output transition if the preceding tx place has the reference to the given color (i.e., if the given flow is transmitted using the respective tx interface).
11. The value of the delay parameter for the rx- and tx- queueing places is calculated
    a) Flow size is calculated based on the parameters of the *Flow* entity in the DNI model
    b) Interface maximal throughput is calculated based on the interface performance description
    c) Protocols overheads are added to the flow size
    d) Processing delay is calculated as bruttoFlowSize/interfaceMaxThroughput
    e) Switching delays (described in IntermediateNodePerformance) are added to the delays of the rx queueing places.
    f) If the parameters are modeled as probabilistic functions, then the respective average values are taken for the calculation (transformation without this simplification is possible as an extension)

Rule EndNode2Place

In this rule, a SubnetPlace is generated for every instance of the EndNodeNode. The actions are executed in this rule are similar to the IntermediateNode2Place. The main difference is lack of the *switching* transition and the presence of the traffic sources. In the list of transformations steps, we focus on the differences.

We distinguish four types of an EndNode: Generator (produces the traffic), Receiver (receives the traffic), Traversal (acts as IntermediateNode), Hypervisor (hosts VirtualNodes). The rule is divided into four parts that implement the functionalities related to the respective node types.

1. Determine which subset of types apply for this node.
2. For traversal node.
    a) Between the rx and tx queueing places, add Bridge QueueingPlace. Connect the rx ququeing places to the Bridge queueing place using a new transition called RXSW. Repeat for SWTX transition for connecting the Bridge with the tx queueing places.

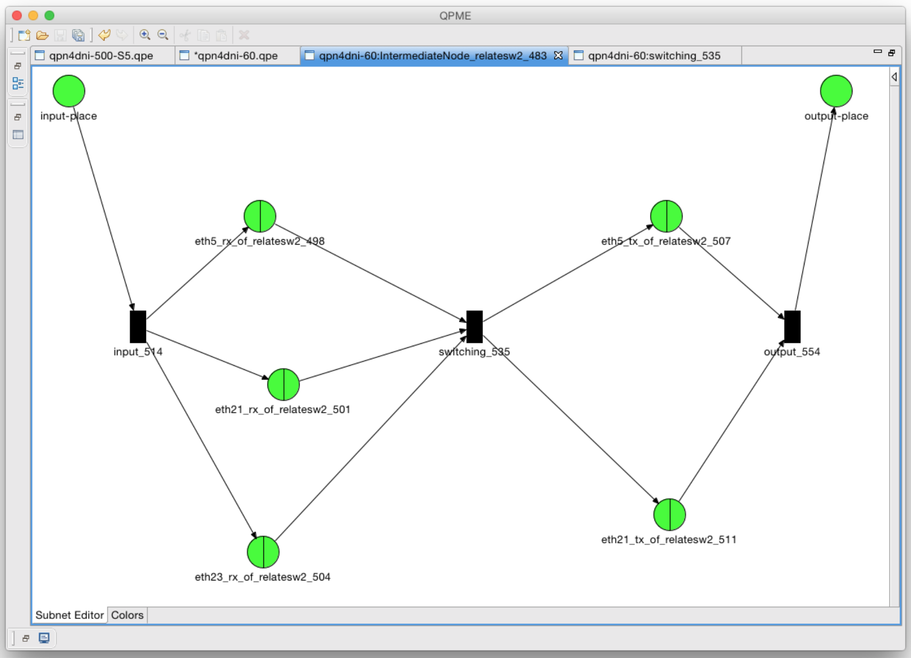

Figure 3.3: Example of a QPN representation of an IntermediateNode.

b) Using the routing information, assign the respective traversing colors to the proper pairs of rx, tx queueing places.

c) For each traversing color, add reference to the SubnetPlace, input, output and bridge place.

d) Using the routing information (get the rx place for a given color), add a new mode to the input transition. The tokens of that color will be routed through the given rx place.

e) Repeat for the RXSW, SWTX, and output transitions.

3. For generator node.

   a) Get all colors for which the given node servers as beginning of route (starting colors).

   b) For each starting color create a new TrafficSource SubnetPlace. Connect the subnet place to the RXSW and the SWTX transitions.

   c) Using the routing information, assign the respective starting colors to the respective tx queueing places.

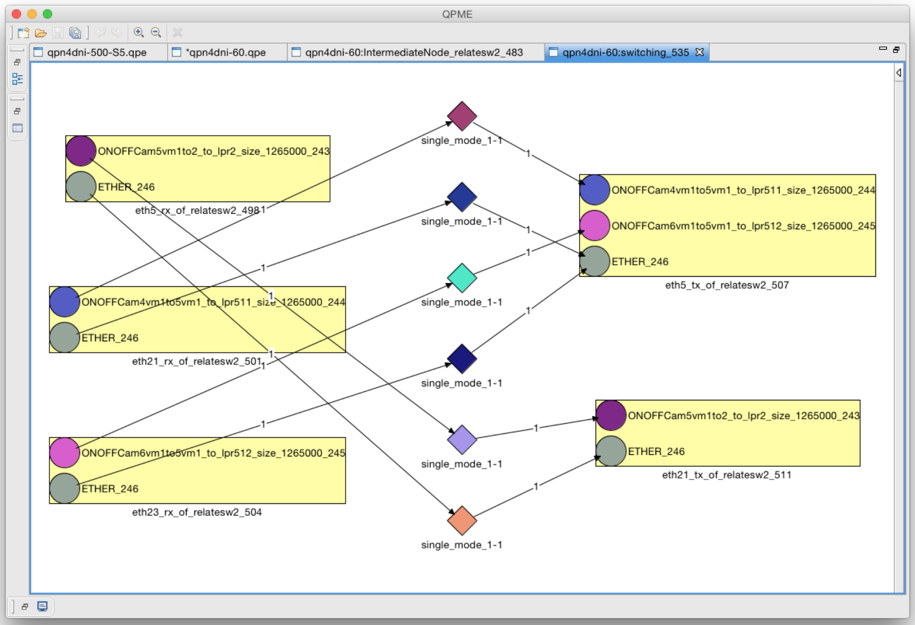
QPME
qpn4dni-500-S5.qpe
*qpn4dni-60.qpe
qpn4dni-60:IntermediateNode_relatesw2_483
qpn4dni-60:switching_535
ONOFFCam5vm1to2_to_lpr2_size_1265000_243
ETHER_246
eth5_rx_of_relatesw2_498
ONOFFCam4vm1to5vm1_to_lpr511_size_1265000_244
ETHER_246
eth21_rx_of_relatesw2_501
ONOFFCam6vm1to5vm1_to_lpr512_size_1265000_245
ETHER_246
eth23_rx_of_relatesw2_504
single_mode_1-1
single_mode_1-1
single_mode_1-1
single_mode_1-1
single_mode_1-1
single_mode_1-1
ONOFFCam4vm1to5vm1_to_lpr511_size_1265000_244
ONOFFCam6vm1to5vm1_to_lpr512_size_1265000_245
ETHER_246
eth5_tx_of_relatesw2_507
ONOFFCam5vm1to2_to_lpr2_size_1265000_243
ETHER_246
eth21_tx_of_relatesw2_511

Figure 3.4: The modes for an exemplary switching transition of an IntermediateNode.

d) Add a new mode to the RXSW transition to connect the tx queuing places with all traffic sources. Set the mode to consume one ether token, produce on ether token, and to produce zero tokens for each starting color and traffic source (see Fig. 3.5). This step is required for the QPN to be valid and has no influence on the simulation.

e) For each starting color, add a new mode to the SWTX transition to connect the TrafficSource with the respective tx queueing place.

f) For each starting color, add a new mode to the output transition to connect the respective tx queueing place with the output place.

4. For receiver node.

a) Get all colors for which the given node servers as route termination (ending colors).

b) For each ending color add the color references to the SubnetPlace, input, and the dummy place.

c) For each ending color add a new mode to the input transition.

d) For each ending color add a new mode to the RXSW transition. Set the mode to remove all incoming tokens.

5. For hypervisor node.
   a) Based on routing information derive starting and ending colors of the VirtualNodes that are hosted on the hypervisor node.
   b) Derive colors that start and end in the virtual nodes hosted on this node (inner-node-traffic).
   c) Create VMMIntermediateNode SubnetPlace. Apply the rule IntermediateNode2Place. Connect the VMMIntermediateNode place to the RXSW and SWTX transitions.
   d) Add new mode to the RXSW transition so that the colors that end in the virtual nodes are directed to the VMMIntermediateNode.
   e) Add new mode to the SWTX transition so that the colors that start in the virtual nodes and do not end in the virtual nodes are directed to the proper tx queueing place.
   f) Create the virtual nodes SubnetPlaces. Apply the rule EndNode2Place or IntermediateNode2Place based on the performance descriptions of the virtual nodes.
   g) For the connection between the virtual node and the VMMIntermediateNode apply the rule Link2Transitions. Refer to Figure 3.6 for graphical representation of the hypervisor node.
6. Calculate delays for the queueing places analogously to the procedure for the IntermediateNode.
7. Add the SubnetPlace to the QPMEDocument.

Rule Link2Transitions

In this rule, a pair of transitions is generated. Each transition forwards the tokes from source SubnetPlace to the destination SubnetPlace—each transition in opposite direction. We assume, that the contention in the network happen in the software or in the network interfaces. Thus, we model links as a pair of immediate transitions (one in each direction).

1. Based on the routing information, determine the colors that traversing the given link. For each color, determine direction. Assume abstract directions: e.g., L, R. (SubnetPlace1 $\rightarrow$ SubnetPlace2 $==$ R, otherwise L).
2. For each color traversing the link in R direction, add a new mode to the R transition. The mode consumes single token on the input and generates single token on the output.
3. For each color traversing the link in L direction, add a new mode to the L transition. The mode is configured exactly as in R transition.

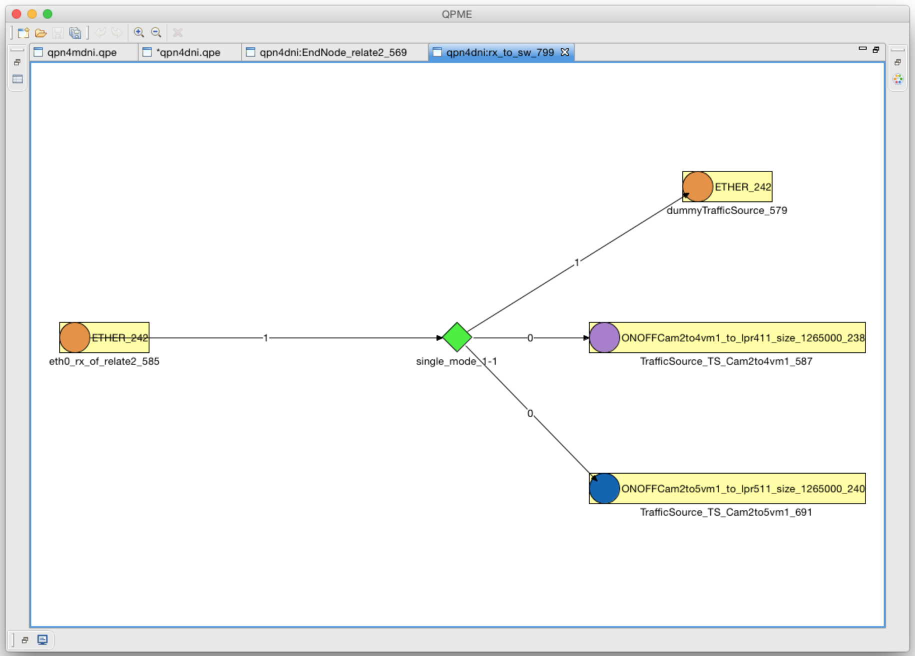
QPME
qpn4mdni.qpe
*qpn4dni.qpe
qpn4dni:EndNode_relate2_569
qpn4dni:rx_to_sw_799
ETHER_242
dummyTrafficSource_579
1
ETHER_242
eth0_rx_of_relate2_585
1
single_mode_1-1
0
ONOFFCam2to4vm1_to_lpr411_size_1265000_238
TrafficSource_TS_Cam2to4vm1_587
0
ONOFFCam2to5vm1_to_lpr511_size_1265000_240
TrafficSource_TS_Cam2to5vm1_691

Figure 3.5: The mode of the RXSW transition for a generator node. The incoming ether tokens do no trigger firing of the tokens that represent traffic.

Operation createTrafficSourceSubnet

This operation creates a new SubnetPlace that represents a traffic source. The Subnet-Place is used later in the EndNode2Place rule (by nodes of type generator). The steps are the following.

1. Based on the traffic model, derive the node on which the traffic source is deployed.
2. Derive the color for the traffic that is generated in the given traffic source (generated color).
3. Create new WorkloadControl color.
4. For the ending colors of this node, add color references to the input place of the traffic source. Add respective modes to the input transition to destroy all incoming tokens.
5. Add the reference to the generated color to the output place. Add respective mode to the output transition. Set the mode to consume single token and produce single token.

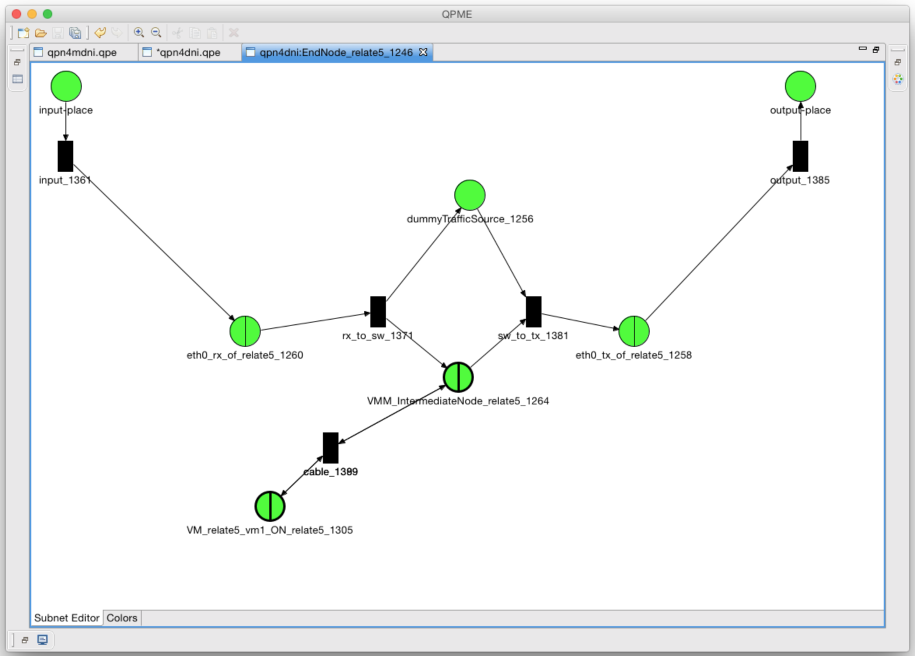

Figure 3.6: The QPN representation of a PhysicalNode that hosts a single VirtualNode (named VM_relate5_vm1_ON_relate5). The VMMSwitch (VMM_IntermediateNode) is added between the RXSW and SWTX transitions.

6. Create WorkloadControl queueing place. Add reference to the WorkloadControl color. Set initial marking to 1 token of the WorkloadControl color.

7. Add a new mode to the input transition to destroy the tokens with WorkloadControl color.

8. Add ordinary places WorkloadStart and WorkloadStop. Connect them with transitions (called dummy) to the WorkloadControl queueing place. In each transition add a single mode that passes the tokens with WorkloadControl color.

9. Derive the workload actions based on the WorkloadDescription. Follow the order of the actions stored in the DNI model.

10. For StartAction.

    a) Ignore all actions that are preceding the StartAction. StartAction is also ignored.

11. For StartAction.

   a) Ignore all actions that are preceding by the StopAction. StopAction is also ignored.

12. For WaitAction.
    a) Create new queueing place that hold the WorkloadControl token for the time defined in the WaitAction entity.
13. For TransmitAction.
    a) Create new transition with a single mode. The transition consumes a single WorkloadControl token, and produces simultaneously two tokens. First, the consumed WorkloadControl that is passed to the next abstract action in the workload. Second, the token representing the traffic (see example in Fig. 3.7). Connect the transition to the output place of the SubnetPlace.
14. For BranchAction.
    a) Add a new transition that consumes the token generation color and produces single token generation color on each of the outputs. The number of outputs equals the number of branches.
    b) On each branch add a dummy ordinary place.
    c) On end of each branch add a transition that joins all branches. Add new mode that consumes a single token on all inputs of the transition and produces a single token on the output (synchronization point).
    d) For each branch process its sub-actions recursively.
15. For SequenceAction.
    a) Get the sub-actions.
    b) Process sub-actions recursively.
    c) Each sub-action should be connected with the preceding action using a dummy transition that passes the token generation color.
16. For LoopAction.
    a) Create new Loop SubnetPlace according to the createLoopSubnet operation.
    b) Connect the Loop SubnetPlace with the preceding action using a dummy transition that passes the token generation color.
    c) Connect the Loop SubnetPlace with the output transition for all colors different to the token generation color.

Operation createLoopSubnet

The Loop SubnetPlace is created to repeat the given sub-workload for a given amount of repetitions. At the input of the Loop, the single WorkloadControl token is transformed into X LoopControlTokens, where X equals to the number of iterations of the loop.

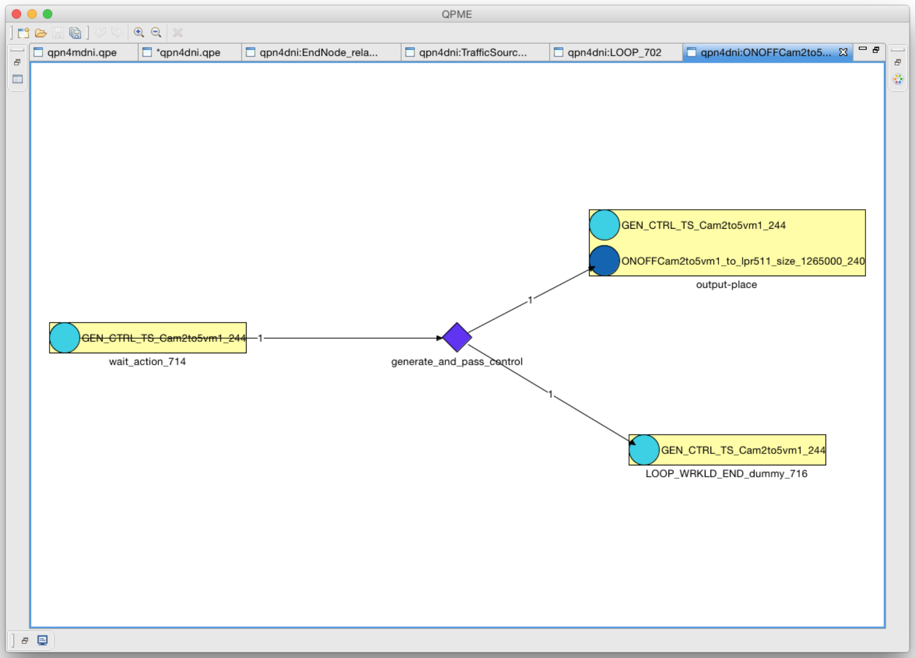

Figure 3.7: The QPN representation of a TransmitAction. A single WorkloadControl token triggers generation of a new token representing the traffic (named *ONOFF-Cam2to5vm1_to_lpr511* in this example). The WorkloadControl token is passed further tho the next AbstractAction.

Once the sub-workload is executed, a single LoopControlToken is deposited in the LoopIterDone ordinary place. Once the LoopIterDone place collects X LoopControlTokens, the tokens are consumed, and a single WorkloadControl token is produced and passed to the output of the SubnetPlace. All tokens representing the traffic (i.e., with color other to the WorkloadControl and LoopControlToken) are passed directly to the output place of the Loop SubnetPlace (see Fig. 3.8).

3.4.3 Post-transformation Operations

In the *DNI-to-QPN* transformation, the post-block is used for renaming the resulting entities (to ensure uniqueness of the objects) and to connect the objects that couldn't be connected within the rules. The following actions are run in the post-block.

1. Every input and output place of a subnet is locked
2. Every Place becomes a SimqpnPlaceConfiguration object where the gathered statistics are specified

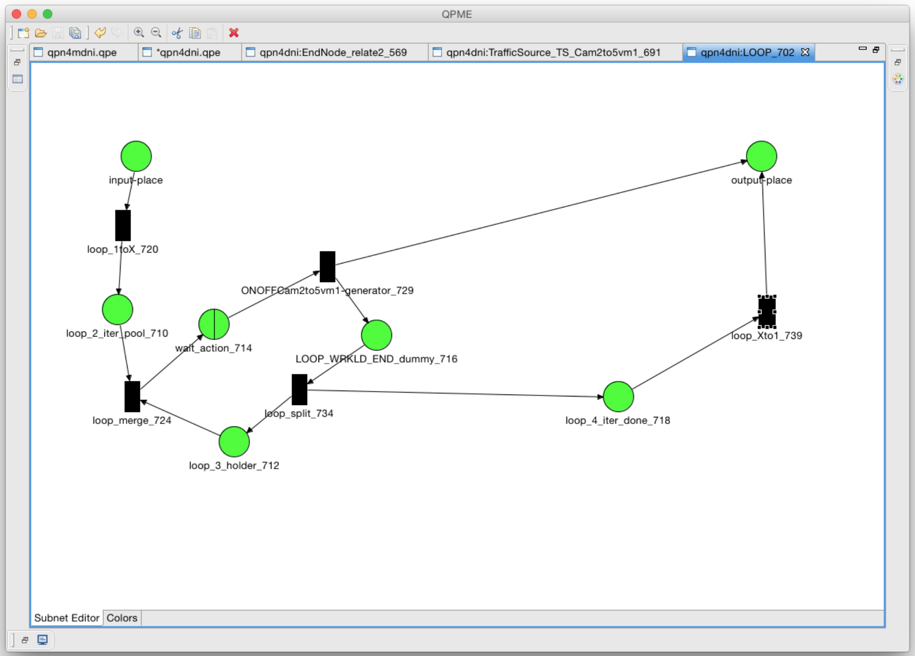

Figure 3.8: The QPN representation of a LoopSubnet. The sub-workload of the loop is included between the *WaitAction* and the *LoopWrkldEnd* places—in this example, the sub-workload of the loop consist of a single TransmitAction named *ONOFFCam2to5vm1*).

3. Every unused Queue is removed (e.g., for NetworkInterfaces that are down [isUp=false])
4. The output model is validated, especially:
    a) We check if every transition connects only with places, and every place connects only with transitions
    b) We remove places without color reference (they are simply unused)
    c) We check if the set of color references of a Subnet is equal to the sets of color references of the input and output places of that Subnet.

3.5 miniDNI-to-QPN

3.5.1 Pre-transformation Operations

In the *miniDNI-to-QPN* transformation, the pre-block is used for declaration of global variables that need to be accessible for the whole transformation, e.g., mapping of DNI-flows to QPN-colors, defining global QPN-colors (ether-color).

3.5.2 Transformation Rules

The rules execute in the order of appearing.

Rule Network2Document

This is a root-rule that creates an empty QpmeDocument and initializes containers for queues, places, colors, transitions. Additionally, it discovers all Flows in the miniDNI model and creates colors for them.

Rule Node2Subnet

In this rule, a SubnetPlace is generated for every instance of the Node. The actions are executed in the following order.

1. The constant Input and Output places and transitions are added to the Subnet. The ether color is assigned to the places.
2. If there exist traversing colors (colors for flows that do not start nor end in this Node), the *pass* transition is created, and for each traversing color a mode is added (see Fig. 3.9).
3. The *dummyTrafficSource* is created between the input and output transitions. The ether color reference is added to the place.
4. For each TrafficSource entity, the following procedure is executed
   a) A *loop_sw* queueing place is added to control the messages generation. The delay for the queueing place is calculated based on the number of messages per second (a parameter of the Workload entity in miniDNI).
   b) The initial marking for the *loop_sw* place is set to one token of Workload-Control color.
   c) A *sw* (software) transition is added. This transition consumes single WorkloadControl token from the *loop_sw* place, then immediately returns the same token to the *loop_sw* back and generates a single token that represents a single message of the Workload of a given Flow.
   d) The *sw* transition is connected to the output transition using a dummy ordinary place. Only the tokens that represent traffic are passed to the output. The WorkloadControl tokens are passed only between the *loop_sw* and the *sw* transition.

e) An example of the QPN representation of a Node with two traffic sources is presented in Figure 3.10.

5. The input transition destroys each token for which the given node is marked as destination. The input transition receives a mode that consumes any tokens on the input and produces zero tokens on the output. No tokens arrive to the traffic sources as those are only responsible for generating the traffic. Colors that should be forwarded (traversing) do not arrive to the input transition; they are passed from the input place directly to the *pass* transition (see Fig. 3.9).

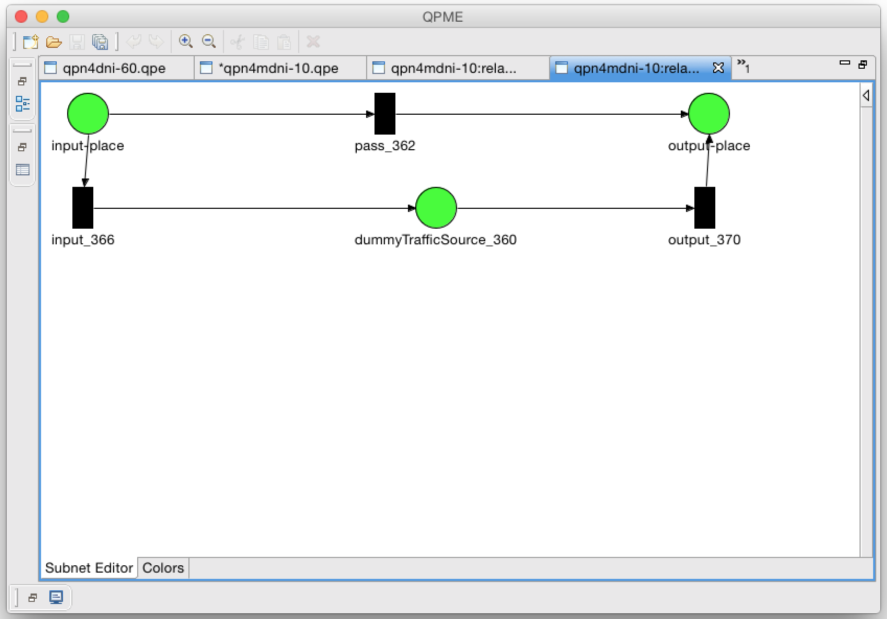

Figure 3.9: example of a QPN representation Node that does not generate any traffic, but only forwards (pass transition) or acts as a termination node .

Rule Link2PT

In this rule, a Link is transformed into a pair of „transition-queueing place-transition" objects. Each pair of such objects represents a single direction in which a flow can traverse the link. Conceptually, it can be presented as in Figure 3.11. In the QPME, it looks like presented in Figure 3.12.

Further in this rule, the colors are divided into those traversing the link in different directions. For each direction the proper modes are added to the transitions. For each

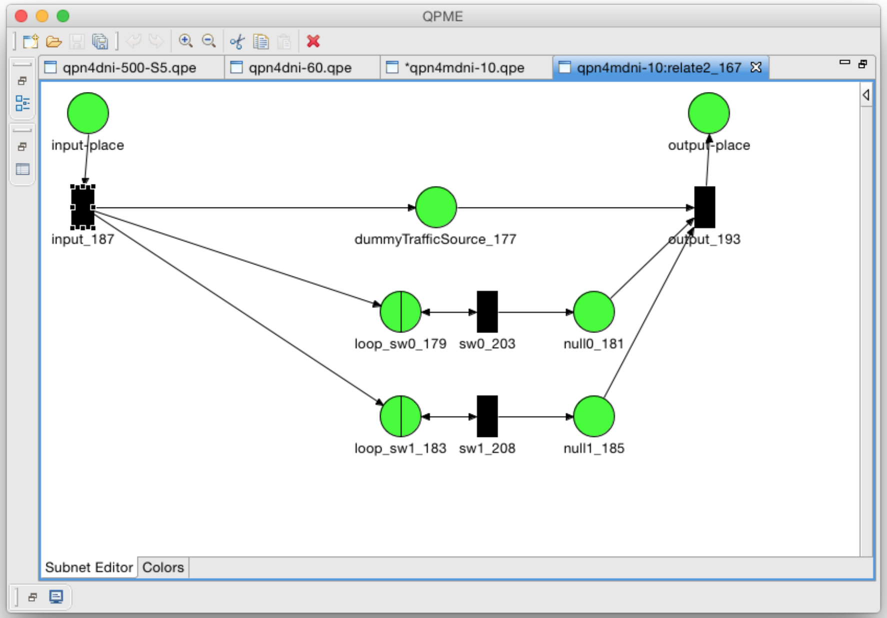

Figure 3.10: The example of a QPN representation Node that acts as a traffic source or terminating node. The missing *pass* transition means that this node is not traversal for any flow.

color traversing the link, an appropriate delay is calculated for the queueing place. The delay is stored in the LinkPerformance entity in the miniDNI model and it combines the delays expressed in the DNI as the sum of delays for the NetworkInterfaces and the Link itself.

Finally, the Links are connected to the Nodes by adding proper modes to the four transitions that are present in the QPN representation of a Link. A mode is created only if a given color is traversing the selected link in a selected direction (see Fig. 3.14). In case of an unused link, the link might be deleted (this is possible only in some special cases) or ether color is the only color that traverses the link (see Fig. 3.13).

3.5.3 Post-transformation Operations

In the *miniDNI-to-QPN* transformation, the post-block is used for renaming the resulting entities (to ensure uniqueness of the objects) and to configure the statistics of the objects.

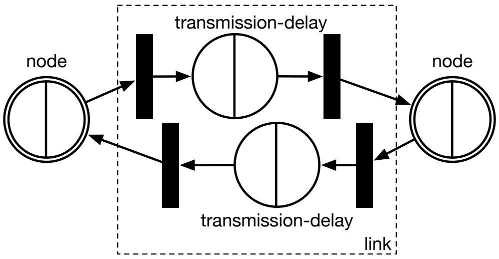

Figure 3.11: A pair of transition-queueing place-transition objects representing a Link in QPN. Conceptual presentation.

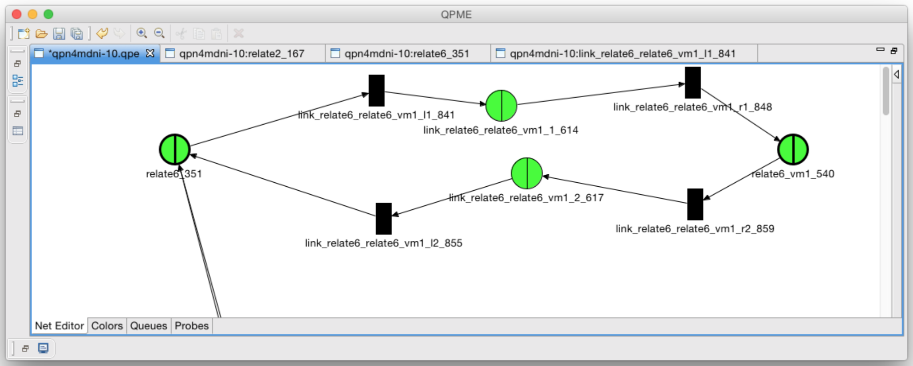
QPME
*qpn4mdni-10.qpe
qpn4mdni-10:relate2_167
qpn4mdni-10:relate6_351
qpn4mdni-10:link_relate6_relate6_vm1_l1_841
link_relate6_relate6_vm1_l1_841
link_relate6_relate6_vm1_1_614
link_relate6_relate6_vm1_r1_848
relate6_351
relate6_vm1_540
link_relate6_relate6_vm1_2_617
link_relate6_relate6_vm1_r2_859
link_relate6_relate6_vm1_l2_855
Net Editor
Colors
Queues
Probes

Figure 3.12: A pair of transition-queueing place-transition objects representing a Link in QPN. Representation in the QPME user interface.

3.6 Routing-format-conversion

The routing format conversion is done in three steps. For each step a separate part of the meta-model is designed. The three representations of routing in the DNI meta-model are redundant. The redundancy is caused by the evolution of the meta-model. The redundancy will be removed in the future version of the meta-model. The three steps of conversion can be schematically presented as in Figure 3.15 and described as follows.

For transformation of classical to flow-based routing:

1. Input is a classical routing scheme (see Fig. 3.16)
2. Duplicate each route that is used by more than a single flow.
3. Assign a single flow to each route that is used by that flow (see Fig. 3.17)

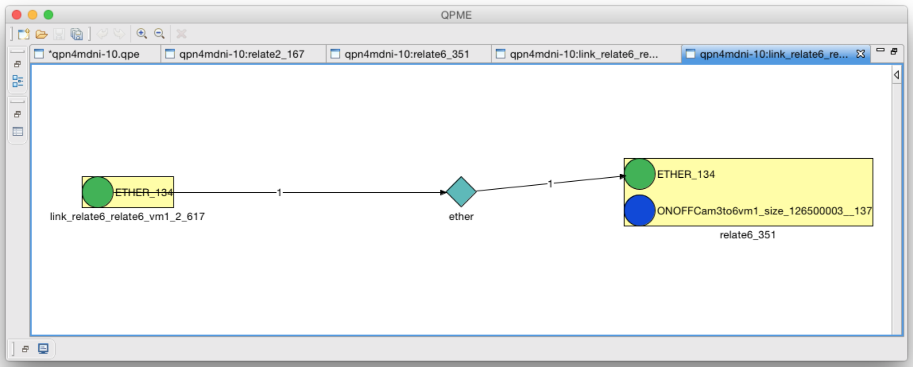
QPME
*qpn4mdni-10.qpe
qpn4mdni-10:relate2_167
qpn4mdni-10:relate6_351
qpn4mdni-10:link_relate6_re...
qpn4mdni-10:link_relate6_re...
ETHER_134
link_relate6_relate6_vm1_2_617
1
ether
1
ETHER_134
ONOFFCam3to6vm1_size_126500003__137
relate6_351

Figure 3.13: One of the four transitions included in the QPN representation of a Link. The link in the picture does not carry any traffic in the given direction. Only the mode for the ether color is added to guarantee the liveness of the QPN.

4. For every Node that the route connects, create a *Direction* entity, that stores the current node, next interface, flow, and the destination of the flow (see Fig. 3.18).
5. Remove duplicate direction entries. Remove routes without flow assigned.

For transformation of flow-based to classical routing (data is being lost in this transformation):

1. Input is a set of Direction entities.
2. For each flow, derive the source and destination node. Create an empty route.
3. Follow every empty route beginning from the starting node and processing the path that is derived by the Direction entities. Add respective hops to the route until the destination node is reached.
4. Intermediate result: set of FlowRoutes with flows assigned to each FlowRoute.
5. Group the FlowRoute entities by a tuple: source node, destination node. For each group that has more than one route (i.e., there are more routes between the given pair of nodes), select the route with the shortest number of hops and remove the rest (here, the information about redundant routes is lost).
6. Remove flows descriptions from the FlowRoutes. The result is a set of Route entities.
7. (optional) For every Route entity, create a reverse route if such route does not exist yet.

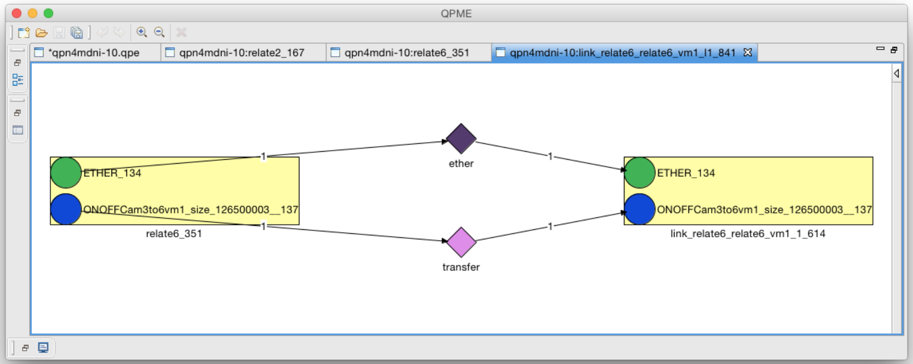
QPME
*qpn4mdni-10.qpe
qpn4mdni-10:relate2_167
qpn4mdni-10:relate6_351
qpn4mdni-10:link_relate6_relate6_vm1_l1_841
ETHER_134
ONOFFCam3to6vm1_size_126500003__137
relate6_351
ether
transfer
link_relate6_relate6_vm1_1_614
1

Figure 3.14: One of the four transitions included in the QPN representation of a Link. The link in the picture does carry traffic in the given direction.

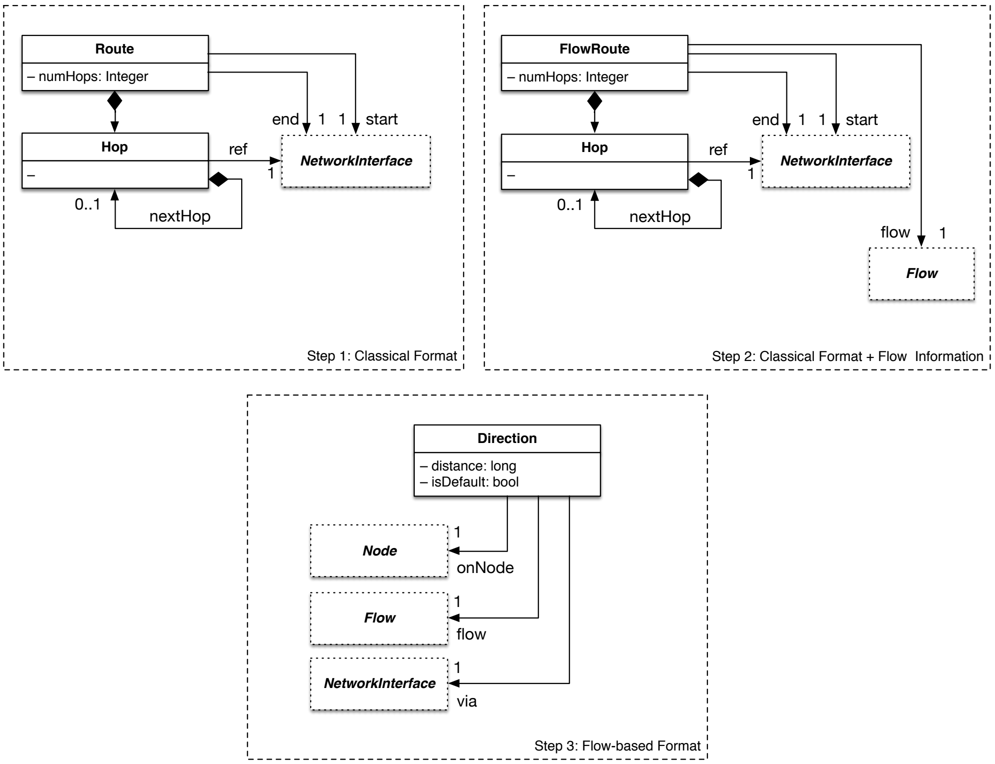

Figure 3.15: The meta-models of the initial, intermediate, and final step of the routing conversion procedure.

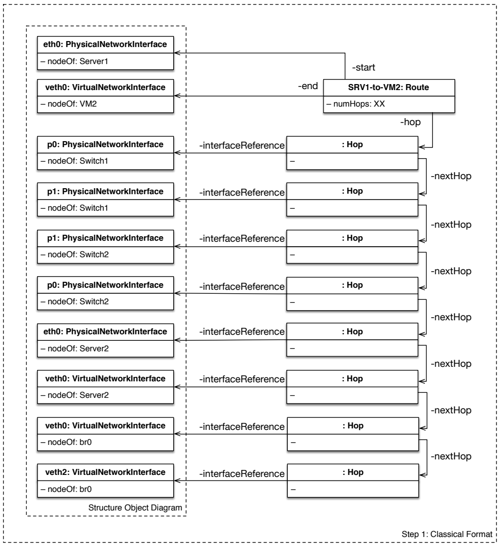
eth0: PhysicalNetworkInterface
– nodeOf: Server1
veth0: VirtualNetworkInterface
– nodeOf: VM2
p0: PhysicalNetworkInterface
– nodeOf: Switch1
p1: PhysicalNetworkInterface
– nodeOf: Switch1
p1: PhysicalNetworkInterface
– nodeOf: Switch2
p0: PhysicalNetworkInterface
– nodeOf: Switch2
eth0: PhysicalNetworkInterface
– nodeOf: Server2
veth0: VirtualNetworkInterface
– nodeOf: Server2
veth0: VirtualNetworkInterface
– nodeOf: br0
veth2: VirtualNetworkInterface
– nodeOf: br0
Structure Object Diagram
-start
-end
SRV1-to-VM2: Route
– numHops: XX
-hop
-interfaceReference
: Hop
-nextHop
Step 1: Classical Format

Figure 3.16: Step 1: Example of transformation *Route* via *FlowRoute* to a set of *Destination* entities.

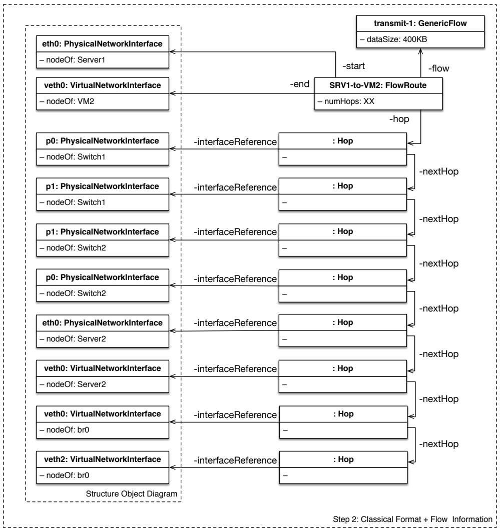
transmit-1: GenericFlow
– dataSize: 400KB
-flow
eth0: PhysicalNetworkInterface
– nodeOf: Server1
veth0: VirtualNetworkInterface
– nodeOf: VM2
p0: PhysicalNetworkInterface
– nodeOf: Switch1
p1: PhysicalNetworkInterface
– nodeOf: Switch1
p1: PhysicalNetworkInterface
– nodeOf: Switch2
p0: PhysicalNetworkInterface
– nodeOf: Switch2
eth0: PhysicalNetworkInterface
– nodeOf: Server2
veth0: VirtualNetworkInterface
– nodeOf: Server2
veth0: VirtualNetworkInterface
– nodeOf: br0
veth2: VirtualNetworkInterface
– nodeOf: br0
Structure Object Diagram
-start
-end
SRV1-to-VM2: FlowRoute
– numHops: XX
-hop
-interfaceReference
: Hop
-nextHop
Step 2: Classical Format + Flow Information

Figure 3.17: Step 2: Example of transformation *Route* via *FlowRoute* to a set of *Destination* entities.

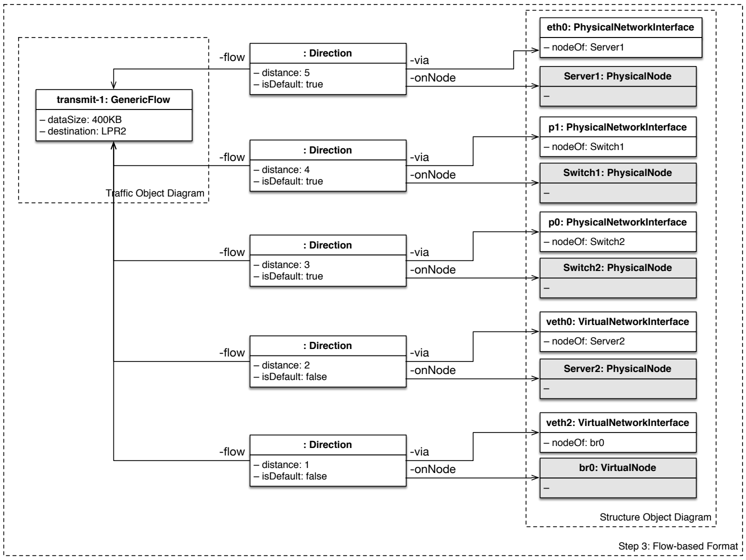

Figure 3.18: Step 3: Example of transformation *Route* via *FlowRoute* to a set of *Destination* entities.

Chapter 4

Conclusion

Bibliography

[BBE⁺08] Jean Bacon, Alastair Beresford, David Evans, David Ingram, Niki Trigoni, Alexandre Guitton, and Antonios Skordylis. TIME: An open platform for capturing, processing and delivering transport-related data. In *Proceedings of the 5th IEEE Consumer Communications and Networking Conference (CCNC), Las Vegas*, 2008. [see page 7]

[BHK13] Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo)*, 2013. [see page 1]

[DK14] Antonio Garcia-Dominguez Richard Paige Dimitris Kolovos, Louis Rose. *The epsilon Book*. 2014. Available online `http://www.eclipse.org/epsilon/doc/book/`. [see page 15]

[FHH02] A. J. Field, Uli Harder, and Peter G. Harrison. Network Traffic Behaviour in Switched Ethernet Systems. In *MASCOTS 2002, 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 32–42, October 2002. [see page 6]

[Ing09] David Ingram. PIRATES Data Representation. `http://www.cl.cam.ac.uk/research/time/pirates/docs/datarepr.pdf`, 2009. Accessed July 11, 2013. [see page 7]

[KMF04] Thomas Karagiannis, Mart Molle, and Michalis Faloutsos. Long-range dependence: Ten years of internet traffic modeling. *IEEE Internet Computing*, 8(5):57–64, 2004. [see page 6]

[KPP08] DimitriosS. Kolovos, RichardF. Paige, and Fiona A.C. Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations, vol. 5063 of LNCS*, pages 46–60. Springer, 2008. [see page 15]

[RS10] Piotr Rygielski and Pawel Świątek. Graph-fold: an Efficient Method for Complex Service Execution Plan Optimization. *Systems Science*, 36(3):25–32, 2010. [see pages 22 and 24]

[RZK13] Piotr Rygielski, Steffen Zschaler, and Samuel Kounev. A metamodel for Performance Modeling of Dynamic Virtualized Network Infrastructures (Work-in-progess paper). In *Proc. of the 4th ACM/SPEC Int. Conf. on Performance Engineering*, pages 327–330. ACM, 2013. [see page 3]