Document Properties

Document Number:	D 3.3
Document Title:	**NetIDE Methodology v1**
Document Responsible:	Christian Stritzke (IPT)
Document Editor:	Christian Stritzke (IPT), Claudia Priesterjahn (IPT)
Authors:	Christian Stritzke (IPT) Claudia Priesterjahn (IPT) Roberto Doriguzzi Corin (CN) Alexander Leckey (INTEL) Elisa Rojas (TELCA)
Target Dissemination Level:	PU
Status of the Document:	Final
Version:	1.0

Production Properties:

Reviewers:	Claudia Priesterjahn (IPT), Pedro Andres Aranda Gutierrez (TID), Roberto Doriguzzi Corin (CN)

Document:	CNET-ICT-619543-NetIDE/D 3.3		
Date:	April 15, 2015	Security:	Confidential
Status:	Final	Version:	1.0

Abstract:

NetIDE develops a toolkit to aid Network Application developers in the development of Applications. The toolkit comprises a collection of tools and artefacts to use during the development process. Deliverable 3.3 provides a methodology which describes the development process of a Network Application with NetIDE and links the aforementioned tools and artefacts. It also describes the user interfaces of tools and the concrete syntaxes of the languages to create the artefacts. The methodology and the tool and language descriptions form a basis for a roadmap of the tools and languages to be developed in Years 2 and 3.

Keywords:

software defined networks, networking, future internet, openflow, integrated development environment, methodology

Contents

List of Figures

Listings

List of Acronyms

IDE Integrated Development Environment

SDN Software-defined Networking

GUI Graphical User Interface

API Application Programming Interface

RAM Random-Access Memory

JSON JavaScript Object Notation

CPU Central Processing Unit

1 Introduction

NetIDE will provide a set of tools and languages, namely *NetIDE Developer Toolkit*, to support developers implementing and network administrators deploying and configuring Network Applications (i.e., applications which run on top of an Software-defined Networking (SDN) controller). Developers and administrators further create artefacts using the NetIDE Developer Toolkit. These artefacts are processed again by other tools. Both tools and artefacts have been briefly described in Deliverable D 2.2 [1]. This document provides additional details about them from a developer's or administrator's perspective.

In this deliverable, we document the *NetIDE Method*. The NetIDE Method describes the whole development life cycle of Network Applications using the NetIDE Developer Toolkit. The development life cycle includes implementing and annotating a Network Application, compiling and packing for delivery to network administrators who test, install and execute the Network Application on a productive system. The development life cycle links manual and automatic tasks and specifies the produced and processed artefacts. The NetIDE Toolkit focuses on enabling developers to conveniently implement reusable applications and administrators to test and deploy these applications with as little manual work as possible.

We start in Chapter 2 by giving a general overview of the method. The artefacts and tools introduced there are explained in Chapter 3. This chapter includes a description of user interfaces and a more detailed insight in the development processes with NetIDE unlike D2.3 in which have a stronger focus on the APIs and semantics of the artefacts used in the development process. Chapter 4 summarizes this document and provides a roadmap for the implementation of the described tools in the remainder of the project.

2 The NetIDE Method

NetIDE supports developers in five different phases of the development of Network Applications: Development, Compilation, Testing, Deployment, and Execution on productive systems. The NetIDE Method specifies the process and artefacts that are produced or edited in each phase. In this chapter, we describe the phases and artefacts in detail and list the tools which support developers in each phase in Sections 2.1 - 2.5.

2.1 Development Phase

Developers implement Network Applications during Development Phase producing Network Application code for a controller of their choice. Aside from implementing, developers create metadata in order to specify data that is relevant for the Deployment and Execution Phases later on. Figure 2.1 shows the activities and artefacts of the Development Phase. Activities are depicted by blue rounded rectangles, artefacts by white, cut-off rectangles, and tools by green rectangles. In the remainder of this section, we explain the Development Phase from the point of view of the artefacts because the activities of the Development Phase only consist of creating these artefacts.

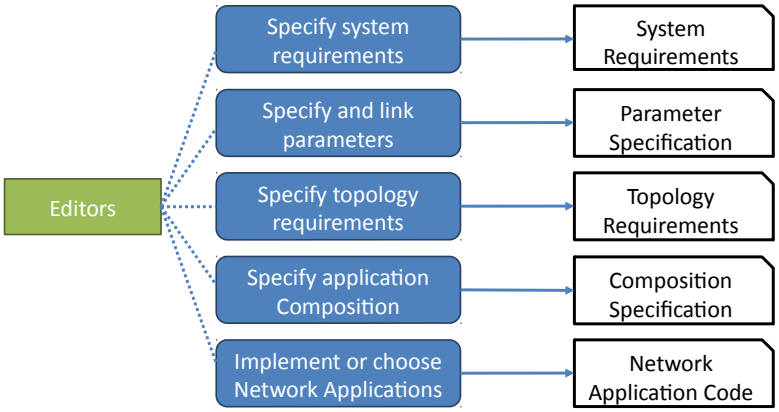


Figure 2.1: Activities and artefacts of the Development Phase

System requirements specify a set of systems on which a Network Application can be deployed. This includes hardware as well as software requirements. Application developers can, for example, state that their Network Application only runs on machines with more than 512 MB RAM, an installed Java runtime, and connected switches which support OpenFlow 1.0 or higher.

A **Parameter specification** enables developers to provide an interface with which network operators can configure a Network Application before deploying it. Parameters can be for example firewall rules, a preferred routing algorithm (for a routing application), or whether the Network Application provides a web interface.

Topology requirements specify a set of topologies on which a Network Application can operate. Not every Network Application performs well on every topology, thus, developers can make use of the possibility to narrow down the number of topologies for which the Network Application works. A simple way to express a topology requirement is a concrete topology. This option can be used

when a Network Application is tailored to one specific network. However, Network Applications can often run on multiple topology configurations. In order to specify a larger set of possible topologies, developers can specify abstract elements in the topology model. This allows, for example, Network Applications which only run on meshes of a certain dimension.

A **Composition specification** specifies the interconnection of Network Applications. Developers can compose Network Applications for different controller frameworks in order to form a single Network Application which combines the functionality of the Network Applications it is composed of. A Composition Specification links Network Applications stating how events from switches are propagated and how to handle flow mods coming back from the controllers in order to produce one single composed Network Application.

2.2 Compile Phase

Most of the work at the Compile Phase is done automatically in the NetIDE Method. At the end of this Phase, there is a **NetIDE Package** containing the requirement and parameter specifications as well as the Network Applications themselves. Figure 2.2 shows the activities and artefacts of the Compile Phase according to the syntax used in Figure 2.1. In the remainder of this section, we explain the Compile Phase from the point of view of the activities of this phase.

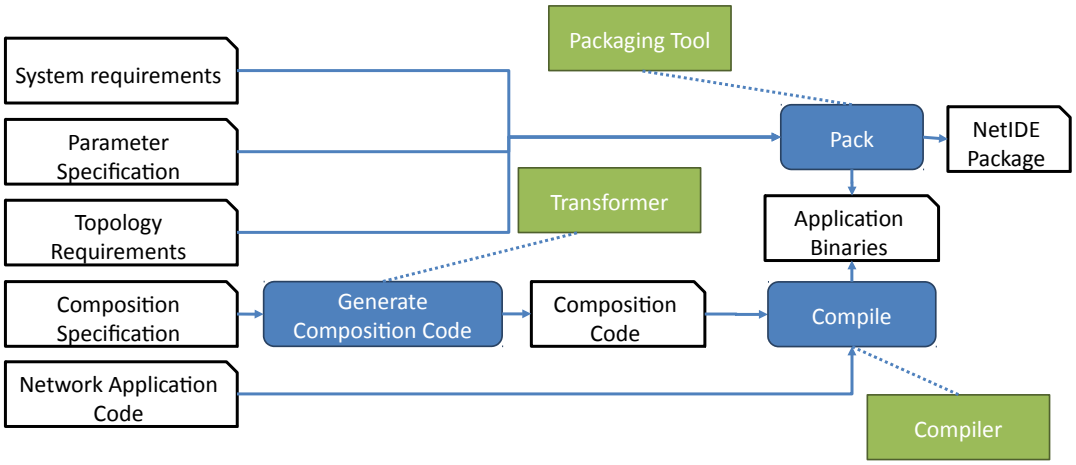


Figure 2.2: Activities and artefacts of the Compile Phase

The activity **Generate Composition Code** describes the transformation from a composition specification to runnable code. A transformer generates **Composition Code** which reflects the composition specification and can be executed. The composition code intercepts the messages from a server controller to a client controller, and vice versa, and propagates them according to the specification. A switch event message for an incoming packet can for example be propagated to a firewall application first and, if the packet is not dropped, passed on to a learning switch application.

The Network Application code and the composition code are compiled into binary and/or object code by the existing compilers for the respective languages. **Application Binaries** are, naturally, only produced for compiled languages such as Java or C.

2.3 Simulation / Test Phase

Before running a Network Application on a production system, it can be tested in a virtual simulation environment. Figure 2.3 outlines the activities, artefacts, and tools involved in this phase.

Testers have to **set up a simulation environment**, i.e., specify the hardware and software running on the target machine. They also have to **set up a virtual network**. The configuration for the **simulation system** is completed afterwards. The NetIDE Package is deployed on a virtual machine with a virtual network. Testers can now run the machine together with the Network Applications in the package and a network simulator on which the Network Applications are tested.

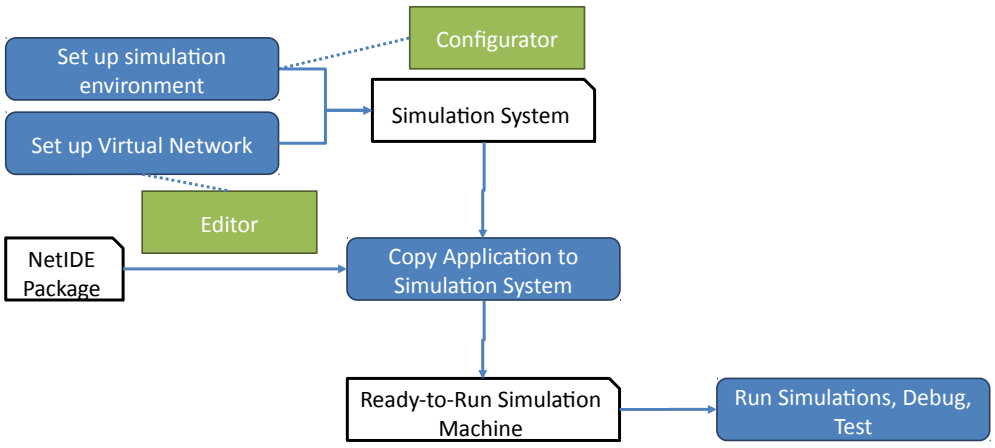


Figure 2.3: Simulation / Test

Testing involves the execution of the Network Applications as well as debugging, profiling, etc. This part of the NetIDE Method is explained in Section 2.5.

2.4 Deployment Phase

It is the task of a network operator to perform the steps for deployment on a production system in this phase which is outlined in Figure 2.4. Network operators have a **target system** at hand on which they deploy a (composed) Network Application. The target system comprises a controller with a given hardware and software configuration as well as a physical network topology. Being provided with a NetIDE package containing all artefacts described in Section 2.1, network operators go through the following steps.

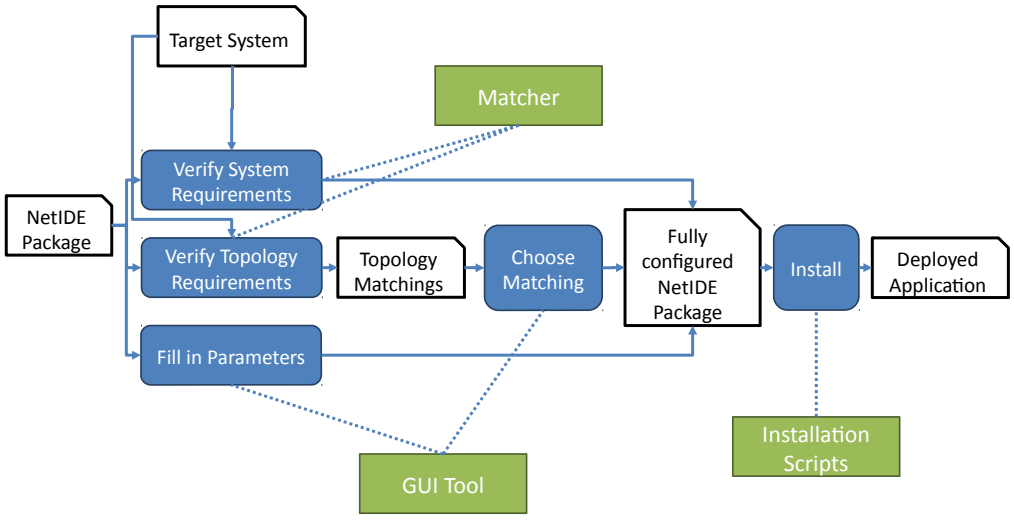


Figure 2.4: Deployment Phase

In the step "**Verify System Requirements**", NetIDE checks if the target system fits the requirements specified at the Development Phase and stored in the **NetIDE Package**. It obtains information about hardware and software configurations of the target system and verifies them against the system requirements. The package can be deployed on the target system if the requirements are met. Network operators are asked to install or reconfigure software on the target system if an unmet software requirement can be met that way. The package is not deployable if the requirements cannot be met in any way.

NetIDE also **verifies topology requirements**. The network topology of the target system is checked against the topology requirements in the NetIDE package. There can be three outcomes of that matching: a single matching, multiple matchings or no matchings at all. The package can be directly deployed in the first case. Network operators are prompted to choose a matching out of many matchings in the second case. This step is called "**Choose Matching**". Deployment is not possible in the third case.

Network operators have to **fill in parameters** specified in the parameter specification before deployment. After these steps, if the target system has passed all the verification steps, the package can be **installed**. This step includes copying and extracting the NetIDE package on the target system and installing or reconfiguring software if required.

2.5 Runtime

NetIDE **executes** the (composed) Network Application using the Network Engine on a simulation environment or production system in the last phase (cf. Figure 2.5). Controllers are launched automatically with the according parameters via scripts executed on the target system which start controllers and their Network Applications as well as the Network Engine.

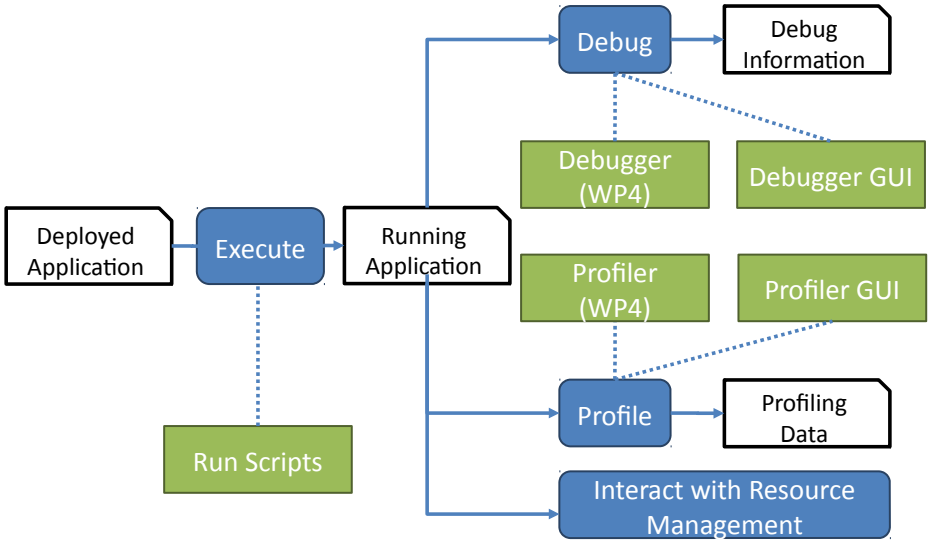


Figure 2.5: Execution / Runtime

The Network Engine from WP4 monitors running Network Applications and produces **Debug and Profiling Information** if chosen by operators or testers. This information is visualized in the IDE in order to make it easily accessible. Testers also have the possibility to manipulate flow tables or, in virtual networks, change the topology during runtime for debugging purposes.

3 Languages and Tools

This chapter describes the languages and tools that are used while developing Network Applications with the NetIDE Method. For each tool and language, we present a Graphical User Interface (GUI) mock-up or a language snippet to illustrate our concepts. We start by describing the specification languages and their processing throughout the NetIDE Method at the Deployment phase: the Parameter Specification Language in Section 3.1, the System Requirements Specification Language in Section 3.2, the Topology Requirements Language in Section 3.3, and the Composition Specification in Section 3.4. Afterwards, we give an overview of additional tools for the NetIDE Method in Section 3.5.

3.1 Parameter specification

This section describes the "**Specify and Link Parameters**" activity at the Development Phase as well as the "**Fill in Parameters**" activity at the Deployment Phase. Network Application developers using NetIDE may define an interface between Network Applications and network administrators. Therefore, we provide a Parameter Specification Language and a template engine which maps parameters to parts of the code or a controller command line. In the following sections, we describe how the Parameter Specification Language and the template engine work together for the definition of parameters. Parameter specification comprises two steps: specifying parameters by names, types, and semantic restrictions and mapping the parameters to the Network Application.

3.1.1 Specifying Parameters Using the Parameter Specification Language

In this section, we explain the specification and assignment of parameters for Network Applications using the Parameter Specification Language. The Parameter Specification Language provides the means to specify names and types of parameters. The specification can be expressed in a well-established format such as JavaScript Object Notation (JSON) [2]. A specification comprises a parameter description represented by the JSON object *parameters* and a type description indicated by the JSON object *types*. Parameters are specified by a name and a type. These types can be basic types such as integer, string, or boolean. They can also be types specified in the type description. The type description specifies types similar to *structs* in the C programming language. These types are a combination of basic types and optional constraints which restrict their possible values. The complete Application Programming Interface (API) for parameter specification can be found in D2.3.

The parameter description in Listing 3.1 shows a specification of parameters for a firewall application which can be controlled and monitored via a web interface. The first parameter is named *WebInterface* (expressed as a JSON object of the same name in line 22) with which the web interface can be switched on and off. The type *boolean* of that parameter is set with the attribute *type* inside of the object. Network Administrators can configure the rules of the firewall with the second parameter *FirewallRules* (line 25). This parameter is typed as a list of *firewall rules* together with an option for a *default policy*.

There is a type definition (lines $1 - 19$) above the parameter specification which specifies how firewall rules are structured (lines $7 - 18$). Each rule defines a *policy* after which packages with

certain properties are handled. The type "IP" (lines $3-6$) is defined as a string which matches the regular expression for IP addresses.

Listing 3.1: An example of a parameter specification

```
1  {
2    types: {
3      IP: {
4        type: String,
5        constraint: "match('[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}')"
6      }
7      FirewallRule: {
8        policy: {
9          type: ["allow", "drop"]
10       },
11       dstip: {
12         type: "IP"
13       }
14       srcip: {
15             type: "IP"
16       }
17       ...
18     }
19   }
20
21   parameters: {
22     WebInterface: {
23       type: "boolean",
24     },
25     FirewallRules: {
26       type: {defaultPolicy: ["allow", "drop"],
27               rules:"list(FirewallRules)"}
28     }
29   }
30 }
```

3.1.2 Passing parameters to network applications

Developers can specify how parameters are represented in the Network Application in the second step of Parameter Specification using a template language. Template languages are used to write large strings with placeholders which are replaced during the template rendering process. Most template languages also support statements such as loops and conditional expressions. In our example, we use the Jinja2 [3] template language in which placeholders are enclosed by double curly braces and ("{{...}}") and statements are enclosed by "{%...%}".

We assume two ways of passing parameters to a Network Application: configuring the command line to start the application and editing source code and configuration files.

We take the example from above and assume to have a POX [4] application to which we pass our parameters. A command line template passing the parameters above to the POX command line would look as shown in Listing 3.2:

Listing 3.2: Command line for POX with placeholders

```
./pox app.py --web_interface {{WebInterface}} \
             --default_policy {{FirewallRules.defaultPolicy}} \
             --fwrules {%for r in FirewallRules.rules%} {{r}}, {%endfor%}
```

The command line starts the POX controller with the command `./pox` and executes our example Network Application `app.py`. We assume that this Network Application provides the three parameters `web_interface`, `default_policy`, and `fwrules` as command line parameters. The parameter values assigned by the network administrator are now represented by Jinja2 placeholders for each parameter specified in the parameter specification.

A template rendering engine would now generate a command line replacing the parameter names in double curly braces with the values assigned at deployment time.

Templates to manipulate the source code of a Network Application work similarly. They are, however, not written directly into the code but stored in a folder structure outside of the Network Application's root folder. This enables developers to apply templates without having to edit the Network Application code itself. This is especially useful when they need to provide parameter specifications for Network Applications from foreign sources.

The Network Application from the example includes a python module "globals.parameters" in which flags and firewall rules are assigned to variables. The template file for this module is shown in Listing 3.3:

Listing 3.3: A python code snippet with placeholders

```
# globals/parameters.py.j2

WEBINTERFACE = {{WebInterface}}

DEFAULTPOLICY = {{FirewallRules.defaultPolicy}}

RULES = [{%for r in Firewall.rules%}
          {pol: "{{r.policy}}",
           srcip: "{{r.srcip}}",
           dstip: "{{r.dstip}}",
           [...] # More Attributes...
          }
        {%endfor%}]
```

This python module assigns parameters as the global configuration variables `WEBINTERFACE`, `DEFAULTPOLICY`, and `RULES`. The Jinja2 placeholders are replaced with the assigned parameters during rendering. The placeholder `WEBINTERFACE` can be assigned with *true* or *false*, `DEFAULTPOLICY` can be *allow* or *drop*. The firewall rules are assigned with a list of python dictionary object with a key:value pair for each attribute of a rule.

This is a very simple example for templates generating python code. The template file structure and the templates themselves can become much more complex if the parameters in the code are hard-coded in arbitrary parts of the code.

Network administrators have to assign values to the specified parameters before deployment. We generate a form in the Integrated Development Environment (IDE) to be filled out by network administrators using check boxes for boolean values, radio boxes for enum values, etc. Constraints are validated as soon as values are entered to ensure the validity of the parameters.

Figure 3.1 shows a form for the example. Aside from check boxes and radio buttons, the firewall rules are visualized by a table with a row for each rule and a column for each attribute of a rule.

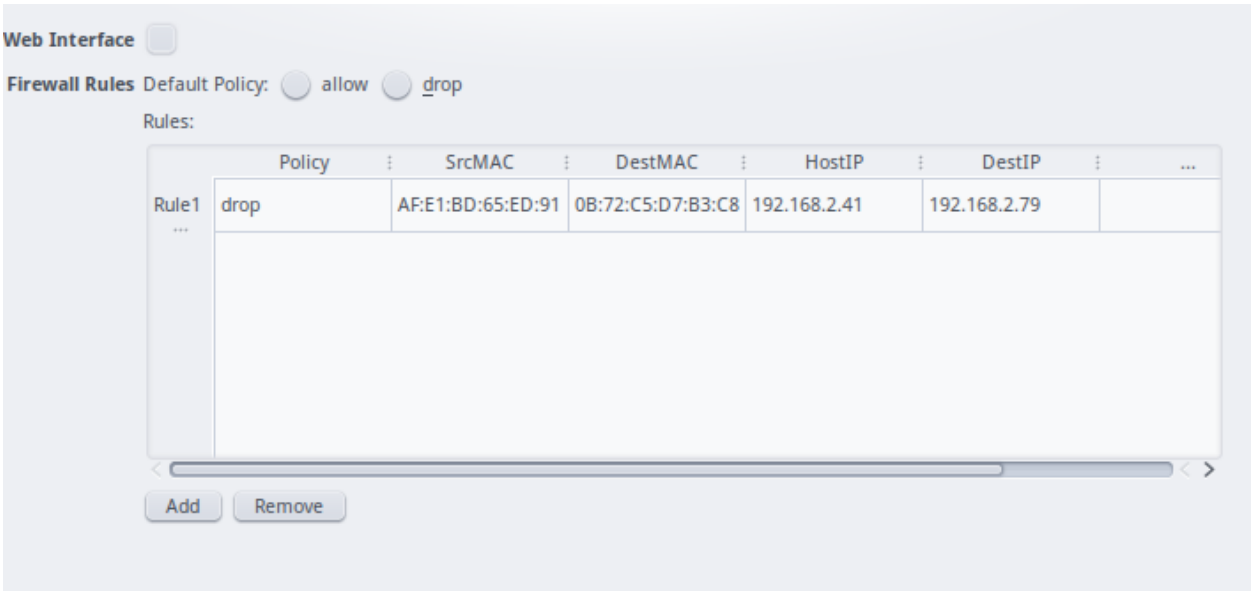


Figure 3.1: Parameter form at deployment time

3.2 System Requirements

This section describes the activities "**Specify System Requirements**" and "**Verify System Requirements**" of the NetIDE Method.

A system has to fulfill a certain set of requirements if network administrators intend to deploy a controller on it. Developers can specify hardware and software requirements for target systems in order to enable network administrators to automatically validate their target system and determine whether it fits the requirements. The system requirements specification are represented as a single file which contains the following.

Hardware and software requirements are expressed in the JSON format. Developers can specify hardware, software, and network requirements using JSON objects with the respective name. CPU specifications and required main memory can be specified using the inner objects of the *Hardware* object. Required controllers, libraries and programming languages can be specified in the *Software* object. The *Network* object can be used for required protocol versions and switch vendors. An example is shown in Listing 3.4:

The complete API of the system requirement specification can be found in Deliverable D 2.3 [9].

Listing 3.4: Example for a system requirments file

```
1  {
2    Hardware: {
3      cpuarch: "x86",
4      cpufreq: "800",
5      cpucores: "1",
6      memory: "512"
7    },
8    Software: {
9      Controllers: [
10       {name: "ryu", version: "3.15"},
11       {name: "pox", version: "carp"}
12     ],
```

```
13      Libraries: [
14            {name: "python-jinja2", version: "2.7.3"}
15      ],
16      Languages: [
17            {name: "java", version: "1.7"}
18      ]
19    },
20    Network: {
21      ofversion: "1.3"
22    }
23  }
```

This example specifies a required system with an x86 CPU with a minimum frequency of 800 MHz and at least one core. There also has to be at least 512 GB of RAM. The system needs the controllers "Ryu" and "POX" installed at specific versions as well as the jinja2 library for python and a Java 7 development kit. The underlying network needs switches with an OpenFlow version of at least 1.3.

Network administrators can validate the target system against the requirements at deployment time. The validator will issue warnings if the hardware requirements are not met and suggest the installation of missing software.

3.3 Topology requirements

In this section, we describe the activities "**Specify Topology Requirements**", "**Verify Topology Requirements**", and "**Choose Matching**" from the NetIDE Method.

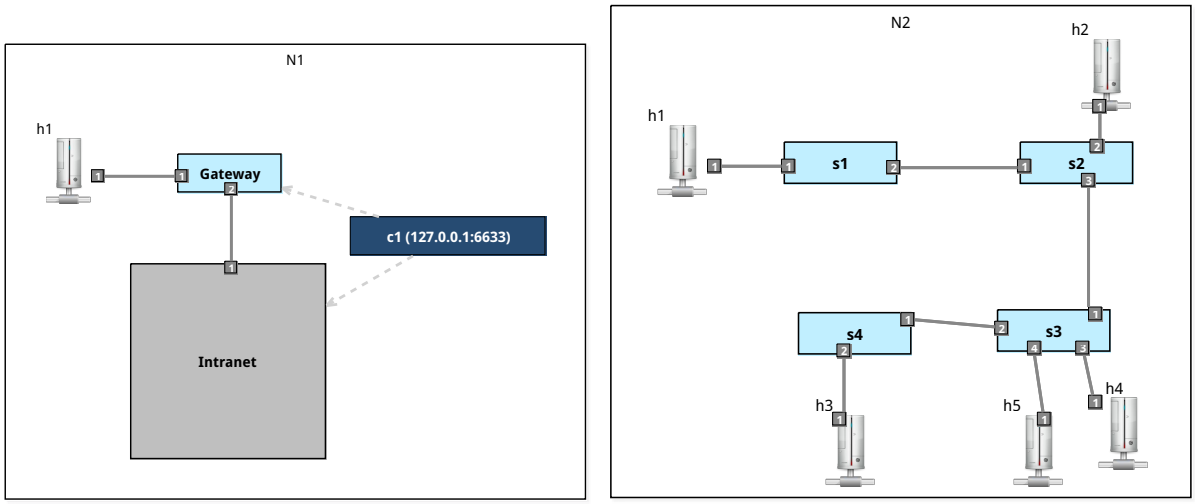
Developers can specify topology patterns on which network applications can run properly and check if the requirements can be met. NetIDE is going to provide an editor and a matching tool for that. These tools are described in this section. The editor for topology patterns is an extension of the concrete topology editor which is a part of the current NetIDE implementation and described in Deliverable D 3.2 [5].

First, we explain a simple running example to visualize the concept of topology requirements. The example features a Network Application for an internal company network (Intranet) which has to handle requests from the internet. It establishes a gateway switch which sends incoming packets from the internet to a gateway server. The server should then decide how to handle the packets on the application level (i.e. by checking credentials) and forward them to a host in the Intranet. The gateway switch is supposed to redirect all incoming traffic to the gateway server, allow the gateway server to access hosts in the Intranet and allow outgoing traffic from the Intranet. Switches inside the Intranet have standard MAC learning functionality.

The Network Application for the described scenario cannot work on every possible topology, since a gateway switch is required. The topologies needed for such a Network Application are expressed by the topology pattern in Figure 3.2(a). A light blue box in the upper half of the figure symbolizes a switch called *Gateway*. It is connected to one host and has one connection to the *Intranet*. This is represented by the connectors between the ports on the *Gateway* switch and the host *h1*. The *Intranet* itself is a black box which has no further restrictions other than the connection to the *Gateway* switch via a single port. Both the *Gateway* switch and the Intranet are controlled by the controller *c1*.

Network administrators have to verify the concrete topology of their target system against the requirements. Concrete topologies are obtained by a topology discovery on the target controller and translated into a NetIDE topology model. The model in Figure 3.2(b) shows the physical network topology of a target system obtained at deployment time. NetIDE now computes a matching

between the physical topology and the requirements. Two matchings can be found in the example. The first one takes the switch *s1* as a *Gateway* switch and declares the rest of the network as the Intranet. The second one matches *s4* to the *Gateway* switch. Both switches match the pattern of the *Gateway* switch with one port to a host and another port to the rest of the network.



(a) Topology pattern

(b) Concrete physical topology

Figure 3.2: Matching concrete topologies to a topology pattern

Network administrators have to choose between the proposed matchings before deployment. The switch IDs of the concrete topology will then be passed to the Network Application with a template engine, similar to passing parameters as described in Section 3.1. The following example shows a snippet of a Jinja2 template for a POX Network application checking for a gateway switch:

```
1 def _handle_PacketIn (self, event):
2   if event.dpid == {{Gateway}}:
3     ...
4   else:
5     ...
```

This piece of POX code shows a callback function which handles PacketIn events based on which switch it came from. If it is the gateway switch, the handler sends the packet to host *h1* first. The placeholder `{{Gateway}}` is replaced by the actual switch ID of the chosen topology matching during deployment time.

Topology requirements and concrete topologies are stored in an XML-file compliant to the Eclipse Modeling Framework. The complete API of the topology requirement specification can be found in Deliverable D 2.3 [5].

3.4 Composition Language

This section gives a brief overview of the plans for the activities "**Specify system requirements**" and "**Generate composition code**" of the NetIDE Method. The development of the concepts for a composition language is still ongoing work and we have not yet worked out a complete specification of the semantics of a composed network application.

The composition language will enable network application developers to reuse multiple Network Applications and combine them to a single composed Network Application. The composition

language provides a means to specify how events from switches are sent to specific Network Applications and how actions are propagated back from the Network Applications to the switches. Event processing includes sending an event to multiple applications in parallel or sequentially. Propagating actions mostly means handling multiple flow mods written at once. The language needs syntactical elements to enable the specification of the propagation of events and the definition and resolution of conflict situation with multiple writers.

We will explore existing approaches for Network Application composition and assess the applicability of their languages for our approach. These approaches will include Pyretic [6] which provides a language to specify composition of a Network Application out of multiple, independent python modules. Another approach, CoVisor [7], applies the concepts of that language to create a composition language for a hypervisor which manages Network Applications for multiple different controllers. Conflict resolution for multiple writers, however, is not supported by any of these approaches.

The NEMO Network Modeling Language [8] proposes a different way of Network Application composition than Pyretic and CoVisor. NEMO is an intent-based approach providing the means to model a network topology and assign its elements to zones which fulfil a certain function in the network. Hence, NEMO works on a higher level than our planned approach which works on lower-level artefacts such as events and flow mods. Modelling the functionality together with the network, however, creates a strong dependence on a specific topology. Network applications created with NEMO may not be as flexible as composed Network Applications in the NetIDE approach. NEMO is currently working on conflict resolution for multiple writers. We will consider whether their results are applicable in the NetIntegrated Development Environment (IDE) approach. More information on NEMO can be found in Deliverable D 2.3 [9], Chapter 3.3.

We are planning to implement a generator to translate composition specifications a into executable code reflecting the event and action propagation. The hereby created program will run as part of the Network Engine at runtime.

3.5 Miscellaneous tools

Minor tools and languages are listed here.

Source Code Editors Conveniently implementing a Network Application requires feature-rich source code editors with syntax highlighting, code completion, in-line syntax checking, etc. Most of the currently widely used controllers are programmed with standard programming languages like Java, C, and Python. The Eclipse plug-in framework already provides editors for many programming languages such that the development of new source code editors is not the focus of NetIDE.

Transformer for Composition Specification We will provide a transformer which generates runnable code out of a composition specification. This works straight forward and does not require any user interaction other than the provision of a valid composition specification.

Packaging Tool The packaging tools creates an archive (basically tar.gz or zip) out of the given artefacts produced at development and compile time. These artefacts are structured in a predefined file structure. Explanations are written as inline comments.

```
|
\_apps # The Network Apps are located here
   \_app1
```

```
   \_app2
   \_...
\_templates
   \_... # Template files for parameter and topology mappings
\_system_requirements.json # System Requirements specification
\_topology_requirements.treq # Topology Requirements specification
\_parameters.json # Parameter specification
```

Virtual Network Editor The virtual network editor is used to set up networks for simulation purposes. It is an editor which supports a subset of the topology requirements described in Section 3.3. There is a more detailed description in Deliverable 3.2, Chapter 3.

The virtual network editor will be enriched by additional features during the course of this project. We will for example assess the features of the network simulation testbed OmNeT and its .ned network description language [10]. OmNeT features hierarchical network specifications, components for reuse, inheritance, etc. These features enable a much more comfortable and structured way of designing networks and can be very valuable in the virtual network editor as well as the topology requirement editor.

Simulation Environment Configurator System requirement specifications can also be used to define a virtual machine on which tests can be performed. Instead of validating requirements, an installation script which provisions a bare virtual machine to fit the named requirements is derived from the specification.

Installation and run scripts NetIDE will deliver a collection of shell scripts for fast and easy deployment. The scripts copy the NetIDE package on a target machine, extract it, and install the Network Applications in the system, depending on the controller framework. They also install libraries, controllers, and programming languages on the target machine if intended by the network administrator.

Run scripts execute controllers. Depending on the framework this can be a java, an osgi, or a python command line with according parameters.

Debugger / Profiler GUI The data obtained by the tools in the Network Engine is visualized in the Eclipse GUI. Packets in flight and flow tables are shown in the concrete topology editor and profiling data is visualized with diagrams. The GUI also provides the means to remote-control the various tools running in the Network Engine.

4 Summary and Roadmap

4.1 Summary

This deliverable introduces the NetIDE Method – a method describing the development phases for a Network Application from development to compilation, testing, deployment and execution. The NetIDE Method targets Network Application developers who compose Network Applications that were implemented for different controllers possibly using different programming languages. The final Network Application can be annotated with metadata defining the controller hardware and network topologies it runs on. This enables network administrators to deploy a Network Application on their productive network and automatically verify if the Network Application is able to run on the productive network.

The deliverable also describes the tools needed to perform the tasks of the NetIDE Method in detail. This includes several specification languages, graphical user interfaces, and verification tools which come into play during the development process.

4.2 Roadmap

The roadmap for the further conceptualization and implementation of the tools and languages for the NetIDE Developer Toolkit is shown in the Gantt Chart in Figure 4.1.

There is a strong focus on the creation of the composition specification in Year 2 of the project. We plan to determine the semantics for event propagation and conflict resolution in composed network applications and have a prototype of a composition framework completed by Month 24 for the release of version 0.3 of the developer toolkit. We also plan to have a first language prototype ready at that time. Code generation from composition specification will be the focus in Year 3. We also determine how to handle conflicting requirements in composed network applications.

We plan to complete a first topology requirement language and editor ready by Month 22. This also includes a verification tool which verifies a concrete topology against the requirements. We elaborate on that in Year 3 adding advanced features such as the automatic creation of a concrete topology model from a target system's underlying network and live verification of the network against the requirements at runtime.

We will implement the system requirement language and verification tool in Year 3.

A first iteration of the integration of WP4 tools into the IDE will occur in Year 2 once, for example, the debugger yields first results which can be visualized in a graphical user interface. We will extend and polish the prototype from Year 2 in Year 3.

We will release a first version of the deployment tools for the automatic configuration of target systems and the execution of composed and basic network applications at the end of Year 2. The deployment tools will comprise support for all the controllers supported by the Network Engine so far. We will release a second version when NetIDE supports more controllers in the Year 3.

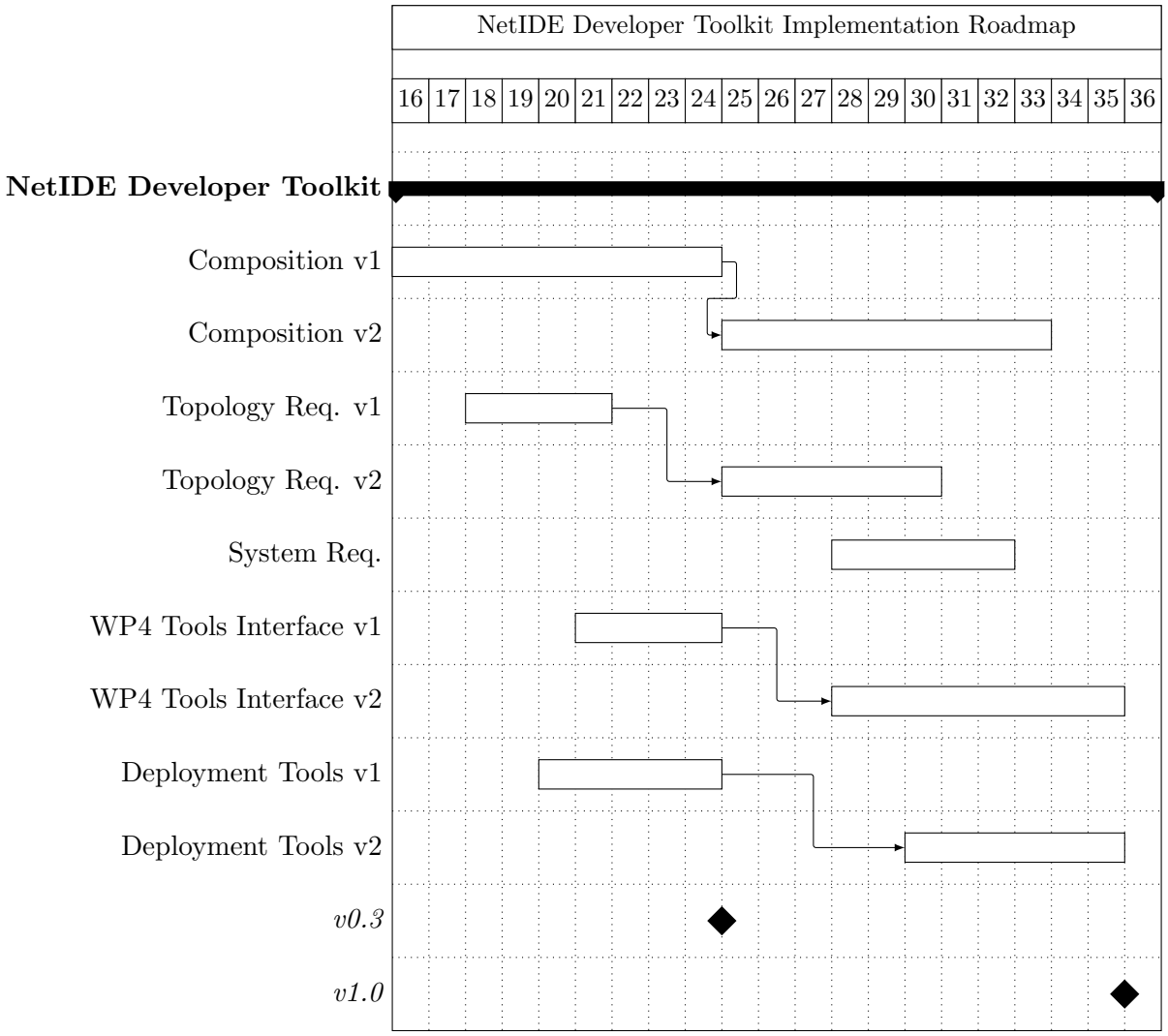


Figure 4.1: Gantt Chart for the NetIDE Toolkit Roadmap

A Bibliography

[1] The NetIDE consortium. D2.2 - NetIDE Intermediate Representation Format specification version 1.0. Technical report, The European Commission, 2014.

[2] Json: Java script object notation, April 2015.

[3] Jinja2 templating engine. http://jinja.pocoo.org/, 2014.

[4] Pox controller. `http://www.noxrepo.org/pox/about-pox/`, 2014.

[5] The NetIDE consortium. D3.2 - Developer Toolkit v1. Technical report, 2014.

[6] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Communications Magazine, IEEE*, 38(5):128–134, 2013.

[7] Xin Jin, Jennifer Gossels, Jennifer Rexford, and David Walker. Covisor: A compositional hypervisor for software-defined networks. *Networked Systems Design and Implementation*, 2015.

[8] Y. Xia, S. Jiang, T. Zhou, and S. Hares. Nemo (network modeling) language. Technical report, Huawei Technologies Co., Ltd, October 2014.

[9] The NetIDE consortium. D2.3 - NetIDE IRF APIs and Integrated Platform v1. Technical report, The European Commission, 2015.

[10] György Pongor. Omnet: Objective modular network testbed. In *Proceedings of the International Workshop on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems*, MASCOTS '93, pages 323–326, San Diego, CA, USA, 1993. Society for Computer Simulation International.