Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics

Automated Planning, Setup and Configuration for Scientific Testbed Environments

Christoph Rudolf

TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

BACHELOR'S THESIS IN INFORMATICS

Automated Planning, Setup and Configuration for Scientific Testbed Environments

Automatisierte Erstellung und Konfiguration von Testumgebungen

Author	Christoph Rudolf
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Nadine Herold, M.Sc.
	Dr. Matthias Wachs
	Stefan Liebald, M.Sc.
Date	December 15, 2016

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, December 15, 2016

Signature

Abstract

Experiments in the context of network infrastructure and software as well as the process of their evaluation, make use of testbed systems. The respective embodiments of testbeds vary greatly in their purpose and environment. As a result, the setup and configuration is a largely manual task.

This thesis offers an approach to structure and automate the setup and configuration process of network testing environments. With that, the thesis identifies key network components and steps necessary for their setup and configuration. These findings are used to propose an extensible, generic process that does not depend on a particular system or software. The design incorporates automated planning and scheduling as a central technique for determining setup and configuration plans. With the help of planning, the process is capable of taking the state of existing components into account when updating testbeds.

As a result of the thesis, an open-source application implementing the theoretical design of the process as a usable tool, is provided. Combined with a separate module for network topology information collection, the implementation forms the *IT NetworkS AnaLysis And deploymenT Application (INSALATA)*. The resulting implementation is evaluated in a case study using infrastructure present at the Chair of Network Architectures and Services.

Zusammenfassung

Testumgebungen stellen eine sichere und abgegrenzte Plattform für Netwerzwerkexperimente und die Evaluation von Infrastruktur und Software dar. Sie sind meist auf einen bestimmten Anwendungsfall ausgelegt, weshalb sie im Allgemeinen große Unterschiede in ihrem Aufbau aufzeigen. Aufgrund dieser Heterogenität erfordert ihre Konfiguration manuellen Aufwand.

Mit dieser Arbeit wird ein Ansatz entwickelt, um Testumgebungen automatisiert aufsetzen und konfigurieren zu können. Zu diesem Zweck werden zentrale Komponenten dieser Umgebungen ermittelt. Darüber hinaus werden für diese Netzkomponenten Aktionen definiert, mit deren Hilfe das Erzeugen und Konfigurieren dieser ermöglicht wird. Auf Basis der Komponenten und Aktionen wird ein allgemeingültiger und erweiterbarer Prozess entwickelt, der es ermöglicht neue Testumgebungen zu erzeugen sowie bestehende, unter Berücksichtigung existierender Infrastruktur, anzupassen. Um stets einen anwendbaren Ausführungsplan von Aktionen für Änderungen an einer Umgebung zu erhalten, wird *automated planning and scheduling* als Teilbereich der Künstlichen Intelligenz verwendet.

Als Ergebnis dieser Arbeit steht eine Implementierung des theoretischen Prozesses. Diese Implementierung bildet zusammen mit einem weiteren Modul zur Informationsgewinnung in Netzwerken die Anwendung *INSALATA (IT NetworkS AnaLysis And deploymenT Application).* Anhand einer Fallstudie auf Basis von Infrastruktur des Lehrstuhls für Netzarchitekturen und Netzdienste, werden Einsatzmöglichkeiten der Anwendung evaluiert.

Contents

Chapter 1

Introduction

Testbeds fulfill an important role in the evaluation of new network technology. They can consist of a mixture of physical and virtual infrastructure or be simulated completely on a single host. In many cases their topology is setup to either model a real-world network or revolve around a specific experiment.

For system administrators, they provide an environment for testing new infrastructure, topology and software before deployment into a production network. This way, new technology can be configured and familiarized within a safe environment. This mitigates the risk of unforeseen problems during deployment into production later on.

Depending on how accurate a testbed mirrors the behavior of real-world networks, the training of intrusion detection systems (IDS) can be an additional use case. Further security related topics, like training administrators in terms of possible cyber attacks, are also in need of a safe testing environment that ideally mirrors the real infrastructure.

Due to the different applications of testbeds, they are subject to frequent changes. Besides adding new hardware and software for testing purposes, being able to reset to a previous state is a necessary feature. The emphasis on reconfigurability is even stronger for scientific testbed environments. These testbeds are used for testing purposes by researchers and students and are often used by multiple users for an even wider variety of scenarios. In this case, switching fast between different experimental setups is a highly desired feature. If this is done manually, frequent changes to testbeds make them prone to misconfiguration. To resolve this problem, a structured and reliable setup and configuration process is required.

1.1 Goals of the thesis

The focus of this thesis is to create an automated setup and configuration process for resources inside a testbed. The goal for this process is to provide a means for an easier and structured testbed configuration. In addition to that, an implementation of the proposed setup process as a usable tool is created and used to conduct a case study for evaluation purposes. This implementation is integrated into the larger *IT NetworkS AnaLysis And deploymenT Application (INSALATA).*

1.2 Research questions

The overall research question this thesis answers is: **How testbeds can be setup and configured automatically?** This question is subsequently split into four smaller research questions.

Q1 Which information is necessary in order to setup and configure a testbed up to a reasonable level of complexity and how should this information be structured?
The level of detail up to which the testbed is automatically configured correlates with the information that has to be specified. Therefore, we have to answer what components can be covered by the automated setup process.

In order to enable users to describe a certain configuration, a data format is necessary. This allows to store testbed states which can be used to revert back to or share them with others. With this question, we also answer what type of data format is suitable for storing the information needed.

Q2 What are the individual steps during the configuration of the network components?
By breaking the process into pieces, the complexity of a testbed setup becomes manageable. It also allows to reconfigure existing testbed components by applying fine grained changes instead of deleting all components and setting them up from scratch. The answer to this question will provide individual steps that are part of the setup process and have to be implemented.

Q3 How to determine changes between testbed states?
To apply a certain configuration, the current components of the testbed have to be considered. Therefore, the thesis has to provide a solution for change detection at a level of detail that matches the granularity of the individual steps.

Q4 What is a generic approach to determine an execution plan consisting of the aforementioned steps?
We determine what constraints there are regarding the order in which setup steps are executed and how to find a generic and adaptable way to derive an executable step-by-step plan from a list of detected changes.

1.3 Outline

The thesis is structured as follows. In Chapter 2 we give background on testbeds in general. Chapter 3 starts off by defining the environment of the thesis. Afterwards, the given problem is analyzed in detail. Based on the problems determined, we describe our requirements for the automated setup process and introduce existing technology that is promising to be applicable in a solution. In the following Chapter 4 we assess related work using the previously defined requirements. Chapter 5 revolves around the design of the automated process. We describe its data model and elaborate a solution to the problem of change detection. Furthermore, we identify the necessary setup steps, their constraints and propose a solution to the planning problem. The chapter closes by describing design decisions regarding the setup steps used for building the testbed. Important details of the implementation are discussed in Chapter 6. Chapter 7 refers back to the initial requirements and evaluates how they are met. Furthermore, the chapter evaluates features of the implemented tool in a small practical example and by conducting a case study. Chapter 8 concludes with a summary of the results in regard to the research questions and discusses aspects that can be considered in further research.

Chapter 2

Background on testbed types

Testbeds provide a testing ground for network administrators and researchers to evaluate new network infrastructure or software. Testbed users aim to apply the insight gained during experiments conducted in the testbed to a real-world network. Whether or not these results are applicable to the real world, depends on the degree of realism provided by the testbed.

The realism a testbed environment offers correlates with the expense of setting it up. As one can easily imagine, an identical duplication of a network in terms of hardware, software and configuration, would provide an ideal testbed, while coming at a great expense. Owezarski et al. [1] distinguish between four different types of network abstractions that can be targeted in a testbed setup. These types are discussed subsequently.

2.1 Mathematical models

In this case the complete setup is entirely modeled using mathematical objects. There aren't any real components involved nor are there any network packages built. Conducting an experiment in this case revolves completely around mathematical evaluation of the modeled network. This provides the most lightweight but also the most abstract and therefore least realistic method of testing. [1, 2]

2.2 Simulation

Simulation describes the process of modeling networks including their components entirely by software which is run on a single machine. A widespread example would be *ns* (currently *ns-3*) which acts on a discrete sequence of events. Network components in *ns-3*, called *nodes*, provide implementations of the Internet protocol suite up to the transport layer and can be integrated into more sophisticated testbeds [3, 4]. As there are no real or full virtual hosts involved, simulation provides a cost-efficient way to conduct network experiments and measurements.

2.3 Emulation

Network emulation takes an additional step towards realistic testbed behavior by making sure the specific component being tested, hardware or software, is real [1]. This means that for software evaluation in a testbed, the underlying hosts and the network itself may still be virtual. They can be created as fully persistent virtual machines with independent filesystems or more lightweight by operating-system-level virtualization (containerization) in Linux [5]. If a hardware component is being experimented on, emulation has to make sure the specific component is the actual hardware. This mixture of real and virtual elements make emulation easier to achieve compared to a real system that mirrors a real network, as a small number of machines can provide all distinct network components (e.g. switches, hosts, routers). While they are still an abstraction of the real world, testbeds using network emulation can be reconfigured by software and allow for reproducible experiments [6].

2.4 Real system

In case of a real system, every component is implemented by real hardware. Ideally, all components of such a testbed would use the identical hardware and configuration as the production network for which experiments are conducted for. While this is the most accurate testing environment possible, it is also highly inflexible with almost no scalability [7].

2.5 Summary

As stated by Owezarski et al. [1], network emulation provides the best tradeoff between cost and realism. Figure 2.1 shows where each of the network abstraction levels reside in terms of cost and realism. The setup tool implemented in this thesis aims to configure testbeds making use of virtual machines while allowing to connect physical components that cannot be emulated easily. Therefore, testbeds considered in this thesis are located between pure network emulation and a real system. However, the proposed design is generic and can handle other testbed categories as well. Nevertheless, for less realistic testbeds, especially for purely mathematical models, our approach might be too complex.

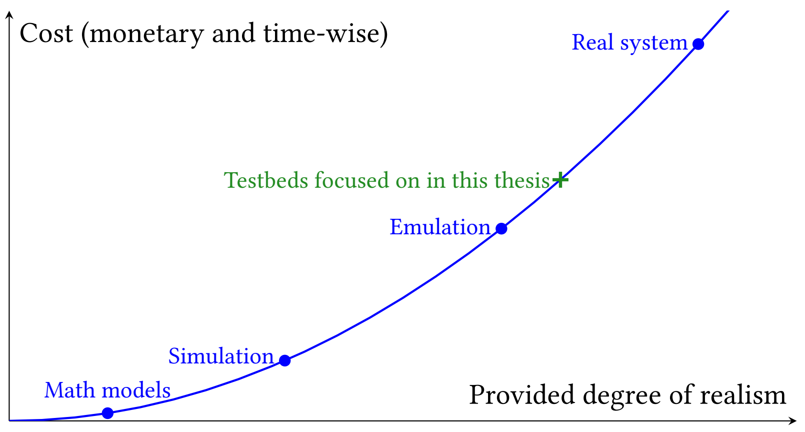


Figure 2.1: Network abstraction levels by relative cost and realism. Adapted from [1].

Chapter 3

Analysis

In this chapter, we analyze relevant problem- and solution-oriented aspects regarding the configuration and setup of testbeds. We start by describing the environment of this thesis, as well challenges of a testbed setup process. Afterwards, we utilize this information to introduce central requirements. The later part of this chapter will introduce existing technology available to solve problems like the ones arising from the research questions and requirements (e.g. host configuration and execution planning of setup steps).

3.1 Environment

The development of the *IT NetworkS AnaLysis And deploymenT Application (INSALATA)* was initiated as a reaction to issues encountered during the management of the testbed at the Chair of Network Architectures and Services. The project's main goal is to provide a solution for collecting network information and using this data for replaying it as a testbed infrastructure. This separates INSALATA into two parts, each covering one of the two aspects.

This thesis focuses on the latter aspect of the main goal by proposing an automated setup and configuration process as a generic approach for setting up and configuring network components in a testbed. With this, we want to solve issues of a manual testbed setup. Insight on these problems is given in Section 3.2. The topic of information collection inside networks is covered in [8] by a related thesis. The results of both theses are combined to form the complete INSALATA project.

3.2 Problem Analysis

Due to the variety of applications of testbeds, they differ greatly in terms of their respective form. This makes the setup and configuration of testbeds a largely manual and therefore time consuming task.

The environment for this thesis is given by a virtual testbed operated by the Chair of Network Architectures and Services. The challenges of testbed configuration we identified are drawn from this testbed. Without any kind of automation, the potential steps for configuring a testbed are:

- resource allocation and distribution of physical hardware and virtual machines
- installation of desired operating systems
- assignment of IP addresses (manually or by setting up a DHCP server)
- configuration of routing tables
- configuration of firewall rules
- setup of additional services (e.g. DNS, web server)

Besides being time consuming, the manual configuration of network setups in a testbed makes it prone to misconfiguration. Especially in multi-user scenarios, this is likely to lead to inconsistencies which can have undesired effects on the outcome of experiments. Without a structured description of the testbed state, it is hard to save or share an experiment's configuration. This makes it difficult to reconstruct experiments later on or sharing the configuration for an experiment with someone using a different testbed.

Drawing from the testbed infrastructure at the Chair of Network Architectures and Services, we can also analyze which network components are key elements in a testbed and have to be considered for an improved setup solution.

We identify experimentation hosts created by different testbed users. These virtual hosts are distributed over multiple networks where hosts of certain users are not limited to a single network. Rather, hosts of all users are mixed within the existing layer 2 networks. Two central routers connect networks and provide Internet access for other hosts. Here it is important to note that these routers are shared by many users and do also provide DHCP and DNS for their respective subnetworks. In addition to that, completely isolated networks without Internet access exist as well. While network traffic is not restricted via firewall rules, we also identify firewalls as a key component of testbeds. Routing is implemented by static routing rules which is reasonable as this gives testbed users the most freedom regarding settings for special experiments. We also observe that users use additional hard disks in order to store large experimentation results. For physical components that cannot be emulated easily, it is desirable to integrate them into the testbed.

If a new host or a new network is introduced, users are faced with a list of manual tasks. This includes the creation of the new elements itself, but also making minor changes to various existing network components. In the existing testbed for example, new hosts have to be considered for DHCP, making changes to the configuration of one of the servers necessary. The same goes for routing if an entirely new network is introduced. Changes to a single experimentation host, like adding a new network interfaces, requires the user to issue commands in the hypervisor's command line interface. For all changes, users need to acquire knowledge about the testbed and have to come up with their own list of changes to execute in a correct order. This is prone to being incomplete or faulty, due to the complexity of a growing testbed.

3.3 Analysis of requirements

The following section gives a definition of all requirements we strive to fulfill with the automated setup and configuration process. These requirements are drawn from the problems of the existing testbed we analyzed in Section 3.2. They are further used in Chapter 4 as criteria for evaluating related work. The evaluation in Section 7.1 refers back to the requirements in order to ensure that they are met.

R1 Coverage of basic components – The setup process covers a set of basic components that provide the basis for common testbeds. This requirement stems from the question of what components and which of their aspects are covered (see **Q1**). Based on the network components observed during the analysis of the existing testbed, our solution is required to support hosts with multiple interfaces and disks, networks, routers with routing tables, DNS and DHCP servers as well as firewalls. This set of components is reasonable as it includes the very basic functionality that is necessary in order to provide a usable testbed. For virtual hosts, the number of CPU cores and the amount of memory has to be considered.

R2 Description language – In order to formalize testbed configuration, a description language is necessary. As we want the configuration to be easily sharable with others, a single file is desired. The data format has to cover all aspects of the components defined in **R1**. In case new elements are added to the ones covered by the setup process, the configuration file format has to be extensible. It also has to allow some type of references in order to model *part in* relationships like interfaces being part of a host. Besides that, referencing helps to avoid redundancy.

Even though reading and writing the file manually is not necessarily required in the complete system, human readability is a beneficial property. Further usability in a programming context is increased, if parsing and validating the chosen format is covered by standard libraries in common programming languages.

R3 Virtual and physical infrastructure – The tool is able to create virtual network components with given hardware specifications. For physical components that cannot be emulated virtually, a way to integrate them into the testbed is needed.

R4 Software agnostic design (extensibility) – The automated setup process must not depend on specific software like a certain operating system, hypervisor, tool for configuring hosts or service. While a concrete implementation has to be reduced to support selected software and operating systems, the general design has to be open and generic to allow new modules to be introduced with as less code changes as possible. No specific software should be fixated in a way that replacing it becomes impossible without rewriting parts not related to the software itself.

R5 Open source and public availability – As we want to enable developers to implement modules for additional operating systems, services or virtualization software, the resulting tool is open source. This makes the system publicly available and helps users who want to deploy the system into their infrastructure as they benefit from further implementation by the community.

R6 Separated persistent components – Many testbed control frameworks, some of which are discussed in Chapter 4, provide components only for the duration of an experiment. They include the configuration of all components into the experiment's execution. This requires users to include the installation and configuration of complex software in scripts used to run experiments. In many practical cases this turns out to be hindering, as users tend to explore the capabilities of new software by hand, before running scripted experiments at a larger scale. One of the main goals of the testbeds resulting from the automated setup and configuration process is to provide persistent and separated components. Thus, allowing users to make further changes to components manually. This is important in case a special software or service that is not covered by the automated setup process, has to be configured. With *separated* components, we refer to the fact that all components can be changed freely, without affecting any other existing component. This has some implications regarding lightweight virtualization with shared filesystems which is discussed in connection with related work in Chapter 4. With persistent components, the experimentation phase is clearly separated from the testbed configuration. This takes into account that a testbed setup usually provides an experimentation ground for multiple related experiments on the same infrastructure.

R7 Multi-user support – The testbeds that are setup and configured using the automated setup process have to allow multiple users to use them. The system has to support this in a sense that each user can deploy multiple testbed setups at once. The system has to be able to distinguish between these configurations by being able to keep track of which configuration each component is part of.

R8 Sharing components – The analysis of the existing testbed and its usage showed that it is desirable for key infrastructure components to be shared among users. This is currently the case for central routers, DNS or DHCP servers that aren't immediate parts of an experiment and are only needed to provide their specific service.

This requirement is partially linked to the availability of references in the data format stated in requirement **R2**. Referencing is beneficial in this case, as we don't want users to have to specify all properties of a referenced component again.

R9 Continuous updates – A major requirement for the new automated setup and configuration process is to allow incremental deployment of new configurations by taking existing components into account. This is necessary in order to resolve the problem of testbed users having to make minor changes to various existing elements, like for example adjusting routing tables and DHCP settings. Therefore, the system has to recognize existing components and update changed properties according to the information given in a new configuration file. This is also needed as the components of a testbed are subject to further manually performed changes (see **R6**) that might not be covered by the automated setup. An alternative seen in many testbed frameworks, is the recreation of the all components from scratch. While this is straight forward in terms of the complexity of the setup, all data and software previously installed on the hosts are removed. In order to change only the necessary parts of a component without deleting and recreating it, the system has to implement a change detection mechanism which determines changes between the current state and a new goal state. From there on, a mechanism is needed to derive a working and deadlock-free execution plan.

3.4 Analysis of applicable technology

In this section, we want to introduce existing technology that is either prominent for the configuration of network infrastructure or offers promising features for our solution.

To configure both virtual and physical hosts with all aspects specified by **R1**, the automated setup process can make use of existing configuration management technology. Those are either complete configuration management tools or protocols designed for configuration purposes.

As requirement **R9** makes it necessary to create sound setup plans which are free from deadlocks, automated planning and scheduling does provide interesting options and is described in this section as well.

As far as the setup of hosts is concerned, the process of creating virtual network components is largely dependent on the hypervisor in use. Each hypervisor provides its own toolstack which can be used to deploy virtual components via scripting. For physical components, the creation itself is excluded from the automated setup. The installation of operating systems can be achieved by techniques like *PXE (Preboot eXecution Environment)* which only requires a network interface capable of using the technology.

3.4.1 Configuration management tools

Software systems for management of network infrastructure have developed majorly over recent years. In [9], Morris identifies virtualization, cloud, containers and software defined networking as major factors for the increasing use of automation for administrative tasks. With these new technologies, the creation of server platforms becomes scriptable as they are now decoupled from the concrete physical systems they run on. To avoid repetitive and time consuming manual reconfiguration in changing environments, automation is the desired means of configuration. By depicting infrastructure components in a data model, their properties can be specified once in a definition file which can then be deployed by a software system rather than by using a manual process. This approach of managing IT infrastructure is called *Infrastructure as Code (IaC).* [9]

In the testbed scenario, the user's input specifies a testbed configuration in a certain data model. This makes the IaC approach directly applicable for testbed configuration.

There are currently a variety of tools available which can be used for configuration based on definitions. Examples are *Puppet* [10] and *Ansible* [11]. Each of these tools represents one of two ways to approach configuration. A configuration can either be applied by a central unit that connects to all hosts *(push model).* As an alternative, each of the hosts can actively fetch configurations from a central server *(pull model).* [9]

Ansible uses the push model in combination with an SSH connection which allows it to operate without requiring any client software on the hosts. Using Ansible, the user creates so called *playbooks* which specify a collection of tasks to run on the remote host. These tasks can be the installation of software, the execution of arbitrary commands or copying template files for configuration. The latter one can use the *Jinja* templating engine which allows creating template files that are filled with variable values specified when running the playbook.

Puppet relies on a pull model including an agent on the hosts themselves. The agent pulls configuration files from a central server and determines changes between the host's state and the configuration specified. As the automated testbed setups can rely on prepared images for setting up hosts, the requirement of a user agent in Puppet can be fulfilled easily, thus, giving no distinct advantage to Ansible.

3.4.2 Protocols for management and configuration

In addition to the configuration management tools, standardized network protocols for management purposes exist as well. Widely known examples are the *Simple Network Management Protocol (SNMP)* and the *Network Configuration Protocol (NETCONF).* These protocols provide access to the configuration of hardware components like routers and hardware firewalls.

SNMP is an older protocol used mainly for reading configuration of hardware components. By using an information model named *Management Information Base (MIB)*, the variables which vendors want to provide via SNMP are structured in an extensible hierarchy [12]. There are widely supported standard MIB elements in SNMP. However, each vendor defines proprietary MIB modules, making a generic usage very difficult. Besides reading device settings, MIB elements can be written for configuration. Unfortunately, the number of writable MIB modules is very limited, especially for routers, which prevents the complete configuration of them via SNMP. Instead each vendor defines proprietary command line interface tools in order to write settings to a device in an interactive manual process [13]. By using these tools, properties like IPs on interfaces and static routing entries are configurable on certain devices.

Due to the limited capabilities of SNMP regarding the configuration of components, NETCONF has been created. The relatively new protocol specifically aims to allow the manipulation of the configuration of network devices. It uses either XML or JSON as encoding format and an RPC-based communication model. The Content layer which would specify the configuration data, is not part of the initial specification. [14] Instead a separate data format is created, named *YANG*. By using YANG, the configuration process is ideally decoupled from vendor specifics and can be used identically for all devices. This provides a major improvement over SNMP. Resulting from the conclusion of a 2002 meeting on network management technologies [13], NETCONF also allows transactions over multiple devices in order to provide error recovery in case a configuration process fails. As a downside, NETCONF has not been adopted by many vendors, yet.

3.4.3 Automated planning and scheduling

Research question **Q4** introduced the issue of getting a working execution plan for testbed setup steps. This problem can be viewed as an application of *automated planning and scheduling*. Use cases for automated planning and scheduling are problems in which the initial situation and the goal state can be formally described. This is done assigning attributes in the form of predicates to all objects that are part of a problem domain. In addition to that, a set of actions is necessary. Actions alter predicates of objects and can be put into sequence for transitioning from the initial state to the goal state. The process of finding such a sequence from a given set of actions is referred to as *planning* [15]. In the context of this thesis, the states are the current and desired new configuration of a testbed, with the setup steps being the actions that can be executed.

For our scenario, we are dealing with *classical planning problems* (others are *temporal* or *probabilistic planning problems*). For classical problems, the initial state is known, the number of states is finite, the system is fully observable, all actions are deterministic, the system cannot change without actions, the solving plan is a finite and linearly ordered sequence of actions and all actions can be considered without a duration. [16]

A planning problem is described by a planning language. One widely used language is the *Planning Domain Definition Language (PDDL)*, currently in its latest version PDDL 3.1. In case of PDDL, a problem is separated into a *domain* file and a *problem* file [17]. The domain file can be used for multiple problems within the same domain. It specifies, most importantly, the types and predicates objects can have as well as all actions available. Each action has a definition of variables it works on, preconditions that have to be fulfilled and an effect that alters the predicates of objects. The problem file describes all objects, their (optional) type and their initial state. Inside a goal section, the problem specifies the goal state of all objects by giving their desired predicates. [18]

All problems that can be broken down in the described way are suitable for being solved by planning. The types available for objects can be used to model inheritance allowing actions to be performed on all objects of a base class. With newer PDDL versions, numeric values that can be increased and decreased are available. This enables the tracking of costs as each action can be assigned with a certain cost value that adds up to a global counter. Planners supporting this feature can then minimize this value when searching for a plan. Predicates can be assigned to multiple objects at once, allowing to create relationships between objects by setting a predicate. One way to leverage this feature is by modeling a *part of* relationship between objects like a network and a host.

Both, the domain and the problem file, act as input for a planner (see Figure 3.1). A planner is a program specifically designed to read PDDL and generate an ordered sequence of actions which can be used to transition from the initial state to the goal state for the exact objects given in the problem file.

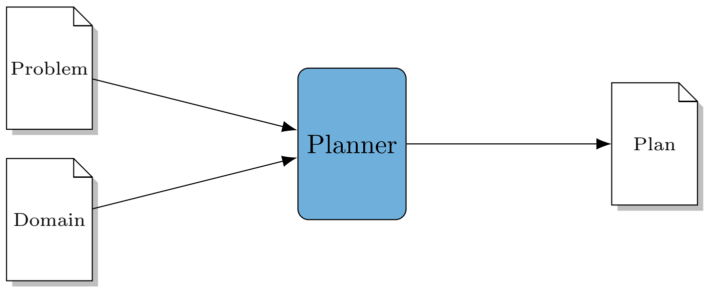


Figure 3.1: Relation between domain, problem, planner and plan in PDDL.

Chapter 4

Related work

This section gives an overview over related work about testbed management frameworks and their capabilities in respect to the requirements stated in Section 3.3. The requirements regarding the description language are compared to existing standard network description languages.

4.1 Data formats for network description

In order to find a suitable data format that is capable of representing the configuration of a testbed up to the level of complexity specified in the requirements, this section assesses existing network description languages and their features.

4.1.1 NML

The *Network Markup Language (NML)* [19] aims to provide a standard schema for exchanging network topology. It is either represented using XML syntax or an OWL RDF/XML syntax. NML revolves around network objects which are sub classed into the topology components *Node*, *Port*, *Link*, *Service* and *Group*.

A *Node* is a generic representation of all elements like hosts, routers but also lower level components like switches. They can have multiple *Ports* which model network interfaces. Networks are not represented by an object, instead they are created by using *Links* to connect the *Ports* of multiple *Nodes*. The *Services* are further sub classed and describe very basic mechanisms of the network, like for example a *SwitchingService* which models the behavior of a network switch.

It is explicitly stated that the current NML schema does not offer classes or properties to describe routing. It also misses components like firewalls or DNS. As it aims for a very generic network description, it does also not cover more detailed properties of a node like a machine's hardware specification. NML encourages the extension of the basic classes it provides. However, extensions need to be defined by an additional schema which is possible in any XML-based description language.

4.1.2 INDL

The *Infrastructure and Network Description Language (INDL)* [20] extends the ideas of NML and reuses the NML model as a basis for a description language that includes more information about the computing devices in addition to the network infrastructure.

A major improvement regarding the use case in this thesis, is the introduction of virtualization in INDL using the concept of a *VirtualNode*. In INDL *Nodes* consist of *NodeComponents* which have concrete implementations in the form of a *MemoryComponent* or a *ProcessorComponent*. This allows modeling virtual infrastructure like it is needed for the targeted testbeds.

Both, physical and virtual hosts including an arbitrary number of interfaces can be specified. However, INDL lacks classes for more specific network elements like routing or firewalls. For interfaces specifically, it also misses properties for MAC and IP addresses.

4.1.3 Tcl-based format in Emulab and ns-2

Emulab, which will be discussed regarding its capabilities as a testbed management software in Section 4.2.6, uses an extended form of the *ns* format. This is a Tcl-based format original defined by the network simulator *ns-2*. As *ns* is able to ignore unknown commands, Emulab claims the extended format is largely compatible and can be used for both systems.

Like the aforementioned formats NML and IDNL, the Emulab format does not specify networks directly and relies on links between nodes to implicitly form networks. This results in a verbose description for networks with many clients. It also doesn't allow to store information that is directly related to a network like the network address or the subnet mask.

Due to its relation to the description format of a network simulator, it allows specification of details like loss rates which are unlikely to be supported by other software.

While it supports key features for the automated setup process like routing or firewalls directly, in terms of services and all additional client-side configuration, userscripts can be defined.

A major drawback of the Emulab format is the lack of predefined parsers for common programming languages. In contrast to that, JSON and XML parsing is well supported.

Therefore, using JSON- or XML-based formats won't require to write a separate parser. In the case of XML, validation against a schema is well supported.

4.1.4 Conclusion about data formats

By looking at existing network description languages, we observe that there is no solution that fits our needs exactly. XML-based formats like NML and INDL can be extended by an additional schema, but a large amount of generic and unnecessary elements remain unused. On top of that, namespaces have to be introduced for an extension which hurts the usability as it makes the files less readable and less clear for the user.

4.2 Testbed management systems

Using all requirements, this section discusses the capabilities offered by related testbed management tools. As we want sharable configurations to be created, the existence of a standard input format is still taken into consideration. An overview of the fulfilled requirements listed in tabular form is given in Table 4.1.

Before discussing the related systems one by one, there are two more distantly related testbed approaches we want to mention briefly.

There are a few popular global scale testbeds like *PlanetLab* [21] or *GENI* [22] which focus on providing large distributed testbeds for Internet experimentation. As they fulfill an entirely different purpose by connecting multiple independently hosted testbeds and do not provide means for infrastructure changes, they aren't taken into further consideration.

In addition to that, Software-defined networks like *Mininet* [23] or network simulators like *CORE* [24] would provide an easily scriptable basis for the targeted testbed setup. However, in addition to the drawbacks of simulators described in Chapter 2, they share a common property which is problematic in our scenario. Due to the lightweight virtualization used, all components share the filesystem of the underlying physical host and don't have separated persistent disks.

4.2.1 LaasNetExp

LaasNetExp [1] is a project of the French Laboratory for Analysis and Architecture of Systems (LAAS) with the goal to create a generic polymorphic platform for network emulation and experiments.

The platform uses a central management server and relies solely on a fixed number of physical experimentation machines. These, initially 38, machines are equipped with four interfaces each. Two of these interfaces are associated with an emulation network that can be reconfigured dynamically by creating multiple VLANs on a hardware switch. To configure the physical machines, the system uses a PXE server containing preconfigured images of all required operating systems. As it is not described how routing is covered and because the number of machines and network interfaces is limited for the physical components, the dynamic allocation of basic network components as described in requirement **R1** is not provided.

While LaasNetExp allows to save the set of VLANs to restore an experiment's topology, there is no insight given on how information about the images associated with the machines is stored. Thus, leaving the question whether or not there is a description language.

As LaasNetExp is neither open source nor publicly available, it cannot be evaluated whether or not the source code is extensible enough to easily switch to a different mechanism for setup and configuration other than PXE.

LaasNetExp's machines can be separated and used by multiple testbed users conducting independent experiments. By keeping emulated networks private and controlling the traffic, they intentionally prevent machines from being part of multiple setups. This, however, also allows them to provide complete separation and enable different users to run their experiments simultaneously.

4.2.2 vBET

The virtual machine based *vBET* [6] aims to provide a versatile and scalable emulation testbed. It can be used on a powerful desktop or server for the deployment of research environments. However, the execution of topology creation is currently limited to a single vBET server.

With vBET it is possible to create hosts, networks, routers, switches and firewalls. Thus, all basic network components besides services are available. All machines are user-configurable, as node customizability is described as a key design goal. Each component is represented by a virtual machine. The introduction of physical machines, however, is not mentioned. The modeling language for network topology vBET uses, is based on the syntax used by the simulator *ns-2* (see Section 4.1.3). While being simplified in terms of readability, it provides all necessary information in order to specify networks, hosts, routers, switches and firewalls.

For virtualization vBet relies on *User-mode Linux (UML)* which includes capabilities for virtual networking.

In terms of resource isolation, vBET implements features in order to separate the network traffic, however, due to the use of User-mode Linux for virtualization, disk isolation is not achieved. On top of that, updates are not performed, instead a teardown of existing components is initiated in case a new topology is deployed. This is practical due to the lightweight virtualization and fast setup.

A major drawback of vBET is that there is no source code or download for the tool itself available to the public.

4.2.3 Baltikum Testbed

The *Baltikum testbed* (described in [25]) is a research testbed consisting of ten servers, multiple manageable switches and an energy management unit for energy efficiency measurements. The testbed uses a central management machine in order to control network experiments. The testbed was originally designed to utilize the physical hardware and allow dynamic reconfiguration as part of the experiment's scripts. A thesis by Stefan Kreuzer [25] proposed a solution to allow the dynamic creation of virtual machines as part of an experiment's setup phase. Before, the testbed was already able to reconfigure physical hosts by booting them with a variety of different images.

The integration of virtual machines into the Baltikum testbed was done by extending the already existing XML format which is used to specify testbed setups. This already shows the importance of an extensible data format like specified as requirement **R2**. With the addition of virtual machines, the testbed is able to manage both physical and virtual hardware, using both equally during experiments.

The solution for virtual machine creation uses *libvirt* which supports a large number of commonly used hypervisors. For networks, *Open vSwitch* is used exclusively. Regarding the supported operating systems, no claims are made, however, as templates are used and shell scripts are expected to carry out software configurations, all Linux distributions seem possible.

As the setup process allows starting arbitrary scripts on both, virtual and physical hosts after creation, the configuration of services is theoretically possible. Networks and an unlimited number of virtual interfaces are also available. However, routing, firewalls, DNS and DHCP are not covered in any way. *Open vSwitch* is technically capable of handling routing as well, but there are no XML elements reserved for specifying routing. The same goes for service related settings.

Much like the configuration of all physical hosts, the setup of virtual machines is part of the experiment execution itself. Therefore, there is no mechanism to handle and modify existing machines.

4.2.4 NEPTUNE

NEPTUNE [26] targets the deployment of virtual network testbeds using *OMF (cOntrol and Management Framework)* to automate the experimental procedure. The tool can create distributed virtual testbeds over a cluster of multiple physical machines. These machines are running hypervisor software and are controlled by a central management entity. Connecting physical components to the testbeds is not covered by NEPTUNE. The virtual machines are setup by cloning a predefined template from a given repository. This creates persistent and separated machines that can be configured and shared by multiple experiments and testbed users.

Network topologies are configured using a web frontend and can be saved in a standardized XML format. The XML format also allows to specify a certain predefined set of software (e. g. an Apache web server) to be installed on the template. In terms of firewalls, the Ethernet frame filtering utility *ebtables* is used. This is commonly combined with *iptables* in order to provide a full-fledged firewall. However, a review of the code did not indicate any support for *iptables*. Routing is also not covered in any way.

The aforementioned capabilities are largely determined by a direct review of the publicly available source code and sample configuration files of NEPTUNE. This was the only source of information besides the cited poster abstract, as the documentation of NEPTUNE is exclusively Italian. However, this may not always provide precise information about the capabilities of NEPTUNE. In addition to that, the limited accessibility of the documentation is a downside in terms of extensibility.

While NEPTUNE uses *Xen* as a hypervisor for virtualization, a brief review of the code shows that the design would allow for a different hypervisor to be added. All calls specific for a certain hypervisor are encapsulated in a single class implementing a *Connector* interface, therefore, decreasing the coupling between the hypervisor in use and the rest of the management tool.

Like most testbed management tools, there is no update functionality for making changes to existing machines without a recreation from scratch.

4.2.5 Algorizmi

Algorizmi [2] is an open-source tool for the setup of virtual testbeds which are used for generating datasets for the evaluation of intrusion detection systems (IDS). It is a Java application that was developed as part of a Master's Thesis at the University of Waterloo and focuses on virtual testbeds running on a multitude of physical hosts.

Algorizmi allows users to save the original configuration of an experiment in order for the setup to be reset and reproduced easily.

In terms of available virtual components, Algorizmi is able to create virtual machine instances based on user-defined images. A host is classified as either a *normal* or an *attacking* host. Depending on the user's choice for a certain virtual machine, different software is automatically installed. A brief review of the publicly available source code shows that the distinction between the two host types is a fixed part of the source code. The software to install on either of the hosts can be changed using the graphical user interface of Algorizmi, however, some basic installations are fixed and hard coded. The usage of `apt-get` also reveals a restriction in terms of Linux distributions. Additional network components like routers or servers providing network services like DNS or DHCP are not covered by Algorizmi.

The design proposal includes a database with account management, making the platform well suited for multi-user setups. Each user can request virtual machines in a certain configuration, whereas all experiments run in their own virtual networks. This way, users and their experiments are completely separated from each other.

The setup of virtual machines upon user requests and the management of users is solved by using the *Eucalyptus framework*. Eucalyptus is an open-source framework for cloud computing providing infrastructure as a service. Eucalyptus is compatible with *Amazon EC2*. Therefore, the Java library for EC2 is used and strictly coupled with Algorizmi's architecture. With Eucalyptus, *KVM*, *Xen* or *VMware* can be used as hypervisors for virtual machines. Reconfiguration is possible manually on each host or theoretically by changing a saved file and resetting the experiment. The latter couldn't be verified as the output format is not specified and could be a binary format.

4.2.6 Emulab

Emulab [27] is a large network testbed that has been actively developed by the University of Utah since 2000. The primary Emulab installation is run by the developers and open to the research community for registration. In addition to that, the open-source software of Emulab can be used to install and operate own Emulab instances. Emulab is also used as basis for other testbed projects like the cloud-based, closed-source *DETER* [28].

Designing a network topology is done by creating configuration files in the Tcl-based format that is also used by the network simulator *ns-2*. This format directly supports the setup and configuration of routing and firewall rules, the latter must be *iptables*. For specific services, hook scripts can be specified as part of the configuration file.

While Emulab originally focused solely on physical hosts, it has since implemented a support for virtual machines using *Xen*. A mixture of both physical and virtual components can be used. The original images for Emulab hosts are based on *Fedora* and provide a set of scripts to make them configurable. Custom images can also be deployed if needed, but have to be prepared with special software before.

Much like Algorizmi, Emulab is designed for multiple users which can request resources (network components) for an experiment which are then reserved for time of the experimentation phase. While all experiments are separated and users have persistent machines for the time they conduct an experiment, they are expected to swap-out an experiment once they are done. This mechanism saves the setup for later swap-in and frees up the machines for other experiments.

What Emulab is missing is a proper mechanism for reconfiguration of an experiment topology. Regarding modification of an existing topology, the Emulab FAQ states that any node with the same name in the old and new topology remains unaffected. If users wish to change hardware settings for the machine (e.g. add additional network interfaces), they have to name the machine differently in the new topology, thus creating an entirely new one. This means there are no fine granular changes like targeted with the solution proposed in this thesis.

4.2.7 Comparison

Table 4.1 gives a summary of all related work we've assessed regarding their compliance with the design requirements. As the overview shows, there is currently no testbed management system that fulfills all requirements we introduced. Emulab proves to be the most sophisticated system and shares the largest subset of requirements with the automated setup process proposed in this thesis.

Table 4.1: Comparison of related testbed management systems regarding their compliance with the design requirements.

Requirements	**LaasNetExp**	**vBET**	**Baltikum Testbed**	**NEPTUNE**	**Algorizmi**	**Emulab**
R1: Basic network components	✗	✓	✗	✗	✗	✓
R2: Description language	?	✓	✓	✓	✓	✓
R3: Virtual and physical infrastructure	✗	✗	✓	✗	✗	✓
R4: Software agnostic design	?	?	✗	✓	✗	✗
R5: Open source and public availability	✗	✗	✓	✓	✓	✓
R6: Separated persistent components	✓	✗	✓	✓	✓	✓
R7: Multi-user support	✓	✗	✗	✗	✓	✓
R8: Sharing components	✓	✓	✗	✓	✗	✗
R9: Continuous updates	✗	✗	✗	✗	✗	✗

✓– Requirement fulfilled, ✗– Requirement not fulfilled, ? – Not enough information

Chapter 5

Design

As the automated setup process is part of a larger system, this chapter first describes the complete system and the role the setup process has. Afterwards, the design of the different elements of the automated setup process is described in detail, thus, providing answers to the research questions prompted in Section 1.2. This includes the data model, how shared components are handled, how change detection is done and how the execution is planned.

5.1 Surrounding system and environment

Figure 5.1 shows the complete *IT NetworkS AnaLysis And deploymenT Application (short INSALATA)*, of which the automated setup process is part of. It consists of a central management unit which allows starting the deployment of a configuration by supplying the appropriate data. The central unit also manages a collector component which is covered in detail by [8]. The collector component is able to provide network information in the same format that is read by the automated setup process. We refer to this information as testbed state or configuration. This includes information about the entirety of all components including their settings. An input file that stores a testbed state is called a configuration file and is depicted as the user's upload in Figure 5.1. The results of the collector component can be stored as testbed states inside a database. Triggering the automated setup and configuration process for a certain testbed needs a configuration file. Besides being collected automatically, a configuration can also be created manually and uploaded by a user. The latter option helps user to supply completely new configurations as well as altering existing ones. The information collection component allows use cases in which information about a physical network is collected for deployment as a virtual testbed. By connecting the information collection processes and the automated setup, the management unit allows to automatically mirror changes into a testbed. The functionality of the collector component is also key when it comes to solving the problem of incremental updates, given as requirement **R9**. In order update a testbed to a new state, the current state of the targeted testbed is needed. This is solved by using the collector component on the targeted testbed directly before a deployment run.

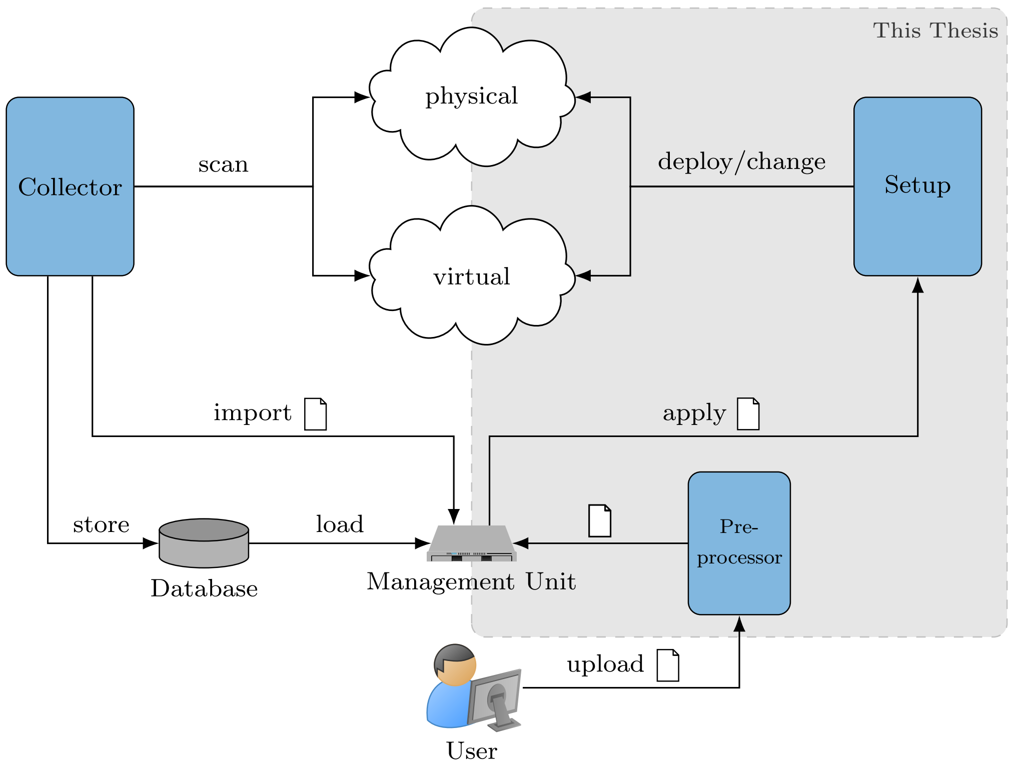


Figure 5.1: Overview of the environment.

To manage testbeds with INSALATA certain arrangements have to be made. All deployed components, virtual and physical, have to be connected to the central management unit through a management network. The management network is supposed to be equipped with a DHCP and DNS server. This allows to configure hosts using their hostname or to initiate a setup for physical components. To create hosts, templates with a configured interface into the management network have to be provided. The surroundings of host configuration are covered in more detail in Section 5.8.2.

5.2 The automated setup and configuration process

Figure 5.2 outlines the complete design of the automated setup and configuration process as it is proposed in this thesis. The role of each component and its details are covered in separate sections of this chapter. Nevertheless, we want to give an overview over the components and how they act together.

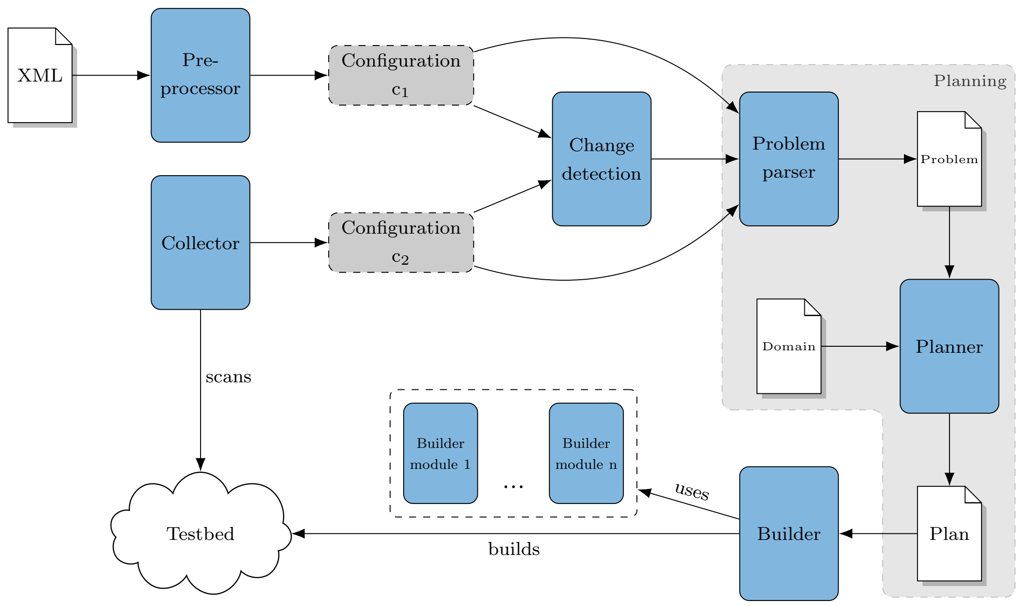


Figure 5.2: Flow diagram of the proposed setup process.

The process starts with the description of a testbed's goal state and aims to transform a targeted testbed according to it. We chose XML as the data format (see Section 5.4.2) to depict this goal state. The preprocessor takes this input, augments it if necessary and generates a representation in the internal data model (see Section 5.4.1). By utilizing the information collection component, the current state of the targeted testbed is gathered as internal data representation. The change detection mechanism determines the differences between these configurations (see Section 5.5). The configurations and its differences are processed by a problem parser (see Section 5.7.3) that expresses the current and goal state of each object as a formal problem file. The problem and a domain file that contains all constraints regarding the setup, are processed by a planner that creates an execution plan (see Section 5.7.3). A builder uses this plan to make all the determined changes to the targeted testbed, bringing it to the initially specified goal state (see Section 5.8).

5.3 Preprocessor

Configuration files supplied by the user are first given to a preprocessor which can validate the input data. This step includes verifying the existence of all identifiers and resolving references to shared components. Additionally, the preprocessor can determine missing data if it is calculable based on the given information. One example would be the range of addresses offered by a DHCP server. The range is needed for configuration and can be specified directly with the DHCP server. If no range is specified, by design, the complete range of addresses available in the network is assigned. Further optional properties which may be written by the preprocessor are mentioned when describing the different objects of the data model in detail in Section 5.4. The preprocessor helps users to save time while writing configuration files and prevents them from having to supply information twice, possibly with inconsistencies.

5.4 Data model

The data model for the automated setup process has to cover all elements defined in requirement **R1** including their properties. The following section covers the internal data model that contains all elements and properties as well as the external representation which is used for importing and exporting testbed states in respect to requirement **R2**.

5.4.1 Internal data model

We start by describing the internal data model used by the automated setup process internally, as the external representation results directly from it. The data model can be described as an object-oriented hierarchy. Instances of these classes are used after the input data format is parsed once. In Figure 5.2, the two instances `Configuration` c_1 and `Configuration` c_2 represent testbeds states using this data model. The class diagram in Figure 5.3 shows all elements with mandatory attributes being written in bold. Attributes handled by the preprocessor are written in italics.

Configuration—A configuration depicts a complete testbed state and acts as the top of the hierarchy. A configuration instance holds all elements that are either obtained by parsing an input file or result from a previous information collection process via the collector component. It holds direct references to all hosts, networks and locations, grouping them by a certain identifier. These identifiers are used to tag hosts and networks and help to keep track of the configurations a component is part of. This is especially important in order to determine when a host or network can be deleted in case of shared components (see **R8**).

Location—Besides hosts and networks, locations are also directly referenced by a configuration. The set of locations contains all location instances used by hosts and networks. A location can be a hypervisor that will be used for hosting a component or a special physical location for all non-virtual components. The identifier is used for referencing among hosts and networks. Next to the identifier, the type of a location is necessary. This is an abbreviation for a certain hypervisor software, described further in Section 5.8.1.

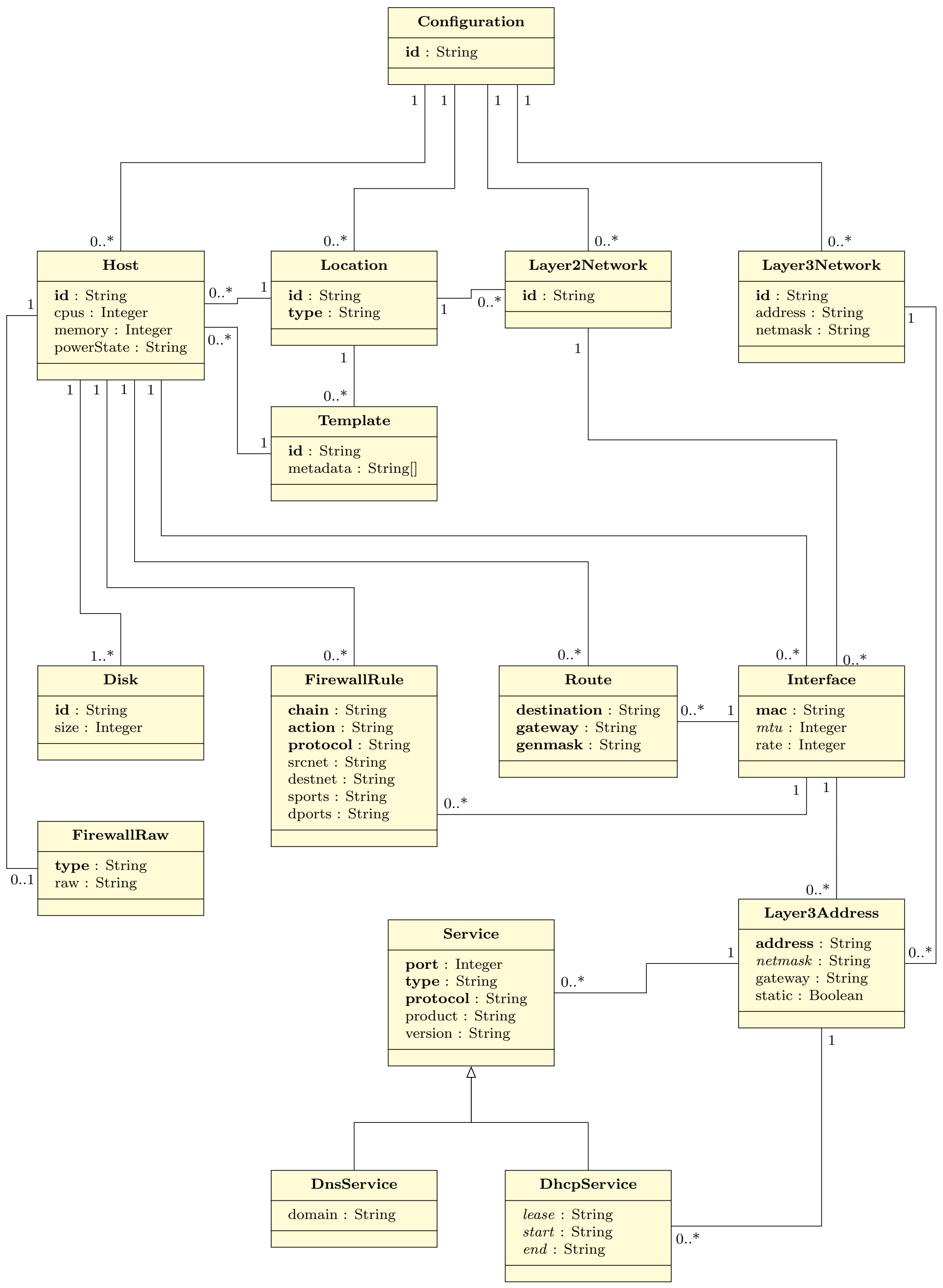


Figure 5.3: Class diagram of all elements covered by the setup process.

Layer2Network—As defined in **R1**, networks are a required component and therefore have to be depicted in the data model of the automated setup process. This includes both, layer 2 and layer 3 networks. Layer 2 networks state which hosts are connected to form a local network. Some of the related testbeds, like for example Emulab, chose to not model networks directly but rather the links between all hosts. However, modeling layer 2 networks allows to form networks just by referencing them by an identifier, making configuration files less verbose. The identifier can be an arbitrary but unique name.

In case the underlying technology (virtual, physical) used during the setup process follows a link-oriented approach, links can be created implicitly for all hosts grouped in the same layer 2 network. On the contrary, if the creation of a network object is necessary for the hypervisor in use, then it would be harder to identify all networks based on a list of links.

Layer3Network—Layer 3 networks are also modeled and referenced by layer 3 addresses. As the *Internet Protocol (IP)* is the commonly used layer 3 protocol, the key properties of IP networks are covered. Therefore, a network stores information like the network address and the netmask for all referencing layer 3 address configurations. The netmask allows to increase the network at a central point. This is especially useful for attached DHCP services which have a default range that covers all available addresses of the network. Like layer 2 networks, they are identified by an arbitrary but unique name.

Host—All components including routers and firewalls, no matter if they are virtual or physical, are represented as a host. This makes hosts an important and central concept of the data model. They are directly referenced by a configuration instance. Their identifier is used as their hostname and has to be unique in order to allow referencing.

Every host has an associated template which holds details for creating the machine. The usage of predefined templates in order to avoid a setup process is commonly used in testbed management software like the ones evaluated in Chapter 4. With the template, a host is also assigned a role that it fulfills in the network, like e.g. being a router or firewall. As routers and firewalls are viewed as hosts, a list of routing table entries and firewall rules may be associated with the host.

As necessary for virtual components, hosts can have a CPU count and amount of memory as properties. A reference to a location instance is mandatory and provides information about the target for deployment, like the type of hypervisor or that the host is a physical component. Each host holds a list of disks that are attached to it. The initial disk contains the operating system and is included implicitly. Additional disks can be used for storing large experimentation data or provide quick access to user-specific files.

Template—The template as it is referenced by a host is identified by a special name that matches the name of the template in the context of its location. This can be the name of the template machine on its hypervisor server or the name of an image file. The `metadata`-property is a flat list of strings that characterize a template. This contains information by the likes of the operating system, software that is available on this template and settings like routing. As these attributes are diverse and likely to be extended, we refrained from creating distinct properties for each one of them.

Interface—Another set of components held by a host instance are interfaces. Hosts can have an arbitrary number of network interfaces. The list of interfaces implicitly includes the interface to the central management network, as this is necessary by design for all hosts. Interfaces are identified by their MAC address making it a mandatory property for the user or at least the preprocessor to specify. This also means that changing a MAC address of an interface results in the interface being viewed as a new one. Each interface is linked to one layer 2 network, essentially modeling the cabling. For experimentation purposes, a limit on the network bandwidth and a value for the maximum transmission unit (MTU) can be set for each interface.

Since it is possible to assign multiple layer 3 addresses to an interface, at least in the case of IP, each interface references a list of addresses. Interfaces without any layer 3 addresses associated are viewed as DHCP interface.

Layer3Address—Layer 3 addresses are modeled by a distinct class. They are a connection point for services and mandatory in case of services being offered. Their main purpose is modeling statically configured layer 3 addresses on interfaces. A layer 3 address instance holds the IP address itself which acts as its identifier, the netmask and a gateway address. These attributes are drawn from IP as the most common layer 3 protocol. The latter properties are both optional. The netmask can be derived from the containing layer 3 network by the preprocessor. As the scanning component does collect addresses that have been assigned via DHCP, a flag marking an address as non-static is provided.

Service—A service acts as generic representation of application layer services and basic network services like DNS and DHCP. A service is associated with a layer 3 address, in whose context it is identified by the combination of a port number and the layer 4 protocol. If the same service is offered over multiple addresses or even interfaces, it is represented by separate instances. The concrete setup routine for each service has to take this separation into account, if necessary for the software.

An additional `type` is optional and can be used to provide a standard description of the service. The type is used in case no specific `product` is given and holds a generic identifier by the likes of *http* or *ftp*. In contrast, `product` is specifically designed to reference the name of the service's software. An optional version can provide even further level of detail and helps to realize scenarios in which an older version of a service is used deliberately.

DNS and DHCP services are modeled by subclasses of `Service` which include the properties needed for their respective configuration. Special application layer services like a web server can also be modeled using a subclass, if the generic properties of `Service` are insufficient.

In case of DNS and DHCP specifically, the first one only defines its domain. All other settings are given by the context of the associated layer 3 address. The DHCP service holds a range of addresses it can assign, as well as a lease time and a gateway address to announce. The range of addresses is optional and per default set to span the complete network associated with the layer 3 address of the service.

Disk—Disks are referenced by hosts and can be shared between them. They have a unique identifier which is either specified by the user or is created automatically by extending the unique hostname. The latter naming scheme is used for the initial disk part of every new host. Existing disks can be referenced to be connected to a host. If a new disk has to be created, a size value is necessary.

Route—An instance of the `Route` class models an entry of a routing table. They are associated with a network which is specified by an address and a netmask, a layer 3 address acting as destination and an interface. As IP is commonly used for layer 3 networks, this format is chosen for the data model as well. The gateway and destination network are not necessarily references to network instances, as routing tables may reference a network or hop that is outside of the scope of the given configuration. Only static routing is covered by the design of this data model. This is reasonable because it allows to configure every setup, especially faulty routes. With dynamic routing protocols, a certain state cannot be ensured. Nevertheless, static routing can be handled by a special routing configuration action.

FirewallRule—A single firewall rule is modeled by a distinct class with multiple instances being referenced by the host using them as a set. With this, only a single firewall per host is possible. As multiple software solutions for firewalls have to be supported, a generic representation is needed. Finding a common format for packet filtering rules is a complex task. Diekmann et al. [29] showed an approach to create a generic data format for firewall rules and created a tool that is capable of transforming firewall settings into this format. Their solution includes a simplification of the rule sets. The proposed format specifies an action which states whether a packet is accepted or dropped. Further information of the rule are its match conditions which can contain a source or destination network, a protocol like TCP, UDP or ICMP, a list of destination or source ports or an interface. These attributes are depicted by distinct properties in the class `FirewallRule`. The networks specified in a rule are not necessarily references as there can be rules referencing networks that are outside of the scope of the configuration instance. Destination ports are given in the format `start:end`. Rule sets generated by the tool are either an over- or an underapproximation. In practice, this leads to either more permissive or stricter rule sets respectively. It is possible to control which approximation is used when using the tool. Packet modification is not covered by the tool, as it sticks exclusively to access control. As the tool analyzes the rules of a specific chain, the chain a rule is part of is noted for each rule. [29]

In order mitigate the deficiencies of an approximation by the tool, saving the raw firewall dump in its software-specific syntax is also possible alongside the generic format. In cases where the identical software is present on the new host inside the testbed, the dump can simply be reinstated. The raw firewall rules are stored alongside the simplified format as an instance of `FirewallRaw`. This class combines the name of the firewall product with the raw firewall rules in the respective syntax.

A major advantage that comes with the generic data format is the option to migrate between different firewalls, switching from the one that has been scanned to another one used inside the testbed.

5.4.2 External representation

In our scenario, the external representation of the internal data model provides an interface for the user. XML is suitable for network description when comparing its capabilities with the requirements declared in **R2**. It is used to store testbed states by many of the related applications examined in Chapter 4.

All components of the automated setup process can be specified using distinct tags, while XML attributes store the detailed properties of each network component. A single XML file can be used to describe a complete testbed state. Files are kept reasonably small, sharable and, to a certain extent, human-readable. By nesting tags, *is in* relationships like interfaces being part of a host can be modeled intuitively. This way, the hierarchy specified in the data model is depicted.

While JSON has the aforementioned capabilities as well, when it comes to referencing an external element, XML provides an include mechanism that allows to import fragments from other sources. This gives XML a distinct advantage over JSON as a data format.

With the existence of a predefined XML schema, all input data can be verified. This, alongside parsing XML configuration files, is supported by standard libraries in most common programming languages.

5.5 Change detection

The mechanism for change detection between testbed states provides an answer to research question **Q3**. Detecting changes is an important feature for incremental updates like the automated setup process aims to provide in order to fulfill requirement **R9**.

The change detection process returns a data structure (e.g. a dictionary) that mirrors the structure and relation of the objects in the data model. For each distinct element and all its members, a keyword regarding how they changed is associated. This allows to quickly find changes between testbed states given by a configuration later on. By mirroring the nested data structure, elements without an identifier are unique in context of their surrounding element.

Changes are detected between two `Configuration` instances. One configuration holds the current state of the testbed, obtained by the information collecting module, while the second instance specifies the desired goal state. For the current state, only elements tagged with the same configuration identifier as the new configuration are considered. Other elements are completely ignored, ensuring that configurations of other users are not altered in any way. This is necessary in order to allow configurations of multiple users to coexist as declared by requirement **R7**.

The process of detecting changes has to be as generic as possible in order to avoid new code in case of additional elements or attributes being added to the data model. The process starts iterating over all members in the new `Configuration`-instance, finding corresponding ones in the current configuration by comparing names of the attributes. If a matching member can be found and therefore, has previously existed, all attributes of the attached objects are compared, noting possible changes. If there is no matching member present in the current state, the member and everything attached is considered to be new. On the contrary, all members of the current configuration have to be checked for a corresponding one in the goal state. If there is no matching equivalent being found in this case, the member is considered as being removed. This leads to the four different annotations of members the change detection can produce.

- unchanged (the attribute or element is identical in the old and new state)
- changed (the attribute or element changed between the old and new state)
- new (the attribute or element is not present in the old state)
- removed (the attribute or element is not present in the new state)

This approach is straightforward for simple data types like strings or integers which are compared directly. If the member is an enumeration, we use a different method. For complex types, the change detection method is called recursively. In case of unordered enumerations like sets, corresponding elements are determined by the element's identifier. If there is no identifier available, the whole set is compared directly. However, this method does not allow to track detailed differences in the distinct attributes of elements. An application of this fallback method are firewall rules and routing tables. For ordered enumerations, matching elements is done by using their index.

The changes detected on a nested element propagate upwards in the hierarchy. This means that a new or removed member or list element flags the containing object as `changed`.

5.6 Setup steps

This section defines the individual steps modeling the setup process (see research question **Q2**). For our solution, the steps are required to model actions to change properties of components introduced in the described data model. We aim to make the granularity of the steps fine enough to avoid unnecessary reconfiguration when there are no changes. However, we do not want steps to be too detailed to avoid large overhead. Instead, a reasonable balance that consolidates related actions into larger steps has to be found. The steps determined here are put into order by a planning component with respect to constraints between them. Table 5.1 shows a complete list of all steps grouped by their general purpose. Each step is discussed in detail subsequently.

Table 5.1: Steps identified during the testbed setup.

Category	**Steps**
Creating and removing elements	`createNetwork, createHost, createInterface, addDisk, removeNetwork, removeHost, removeInterface, removeDisk, deleteHost`
Managing configuration references	`addConfigNameNetwork, addConfigNameHost, addConfigNameDisk`
Configuring components	`configureCpus, configureMemory, configureInterface, configureService, configureNetwork, configureMtu, configureRate, configureRouting, configureFirewall, configureDns, configureDhcp, unconfigureInterface, unconfigureService, name`
Managing power states	`boot, bootAndNamed, bootUnnamed, shutdown, reboot, rebootAndNamed`

5.6.1 Creating and removing elements

Steps in this category manage the creation and removal of components. As physical hardware cannot be created or removed by code, like it is possible in a virtual environment, these steps are either omitted or perform an equivalent by setting up software on a physical component.

With respect to the data model, we identify the steps `createNetwork`, `createHost`, `createInterface`, `addDisk` as well as their counterparts for removal `removeNetwork`, `removeHost`, `removeInterface`, `removeDisk`.

The networks referred to are layer 2 networks. Layer 3 networks aren't explicitly created as they are a result of layer 3 address configuration on connected hosts. The step `createNetwork` is supposed to create a layer 2 network if networks are considered to be separate entities by the underlying hypervisor. If networking is done by introducing links between interfaces instead, this step is implemented doing nothing, as links are then introduced when `createInterface` is called. For physical components, this step can handle the configuration of a managed switch.

The creation of hosts and interfaces is straightforwardly handled by the eponymous steps. By design `createHost` encapsulates the allocation of hardware and the installation of an operating system. For physical hosts, the latter can be provided by a dedicated external component for network installation of operating systems.

The `addDisk` step is meant to add a disk to a host. If the disk to add does not exist, the step handles the creation as well. This is reasonable as referenced disks can be outside of the scope of the current configuration. Here it is difficult to determine whether or not a disk already exists without actively connecting to the hypervisor during the planning phase. In contrast, existing disks are easy consider when the setup process is running and commands on the hypervisor are executed for adding a disk.

Removing hosts with `removeHost` takes references from other configurations into account. If there are references besides the current configuration, only the reference to the current configuration is removed. In contrast, `deleteHost` removes a host regardless of existing references. The step accounts for the special circumstance occurring when the template of an existing host changes. In this case, the only way to apply the correct template is by recreating the machine from scratch.

5.6.2 Managing configuration references

Steps that remove elements like `removeHost` have to be able to check if the host is referenced by other configurations. Those references have to be stored in the first place. Therefore, `addConfigNameNetwork`, `addConfigNameHost` and `addConfigNameDisk` are called during the setup. The purpose of these steps is to simply add the name of the current configuration file to a list associated with the object, marking its usage.

5.6.3 Configuring components

This category includes all steps used to change properties of network elements. For hosts, there are `configureCpus`, `configureMemory`, `configureRouting`, `configureFirewall`, `configureInterface` and `name`. The first two steps are used for virtual hosts in order to change their hardware specifications. Setting a hostname is modeled by the step `name`. With `configureRouting` and `configureFirewall` the routing and firewall rules are applied to a host. As change detection on single rules is a difficult process due to missing identifiers and simply replacing rules is not time consuming in most cases, both steps are intended to replace the whole set of routing or firewall rules. This makes additional actions, like `addRoute` or `removeRoute` as well as actions for removing the configuration, obsolete.

The step `configureInterface` is executed for each previously created and attached interface. It includes assigning static addresses or preparing the interface for DHCP. As interfaces can also be removed from hosts, their configuration can be reverted which leads to a matching step `unconfigureInterface`. For physical components that are attached to a virtual network, the interface configuration includes configuring a connection into the virtual network (see Section 5.8.2).

The step `configureService` is introduced to install arbitrary application layer services on a host. Basic network services (e.g. DNS and DHCP) use distinct setup steps (`configureDns` and `configureDhcp`) to be configured before their clients. Removing the configuration of a specified services from a host is handled by `unconfigureService`.

For interfaces, it is possible to change the network they are part of (`configureNetwork`), set an MTU value (`configureMtu`) or specify a bandwidth limitation (`configureRate`). By reconfiguring the network, the user can essentially rewire virtual setups and move hosts to different layer 2 networks.

5.6.4 Managing power states

Steps for creating and configuring elements have constraints regarding the power state of the modified elements. One example would be a new interface that can only be attached to a host if the host is powered off. Therefore, dedicated steps to issue a boot or shut down are necessary. Available steps for power management are `boot`, `shutdown`, `bootAndNamed` and `bootUnnamed`. A reboot can be achieved by a combination of these.

The latter two steps need some additional explanation. As the setup process uses a central management network like described in 5.1, access to hosts during the configuration phase is done by leveraging unique hostnames. A newly created machine still has its template's hostname assigned. As booting requires the setup process to monitor the host and wait for it to be ready for access via the management network, the hostname is needed for all bootup steps. Therefore, `bootUnnamed` is used for hosts that are yet to be named (see `name`), allowing an implementation of the step that uses the template's hostname instead. As the first reboot after a host is renamed makes its name known to the DNS server of the management network, `bootAndNamed` is a dedicated step for booting used to specify that this host can now be reached by its name for the rest of the setup. By supplying different steps for this, the planning mechanism can react to these constraints.

5.7 Planning

Planning is a central part of the automated setup process. All components that are part of this functionality are highlighted in Figure 5.2. Planning takes the result of the change detection mechanism described in Section 5.5 and utilizes the individual setup steps we identified in Section 5.6. The planning mechanism is additionally supplied with both configuration instances holding the current state and goal state of the testbed respectively. Using this information, the planning component determines which steps are necessary to reach the goal state and how they can be ordered. Ordering steps is subject to constraints as certain setup and configuration steps are prerequisite to others.

This section identifies the constraints of the testbed domains that are targeted by the automated setup process and shows how automated planning and scheduling with PDDL described in Section 3.4.3 can be used to determine an execution plan containing the necessary setup steps. With this section, we provide an answer to research question **Q4** and fulfill requirement **R9**.

5.7.1 Constraints in the testbed domain

Not all setup steps can be executed at any arbitrary point in time during the setup. For example, a host has to be created and must be running before a certain configuration script can be executed on it. Constraints of this kind are diverse in the testbed's problem domain and are discussed subsequently. Not all the given constraints are mandatory to provide an executable plan. Instead, some constraints are added for optimization purposes. These cases are mentioned in the following descriptions.

Simple steps like the ones managing configuration references are not discussed in detail. The same goes for configuration steps like `configureRouting` that only act on eponymous predicates. For other more complex steps, a formal description of each step regarding its preconditions and result is given. The notation used is similar to operational semantics and separates preconditions and effects by a vertical line. The general syntax is:

$$\textit{<step name>}(\textit{<parameter>} \in \textit{<set>}) \colon \frac{\textit{preconditions}}{\textit{effects}}$$

Both the preconditions and the result are symbolized by predicates $P(x)$ on objects $x \in \mathcal{U}$ of the testbed configuration and combined by predicate logic symbols. We consider all objects as elements of $\mathcal{U}$ and define subsets for each type. These are $\mathcal{H}$ for hosts, $\mathcal{N}$ for networks, $\mathcal{I}$ for interfaces, $\mathcal{R}$ for routers, $\mathcal{D}$ for disks and $\mathcal{S}$ for services.

It is important to note that not all predicates used as preconditions are the result of other steps or demand that the step setting a certain predicate has to be executed. Some predicates, like for example *old(x)*, are exclusively set initially by the problem parser (see Figure 5.2) as a result of the changes detected. In the case of the *old(x)* predicate, the problem parser sets this predicate on objects that have been removed in the new goal setup. Other predicates may be set as well, depending on the current state of a component.

5.7.1.1 Initial steps regarding hosts and networks

If the setup is started from scratch, only `createHost` and `createNetwork` can act as a starting point due to the fact they do not have any preconditions.

$$\text{createHost(h} \in \mathcal{H}\text{):} \; \frac{}{created(h) \wedge cpusConfigured(h) \wedge memoryConfigured(h)}$$

$$\text{createNetwork(n} \in \mathcal{N}\text{):} \; \frac{}{created(n)}$$

The creation of a new host ensures that the hardware allocation of CPUs and memory is correct. These specifications can also be set by separate steps that require the machine to be halted. An additional step available on created machines is renaming them, as they are assigned with a template's hostname directly after being created.

$$\text{configureCpus(h} \in \mathcal{H}\text{):} \; \frac{created(h) \wedge \neg running(h)}{cpusConfigured(h)}$$

$$\text{configureMemory(h} \in \mathcal{H}\text{):} \; \frac{created(h) \wedge \neg running(h)}{memoryConfigured(h)}$$

$$\text{name(h} \in \mathcal{H}\text{):} \; \frac{running(h)}{named(h)}$$

Only a host that has been created can be booted. Otherwise the planner would likely issue a bootup to get a host running without the host having been created before.

$$\text{boot(h} \in \mathcal{H}\text{):} \; \frac{created(h) \wedge \neg running(h)}{running(h)} \qquad \text{shutdown(h} \in \mathcal{H}\text{):} \; \frac{running(h)}{\neg running(h)}$$

Note that the step `boot` here represents all the different booting steps listed in Table 5.1 and acts as a simplification to outline the principle constraints. Issuing a shutdown has the reverse effect than booting, but does not have to explicitly state that the targeted host must be created, as the fact that it is running implies the existence already.

5.7.1.2 Complex steps affected by other components

The following, more complex steps, are characterized by the fact that not only the direct parameters are considered in the precondition. For some steps, it is necessary to formulate dependencies about the element that contains the parameter object. When such an object has to be in a specific state, these steps make use of the *part of* predicate in conjunction with the universal quantifier.

$$\text{addDisk(d} \in \mathcal{D}\text{, h} \in \mathcal{H}\text{):}\ \frac{\neg attached(d,h) \land \forall h \in \mathcal{H} : part\text{-}of(d,h) \implies created(h)}{attached(d,h)}$$

Creating and attaching a hard disk is encapsulated in the step `addDisk`. It is required for the host the disk is part of, to be created before. The *part of* relationship used to filter for this host has to be given initially for all elements and does also appear for hosts and their interfaces.

The creation of a new network interface requires an existing network and host. The host has to be powered off due to the interface being also attached during this step. Hot swapping of network interfaces is usually not supported, making this the most generic and safest solution.

$$\text{createInterface(i} \in \mathcal{I}\text{):}\ \frac{\begin{gathered}\forall h \in \mathcal{H} : part\text{-}of(i,h) \implies (created(h) \land \neg running(h)) \\ \land \forall n \in \mathcal{N} : part\text{-}of(i,n) \implies created(n)\end{gathered}}{\begin{gathered}created(i) \land networkConfigured(i) \\ \land mtuConfigured(i) \land rateConfigured(i)\end{gathered}}$$

Much like for the hardware specifications of hosts, the properties of an interface like the attached network and its metrics, rate and MTU, are correctly set after creation. The respective steps can change these values after creation if necessary. Due to their simple structure, they are not listed here in more detail.

The configuration of an interface becomes available after creation and is done by one of two available steps. If all layer 3 addresses of an interface are configured statically, the configuration can simply be triggered on the host that contains the interface. This allows to setup central servers in a network.

$$\text{configureInterface(i} \in \mathcal{I}\text{):}\ \frac{\begin{gathered}static(i) \land created(i) \land networkConfigured(i) \\ \land mtuConfigured(i) \land rateConfigured(i) \\ \land \forall h \in \mathcal{H} : part\text{-}of(i,h) \implies running(h)\end{gathered}}{interfaceConfigured(i)}$$

If an interface relies on a DHCP service to get its addresses, it is beneficial to configure DHCP and DNS beforehand. This is a, not necessarily mandatory, optimization because hosts can then get their address configuration directly without any further reboots. They are also directly handled by DNS servers that act on top of the DHCP server. Therefore, this second step for configuring interfaces is limited to DHCP-dependent hosts and requires all DHCP and DNS services to be configured first.

$$\text{configureInterface}(i \in \mathcal{I}): \frac{\begin{gathered} \mathit{created}(i) \land \mathit{networkConfigured}(i) \\ \land\ \mathit{mtuConfigured}(i) \land \mathit{rateConfigured}(i) \\ \land\ \forall h \in \mathcal{H} : \mathit{part\text{-}of}(i,h) \implies \mathit{running}(h) \\ \land\ \forall s \in \mathcal{S}_{\text{DNS}} : \mathit{dnsConfigured}(s) \\ \land\ \forall s \in \mathcal{S}_{\text{DHCP}} : \mathit{dhcpConfigured}(s) \end{gathered}}{\mathit{interfaceConfigured}(i)}$$

This can also be limited to the DHCP and DNS services in the same subnetwork to achieve further optimization.

In order to configure DHCP and DNS, their interfaces have to be configured and the attached host must be running. Due to services being attached to a layer 3 address which is in itself part of an interface, the formal description of the correct host is rather complex.

$$\text{configureDns}(s \in \mathcal{S}_{\text{DNS}}): \frac{\begin{gathered} \forall i \in \mathcal{I} : \mathit{part\text{-}of}(s,i) \implies \\ (\mathit{created}(i) \land \mathit{interfaceConfigured}(i) \\ \land\ \forall h \in \mathcal{H} : \mathit{part\text{-}of}(i,h) \implies (\mathit{running}(h) \land \mathit{named}(h))) \end{gathered}}{\begin{gathered} \mathit{dnsConfigured}(s) \land \mathit{serviceConfigured}(s) \land \neg \mathit{new}(s) \\ \land\ \forall d \in \mathcal{S}_{\text{DNS}} : \mathit{new}(s) \implies (\neg \mathit{dnsConfigured}(d) \land \neg \mathit{new}(d)) \end{gathered}}$$

For the configuration of DNS servers, more complexity is added as other DNS servers might also need a reconfiguration due to a new server being introduced. To allow DNS servers to update their list of other known servers with a new one, a new DNS server causes the configuration of existing DNS servers to be removed. To prevent unnecessary reconfiguration in case of multiple new servers, only the first one that is considered as new will trigger this.

$$\text{configureDhcp}(s \in \mathcal{S}_{\text{DHCP}}): \frac{\begin{gathered} \forall i \in \mathcal{I} : \mathit{part\text{-}of}(s,i) \implies \\ (\mathit{created}(i) \land \mathit{interfaceConfigured}(i) \\ \land\ \forall h \in \mathcal{H} : \mathit{part\text{-}of}(i,h) \implies (\mathit{running}(h) \land \mathit{named}(h))) \end{gathered}}{\mathit{dhcpConfigured}(s) \land \mathit{serviceConfigured}(s)}$$

5.7.1.3 Steps for removal of components

The following definitions display the constraints regarding the removal of components. All components that are to be removed are flagged with a certain predicate `old`, which is set by the problem parser if the result of the change detection indicates that the element has been removed.

Removing a host can be done if the host has been shut down. The step automatically removes all of the host's interfaces with it.

$$\text{removeHost}(h \in \mathcal{H}): \frac{\mathit{old}(h) \land \neg \mathit{running}(h)}{\neg \mathit{created}(h) \land \forall i \in \mathcal{I} : \mathit{part\text{-}of}(i,h) \implies \neg \mathit{created}(i)}$$

The reverse limitation exists for networks, as all interfaces referencing the network to delete, have to be removed first.

$$\text{removeNetwork(n} \in \mathcal{N}\text{):}\ \frac{old(n) \wedge \forall i \in \mathcal{I} : part\text{-}of(i,n) \implies \neg created(i)}{\neg created(n)}$$

Interfaces are more complex in regard of removal. The host containing an interface has to be shut down for removal of the hardware component. Besides being marked as `¬created`, all settings performed on the interface are removed as well.

$$\text{removeInterface(i} \in \mathcal{I}\text{):}\ \frac{old(i) \wedge \forall h \in \mathcal{H} : part\text{-}of(i,h) \implies \neg running(i)}{\begin{array}{c}\neg created(i) \wedge \neg mtuConfigured(i) \\ \wedge \neg\ rateConfigured(i) \wedge \neg networkConfigured(i)\end{array}}$$

Removing an interface does also require the interface configuration to be removed on the associated host. The step is allowed for interfaces which have already been deleted and results in the *part of* relationship between the interface and its host to be removed.

$$\text{unconfigureInterface(i} \in \mathcal{I}\text{):}\ \frac{\begin{array}{c}\neg created(i) \wedge \neg interfaceConfigure(i) \\ \wedge \forall h \in \mathcal{H} : part\text{-}of(i,h) \implies running(i)\end{array}}{\begin{array}{c}\neg interfaceConfigured(i) \\ \wedge \forall h \in \mathcal{H} : part\text{-}of(i,h) \implies \neg part\text{-}of(i,h)\end{array}}$$

Much like interface configurations, services do also have to be removed from hosts if they are no longer needed. The step `unconfigureService` achieves that.

$$\text{unconfigureService(s} \in \mathcal{S}\text{):}\ \frac{\begin{array}{c}old(s) \wedge \forall i \in \mathcal{I} : part\text{-}of(s,i) \implies \\ (\forall h \in \mathcal{H} : part\text{-}of(i,h) \implies running(h))\end{array}}{\neg serviceConfigured(s)}$$

5.7.2 Goal state of all components

The goal state of all components in any problem is fixed and can be formally specified over the set of all objects $x \in \mathcal{U}$, the predicates $P(x)$ and the subsets of $\mathcal{U}$ for each type which we already introduced in the beginning of this section.

$$\begin{aligned}
\forall x \in \mathcal{U} &: old(x) \Rightarrow \neg created(x) \\
\forall n \in \mathcal{N} &: \neg old(n) \Rightarrow created(n) \\
\forall h \in \mathcal{H} &: \neg old(h) \Rightarrow created(h) \wedge running(h) \wedge named(h) \wedge cpusConfigured(h) \wedge \\
&\qquad memoryConfigured(h) \wedge firewallConfigured(h) \wedge \\
&\qquad \neg templateChanged(h) \\
\forall i \in \mathcal{I} &: old(i) \Rightarrow \neg interfaceConfigured(i) \\
\forall i \in \mathcal{I} &: \neg old(i) \Rightarrow created(i) \wedge interfaceConfigured(i) \wedge rateConfigured(i) \wedge \\
&\qquad mtuConfigured(i) \wedge networkConfigured(i) \\
\forall s \in \mathcal{S} &: \neg old(s) \Rightarrow serviceConfigured(s) \\
\forall s \in \mathcal{S} &: old(s) \Rightarrow \neg serviceConfigured(s) \\
\forall r \in \mathcal{R} &: \neg old(r) \Rightarrow routingConfigured(r)
\end{aligned}$$

$$\forall n \in \mathcal{N} : new(n) \Rightarrow configNameAdded(n)$$
$$\forall h \in \mathcal{H} : new(h) \Rightarrow configNameAdded(h)$$
$$\forall d \in \mathcal{D} : new(d) \Rightarrow configNameAdded(d)$$
$$\forall h \in \mathcal{H} : \forall d \in D : part\text{-}of(d, h) \Rightarrow attached(d, h)$$
$$\forall h \in \mathcal{H} : \forall d \in D : old(d) \Rightarrow \neg attached(d, h)$$

5.7.3 PDDL planning in the automated setup process

Vukovic and Hwang showed how automated planning can be used for the setup and configuration of servers in a cloud migration scenario [30]. Their usage of planning in a related context to the one of this thesis sparked the idea of using it in the automated setup and configuration process. The advantages of this design decision and how PDDL planning solves the specific problem of our application, are discussed subsequently.

With PDDL planning, a variety of planning tools are applicable. These tools are designed to be efficient and are being further developed. In addition to that, the complexity of the domain's constraints is encapsulated in an external entity, the domain file, and therefore decoupled from the code of the automated setup tool. With a separate domain file, changes can be easily made without dealing with the code base of the tool. It also allows to swap domains in case a certain environment has a different set of constraints.

An alternative solution to the problem would be the usage of a static hard coded procedure. However, this is less adaptable and hurts the extensibility as it makes changing constraints more complex. A custom solution would require a large amount of additional design and coding efforts while still being more prone to errors than the sophisticated solutions provided by public available planners.

As described in Section 3.4.3, PDDL relies on actions and predicates to create a plan. In our application, the steps listed in Section 5.6 translate directly to PDDL actions. In Section 5.7.1 we use predicates to indicate the result of steps. These predicates are also directly transferable to PDDL.

The planning mechanism can be separated in multiple parts which are highlighted in Figure 5.2. These parts are a custom coded problem parser and the external PDDL planner. The problem parser starts off by using the result of the change detection to determine the initial predicates of all objects that are part of the problem, formulating a problem file. With our predicates and the steps, we only model that changes have to happen and do not consider the respective values to assign. For example, we do not model a predicate giving the precise amount of memory a host currently has or a step setting the amount of memory for a host to a distinct value. Instead, a predicate indicates that a change in the regard of memory has to be made and a generic `configureMemory`-step is issued as a result. This way the remaining task for the planning mechanism is to find which steps are needed to transfer objects to their desired state and how they are ordered. This way, the planning preprocess developed in this thesis does not has to determine the necessary steps itself. By finding the initial predicates for all objects, an external planner solves the task of finding the necessary steps and ordering them.

By using predicates, relationships are set between elements. Predicates specify interfaces being part of a certain network, disks and interfaces being part of a host as well as services being associated with an interface. In addition to the predicates, all objects are assigned with a certain type. For the planning domain, the types are reduced to `Host` with subtypes `Router` and `Plain`, `Network` which is limited to layer 2, `Interface`, `Disk` as well as `Services` including subtypes for `DHCP` and `DNS`. This typing utilizes the inheritance capabilities of PDDL.

The resulting plan has to be parsed according to the characteristics of the planner's output. A generic `PlanParserBase` that can be subclassed for each planner is provided. The parser will make sure to translate all steps into function calls by matching their names. Each step receives a reference to the complete new configuration instance along with all parameters used by the PDDL action. The parser returns an ordered list of functions associated with their parameters. This requires a callable function for each step at a central connection point. All calls of steps in the plan go to their matching function in the `Builder` module whose role is described in Section 5.8.

5.8 Building the testbed

Utilizing the execution plan, the setup of the testbed is executed by a *builder unit*. This section describes, how the correct functions for setup and configuration are determined from the generic steps provided by the planner. Therefore, covering the final part of Figure 5.2 surrounding the builder and its modules. These modules can be extended and are used by the builder to setup and alter a testbed.

5.8.1 Builder modules

The builder unit provides matching functions for all setup steps and allows a plan to be translated into a list of function calls. Behind a single setup step, a variety of implementations *(builder modules)* are possible. For example, how a host is created when executing `createHost`, depends on whether it is a physical or virtual one and in the latter case which hypervisor is used. This calls for a generic way of matching setup steps with concrete implementations.

In order to find a suitable implementation, every execution of a step evaluates its parameters. The parameters of each function are identical to the ones formulated for the PDDL action (e.g. `createHost` receives a `Host` instance). We identify a total of four properties that are to be considered when picking an implementation:

- the name of the setup step (e.g. createHost, configureInterface, ...)
- the location of the parameter object (physical or type of the hypervisor server)
- the properties of the template (operating system, software, ...)
- the name of the service

Note that we refrain from using a template name as this would require many changes in case of a new template that is compatible with existing steps. While these properties are supplied by a step's parameters, the implementing functions do also have to specify these properties. Matching both enables the builder to find the correct implementation.

The name of the step a function implements is mandatory and has to match with the executing step to be taken into consideration. Other properties are optional if the implementation is not concerned with it, like for example `configureRouting` does not specify a service to work on. Which properties of the step's parameters are taken into consideration is filtered in the central implementation each step is automatically tied with after planning. The location is considered if hardware components are altered. Template properties are taken into consideration for configuration on hosts. The name of the service is only matched for steps acting on service objects.

When searching for the best implementation the one with the largest intersection between the properties of the step's parameters and the properties specified by the function is taken.

To clarify how this works, we look at an example for the step `configureRouting`. Let's assume there is an implementation for this step which specifies the following meta data:

- step name: *configureRouting*
- template: *ubuntu, router*

The configuration step works on every Ubuntu machines that acts as a router. A location is not needed due to the underlying hypervisor being irrelevant in this case. If routing is configured on a host, its template is taken into consideration and if it specifies that it's an Ubuntu template which has routing capabilities, the implementation gets two points. If another implementation specifies a different operating system but also the *router* flag, it will only receive a single point. This results in the higher graded implementation to be taken.

Users can also specify dedicated functions for single elements in an optional configuration file. This file has to associate an identifier of an object with the name of a step and a special function to execute. The building mechanism always looks into the file before conducting the described matching.

This design of the building process allows to extend the automated setup process for any hypervisor, operating system or software to use. It also allows the installation and configuration of services if a suitable building function is supplied.

5.8.2 Host configuration

The following section aims to give insight into the design decisions made for implementation of the individual setup steps.

The automated setup process uses templates when setting up new hosts. This way, every hosts starts as a copy of a predefined standard host acting as the template. In contrast to related testbed management tools like Emulab [27], these templates are lightweight in a sense that only a running SSH server and a preconfigured network interface connected to the management network are needed. With these simple preconfigurations, new templates can be introduced more easily.

The configuration of hosts is solved by using management tools as described in Section 3.4.1. The usage of a push-style tool is favorable, as this keeps templates clean from client software. While the design of the building functions also allows different approaches like for example triggering a shell script, the usage of a management tool is encouraged.

5.8.3 Virtual and physical infrastructure

The concept of the automated setup process is not limited to virtual infrastructure. For physical machines, the setup process assumes that the hardware allocation is finished. This covers CPU count, memory, disks and the number of interfaces. As a result, setup steps have to be implemented differently. For example, `createHost` starts directly with the installation of a certain image utilizing technologies like network booting.

A special case appears when network communication between virtual and physical components has to be provided. In this scenario, we assume that the hypervisor server hosting the virtual network and the physical component are part of the same local network.

One solution would be to leverage the widely available support for *virtual LANs (VLANs)* of hypervisors in combination with a software-based switch implementation like *Open vSwitch*. However, setting up rules to redirect traffic from the machine correctly into the virtual network require changes on the hypervisor server itself and hurts the portability of the solution.

An alternative visualized in Figure 5.4, is the introduction of a dedicated *virtual private network server (VPN)* that is created as part of the automated setup process for all networks with both virtual and physical components. A VPN server can be added to a configuration by the preprocessor as a host with a special template. This is done once the preprocessor recognizes a physical component being part of a virtual network. The creation of the machine then becomes part of the automated setup process and is carried out like for any other host. The VPN service itself is defined as a service and covers the creation of cryptographic keys for authenticating the physical client. With this setup, only the VPN server has to be bridged to be accessible by the physical component from outside the virtual network. The VPN will then ensure that the physical device is part of the virtual local network as intended.

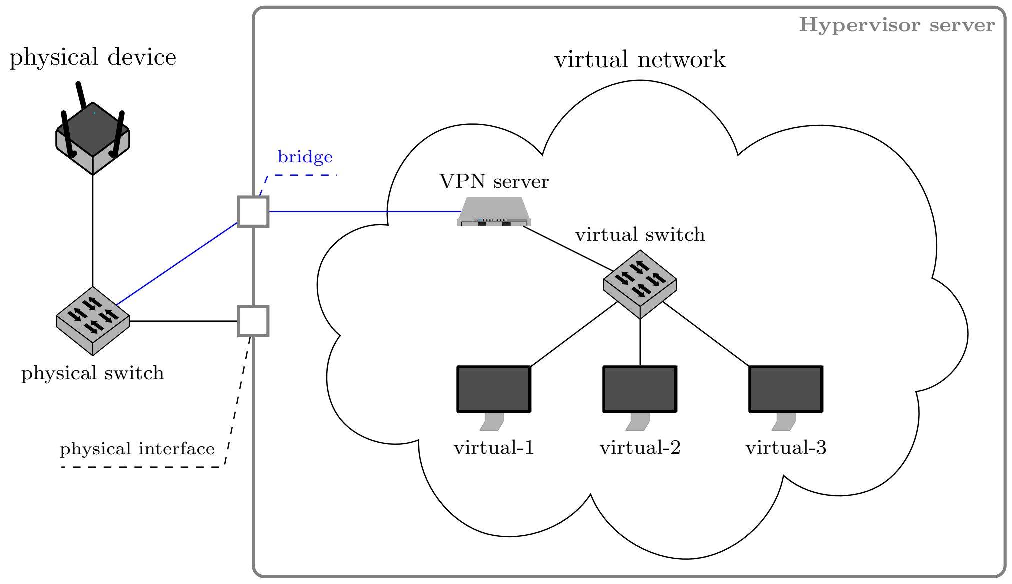


Figure 5.4: Connection of virtual and physical components via VPN.

Chapter 6

Implementation

This chapter gives insight into the tool implementing the automated setup process. This implementation is part of the *IT NetworkS AnaLysis And deploymenT Application (INSALATA).*

Python [31] is the programming language of choice for the implementation of *INSALATA*, as a variety of existing libraries can be employed. Python is highly portable as most common systems provide an interpreter. High-level language features and its multi-paradigm nature allow to write short and readable code.

Table 6.1 shows all of the tool's dependencies in regard of libraries or external applications.

Table 6.1: Dependencies of the automated setup and configuration tool.

Tool/Library	**Version**	**Reference**
Python	3.5.0+	[31]
lxml *(Python library)*	3.4.4	[32]
configobj *(Python library)*	5.0.6	[33]
netaddr *(Python library)*	0.7.18	[34]
Ansible	2.1.2.0	[11]
fast-downward	compiled from source on 2016-09-19	[35]

6.1 Management unit

The management unit is implemented as a service and runs on the management host of the targeted environment. The service combines the functionality of both, the automated setup and the information collection module and offers an API via XML-RPC. A terminal application that can connect to the service and use its API is provided. This client dynamically fetches all available API functions from the server. This way, introducing a new function does not require changes to the client.

The API currently provides functions for running scans, uploading configurations and triggering the automated setup process. All of the listed functionalities are available through the client. If the automated setup process is running, its current state can be retrieved via the client. Further API functions regarding the setup of new information collection runs are subject to future enhancements.

6.2 Data Model

As Python supports object-oriented programming, the data model described in Section 5.4 is implemented by using separate classes for all elements. This results in a class hierarchy identical to the one displayed in Figure 5.3, including inheritance between the different services and their base class. The exact same classes are also used by the information collection component. With this, the communication between both, the information collection component and the setup process, does not necessarily require the exchange of XML data. Instead, instances of testbed components can be shared directly. Besides mutator methods (get, set) for each property, every class provides a method for generating its own XML serialization.

Testbed configurations are serialized to XML using a schema that defines the structure, tags and attributes allowed (see Appendix A). A preprocessor is able to parse such an XML and create an instance of `Configuration`. The XML's root is a `config`-tag that directly includes nodes for layer 2 and layer 3 networks as well as for all hosts.

Listing 6.1 displays a small excerpt of an XML which defines a host with certain hardware specifications and two interfaces.

```xml
<hosts>
    <host id="host-1" cpus="2" memoryMax="2G" memoryMin="512M" powerState="Running"
        location="yggdrasil" template="host-base">
        <interfaces>
            <interface rate="125000" mac="de:ad:be:ef:81:42" network="network-1" />
            <interface rate="125000" mac="de:ad:be:ef:81:43" network="network-2" />
        </interfaces>
        <disks />
    </host>
    <!-- additional hosts -->
</hosts>
```

Listing 6.1: Small XML snippet containing a single host.

References that model a *part of* relationship are represented with nested tags. Associations like references to a network from an interface are expressed by using the network's identifier. Validation by the likes of *Schematron* can be used to validate the integrity of these references [36].

The sample in Listing 6.1 also shows properties like the rate limit for an interface which can be completely omitted. A user writing an XML definition manually does not necessarily has to specify a MAC address like shown in the listing. As it is a mandatory property of the interface, a new MAC address is then generated by the preprocessor instead.

The usage of XML as data format for testbed configurations allows referencing external components. In XML, this can be achieved by using the *XInclude* mechanism. An example is shown in Listing 6.2. The referenced host named *includeHost* has to be specified in the `otherConfig.xml` as shown in Listing 6.1. XML processing tools like `xmllint` [37] are able to resolve these references which is handled by the preprocessor in the tool.

```xml
<hosts xmlns:xi="http://www.w3.org/2001/XInclude">
    <xi:include href="otherConfig.xml" parse="xml"
        xpointer="xpointer(//hosts/host[@id='includeHost'])" />
    <!-- additional hosts (included or fully specified) -->
</hosts>
```

Listing 6.2: Hosts section of an XML importing an external host.

The overall system is supplied with additional static data via configuration files. They allow configuration of the running service but also provide static data important for the system. This includes the different available deployment locations as well as the templates that are available on each one of those. Listing 6.3 shows an example for a configuration file like this, displaying information about a Xen hypervisor server and its templates.

```
[yggdrasil]
hypervisor = xen
uri = http://yggdrasil.net.in.tum.de
login_id = root
login_pass = *****
xen_storage = Yggdrasil Storage
default_template = "host-base"
    [[templates]]
        [[[host-base]]]
        metadata = "ubuntu", "iptables", "systemd"
        [[[router-base]]]
        metadata = "ubuntu", "router", "dnsmasq", "iptables", "systemd"
```

Listing 6.3: Example of a location configuration file.

6.3 Change detection

The mechanism for change detection takes the current and a goal configuration as input and calculates their differences. In terms of the implementation, two instances of the `Configuration` class are given. Python provides strong reflection features based on the fact that classes are essentially dictionaries. This allows to iterate over all properties of an instance in a generic way.

Therefore, a single function `getMismatchingAttr(o1, o2)` is sufficient for comparing two objects of any kind. This function assumes both input values represent the same object in two different testbed states and compares every attribute of them. For lists of complex types, the routine is called recursively by looking at pairs of elements with a matching identifier. By starting from the instance representing the goal configuration, all elements of the configuration are considered. Removed elements are queried separately by iterating over the current configuration.

The result of the change detection routine is a dictionary itself. Each entry of the dictionary represents a single property. The value associated with each key is also a dictionary, providing the key `diff` with one of the four different types of changes that are specified in Section 5.5 as values. For simple types, no further information is provided and the dictionary with the `diff` key is only provided in this rather complex manner to be coherent with complex properties. In case of properties referring to a set or list, the dictionary contains an additional `elements`-key that contains tuples with the identifiers of each element and their respective change dictionary. An example that results from changes on the host from Listing 6.1 is displayed in Listing 6.4. For this list of changes, we removed the second interface of the user, changed the rate limit on the first interface and added a new disk. Note that the whole host is considered to have changed as one of its interfaces changed its rate limit and another one is removed.

```
{
   'hosts': [
      ('host-1', { 'diff': 'changed',
          'cpus': { 'diff': 'unchanged' }, 'memoryMin': { 'diff': 'unchanged' },
          'memoryMax': { 'diff': 'unchanged' }, 'powerState': { 'diff': 'unchanged' },
          'interfaces': {
             'diff': 'changed',
             'elements': [
                ('enxdeadbeef8142', {
                   'diff': 'changed', 'networkId': { 'diff': 'unchanged' }
                   'mtu': { 'diff': 'unchanged' }, 'rate': { 'diff': 'changed' }
                }),
                ('enxdeadbeef8143', {
                   'diff': 'removed', 'networkId': { 'diff': 'removed' }
                   'mtu': { 'diff': 'removed' }, 'rate': { 'diff': 'removed' }
                })
             ]
```

```
        },
        'disks': { 'diff': 'changed',
           'elements': [ ('host-1_disk-extra', { 'diff': 'new' }) ]
        }
    })
  ]
}
```

Listing 6.4: Dictionary with all differences between a host's states.

6.4 Planning

The dictionary with all changes is used to generate a PDDL problem file which is interpreted by a planner to generate the setup plan. The tool uses *fast-downward* [35], an implementation of the eponymous *Fast Downward Planning System* [38], as external planner. The planner implements a large amount of PDDL features and is still under active development by its originator Malte Helmert. Unlike many other planners, it is well documented. *Metric-FF* was also tested and used in the beginning of the implementation phase. However, it provides less features and is no longer being actively developed.

The problem file starts with listing all layer 2 networks, hosts, routers, interfaces and services that are part of either the current configuration or the goal configuration. Predicates are assigned based on the changes noted in the dictionary returned by the change detection mechanism. The goal state is fixed for all problems and is a PDDL representation of the formal description given in Section 5.7.1.

The full PDDL domain for the automated setup process is listed in Appendix B. At this point, we only want to give an idea of how the design from Section 5.7 translates to PDDL. Therefore, Listing 6.5 displays a simplified testbed domain with very few predicates and actions. The example is broken down to depict the update resulting from the changes detected in Listing 6.4. This small case covers enough components to give insight on how the different features of PDDL are used for the testbed domain in practice. The requirements listed for the domain describe certain PDDL features that have been introduced with different PDDL standards. A planner which is unable to support a feature in use, is supposed to notify the user based on this list.

The example contains only the four objects appearing in Listing 6.4 without any inheritance in their types. With `part-of` and `attached`, the example shows how two objects can be assigned with a predicate together. Actions for power management like `shutdown` only require the `running` predicate to be set appropriately. When removing an interface, the host has to be halted in our example. Therefore, the precondition defines that all hosts containing this interface have to be halted. This can be specified in PDDL by using a `forall` quantifier and an implication that only looks for the host the interface is part of. Many other predicates of the real domain are omitted, so not all properties of every object listed in Listing 6.4 are depicted in this example.

```
(define (domain simpleDomain)
    (:requirements :strips :typing :conditional-effects)
    (:types host interface disk)
    (:predicates (created ?x) (running ?h - host) (part-of ?i - interface ?h - host)
                 (attached ?d - disk ?h - host) (rateConfigured ?i - interface)
                 (old ?x))
    (:action shutdown
        :parameters (?h - host)
        :precondition (running ?h)
        :effect (not (running ?h)))
    (:action removeInterface
        :parameters (?i - interface)
        :precondition (and (old ?i)
                      (forall (?h - host) (imply (part-of ?i ?h) (not (running ?h)))))
        :effect (and (not (created ?i)) (not (rateConfigured ?i))))
    (:action addDisk
        :parameters (?d - disk ?h - host)
        :precondition (and (created ?h) (not (attached ?d ?h)))
        :effect (attached ?d ?h))
    (:action configureRate
        :parameters (?i - interface)
        :precondition (created ?i)
        :effect (rateConfigured ?i))
)
```

Listing 6.5: Simplified testbed domain in PDDL.

A simple problem in this domain is shown in Listing 6.6. Only one host, two interfaces and a disk are listed. The initial state ties `host-1` to its interfaces and marks one of them as *old*. The host is currently running. As negative predicates are not stated explicitly, the unattached disk is not mentioned in the initial state. The goal requires old interfaces to be *not created*, interfaces to have their rate limit configured and all disks to be attached.

```
(define (problem simpleProblem)
    (:domain simpleDomain)
    (:objects host-1 - host enxdeadbeef8142 - interface enxdeadbeef8142 - interface
        host-1_disk-extra - disk)
    (:init (part-of enxdeadbeef8142 host-1) (part-of enxdeadbeef8143 host-1)
           (running host-1) (old enxdeadbeef8143))
    (:goal
        (forall (?i - interface) (imply (old ?i) (not (created ?i))))
        (forall (?i - interface)
            (imply (not (old ?i)) (and (created ?i) (rateConfigured ?i))))
        (forall (?h - host)
            (forall (?d - disk) (imply (part-of ?d ?h) (attached ?d ?h))))
)
```

Listing 6.6: Simple problem in PDDL.

With this problem and the domain from Listing 6.5, the planner produces a simple plan shown in Figure 6.1. At first, the plan makes sure to shut down the running host before changing one interface and removing the other one. Afterwards, the disk is added to the halted host in order to match the third goal which states that all disks have to be attached.

```
shutdown host-1
configureRate enxdeadbeef8142
removeInterface enxdeadbeef8143
addDisk host-1_disk-extra
Plan length: 4 step(s).
```

Figure 6.1: Result of the simple planning problem.

All plans resulting from the planning step have the depicted string format. To encapsulate the usage of PDDL in a single module, the planning module parses this output and returns a Python list of tuples with function pointers and parameters.

6.5 Builder modules

The main building module supplies matching functions for each PDDL action and allows for a generic execution of the list returned by the planning module. By leveraging Python's tuple expansion as shown in Listing 6.7, an arbitrary number of parameters can be transferred from PDDL actions to a matching function.

```python
for step in plan:
    step[0](builder, self, *(step[1:]))
```

Listing 6.7: Python's tuple expansion used for parameters.

In Section 5.8.1 we described that all implementations of setup steps have to provide meta data based on which they are selected when executing the respective step. In Python, function decorators are a convenient way to store such additional information associated with a function. Decorators allow to tie meta data directly to the function and avoid a large mapping file instead. This solution also improves the extensibility, as the information is provided with the source code of the function itself. Making changes to the existing source code is not required when introducing a new implementation. Listing 6.8 shows the syntax used to add a decorator and therefore additional information to a function.

```python
@builderFor(action="configureInterface", template=["ubuntu"])
def configureInterfaceAnsibleDebian(logger, interface):
    #source code ...
```

Listing 6.8: Usage of Python decorators to provide meta data for a function.

The syntax shown is only a syntactic feature and refers to `builderFor` which implements the decorator pattern, shown in Listing 6.9. This is done by creating a new function that returns an altered version of the given function. The alternation made in this case is adding the four properties defined in Section 5.8.1.

```python
def builderFor(action, hypervisor=None, template=None, service=None):
    def decorate(f):
        f.action = action
        f.hypervisor = hypervisor
        f.template = template
        f.service = service
        return f
    return decorate
```

Listing 6.9: Implementation of the decorator in Python.

Every function implementing a setup step is required to specify the attribute `action`. In addition to that, the attributes `hypervisor`, `template` and `service` appear in an arbitrary combination. By design (see 5.8.1), templates are characterized by their attributes instead of their name. This way, introducing a new template does not require adding it to every decorator of every suitable function. Instead, existing functions can be considered for usage if the template states a matching operating system or software in its own meta data. The supplied information can be accessed on the function pointer like any other property. In combination with the reflection features provided by the libraries `importlib` and `pkgutil`, all functions inside the building module and its submodules can be queried and evaluated. Due to the way steps are queried, the function name is arbitrary and only used for documentation purposes. Finding the most suitable function is achieved by counting matching attributes like the design specifies in Section 5.8.1. This design translates directly into Python.

In addition to the generic way of finding the correct implementation for a setup step, deviations for special objects are possible and can be entered by the user via an additional file. This file is queried first when finding a setup step and tries to find an entry explicitly for the identifier of the object to work on. If the object's identifier and the name of the step are associated with a certain building function, it is used directly. The external file is written in the standard Python configuration file syntax which can be parsed by standard Python libraries.

```ini
[host-1]
createHost = verySpecialHostCreationFunction
```

Listing 6.10: Example of a config file for special cases.

6.6 Implementation of setup steps

The tool developed with this thesis provides a set of setup step implementations for certain hypervisors, operating systems and software. The Chair of Network Architectures and Services hosts two large virtualization servers running Xen [39] as a hypervisor. These servers are used for evaluation in Chapter 7. For this purpose, the creation of virtual machines is implemented for Xen. Editing virtual components is handled by the hypervisor's toolstack which is the *Xen Project Management API (XAPI)* [40] for the given servers. A sample command creating an interface directly on the hypervisor server is shown in Listing 6.11.

```
xe network-create name-label=test-network
```

Listing 6.11: Creating an interface with the XAPI toolstack.

The servers provide an XML-RPC interface which allows to execute commands remotely via a network connection to the hypervisor. Creating XML-RPC requests is conveniently handled by using Python. Listing 6.12 shows a remote call with the same effect as the direct toolstack command from Listing 6.11.

```python
import xmlrpc.client

xen = xmlrpc.client.Server("http://uri_to_server")
session = xen.session.login_with_password("username", "password")
netRecord = {
    'name_label': "test-network",
    'other_config': {}
}
xen.network.create(session, netRecord)
```

Listing 6.12: XML-RPC call to Xen via Python.

The implementation of configuration steps that act directly on a running host, is largely handled by using Ansible. The concept of the agent-less configuration management tool is briefly introduced in Section 3.4.1. Ansible provides playbooks for common software like the *iptables* firewall or the DNS/DHCP service *dnsmasq*. A small example is used here to provide some insight on how Ansible interacts with the automated setup tool.

The script to configure routing on a Debian-based host in Listing 6.13 is used to display the different features. Ansible scripts, called playbooks, are written in YAML syntax. They contain a sequence of steps to execute on a remote host.

```yaml
- hosts: "{{ target }}"
  user: root
  tasks:
  - name: Enable forwarding
    sysctl: name=net.ipv4.ip_forward value=1 state=present
  - name: Copy ip-up.d hook script
    copy: src=routing dest=/etc/network/if-up.d/routing owner=root mode="u=rwx"
  - name: Create static routes
    template: src=routes.j2 dest=/usr/local/config/routes owner=root mode="u=rwx"
  - name: Run the script once in order to setup the routes without reboot
    shell: /usr/local/config/routes
```

Listing 6.13: Ansible playbook for routing tables.

Every playbook acts on a set of hosts and takes the role of a certain user. The list of tasks contains the ordered sequence of commands to execute. The routing playbook enables routing and copies a hook script used for setting routing after a reboot.

The `template` command of Ansible copies a template to a remote machine and writes its content dynamically. Templates are written using the Jinja2 template engine which includes its own syntax. The idea of this playbook is to create a file that adds routing entries via the `ip` command. The Jinja2 template is shown in Listing 6.14

The variables used in YAML and the Jinja2 template, `target` and `routes`, are passed to Ansible in a JSON file with the respective variables as keys. This file is written by the respective building function before calling the Ansible playbook. Each element of `routes` holds the properties used inside the loop of the Jinja2 template.

```
{% for r in routes %}
   ip route add {{ r['network'] }}/{{ r['mask'] }} via {{ r['next'] }}
{% endfor %}
```

Listing 6.14: Jinja2 template for routing tables.

Chapter 7

Evaluation and case study

This chapter evaluates the features of the automated setup process and the implemented tool in respect to the requirements introduced in Section 3.3. To display its capabilities, a feature evaluation and a case study are conducted. These include the setup of a sample network and the application of incremental updates to it.

7.1 Evaluation

The evaluation refers back to the requirements and assesses their fulfillment. We take both, the design of the process and its implementation as the automated setup tool into consideration.

✓ *R1 Coverage of basic components*—The data model proposed in Section 5.4 covers all network components listed in the requirement. This includes hosts with interfaces and disks, networks (layer 2 and layer 3), routers, firewalls, DNS and DHCP servers as well as services.

Arbitrary services can be associated with an implementation of the `configureService`-step leveraging the mechanism described in Section 5.8.1. However, a generic way of installing services without a matching builder is not provided. The main issue preventing this is that there is no generic mapping of service product names and their package name under which they are available on all different systems and their repositories.

✓ *R2 Description language*—The tool implementing the automated setup process utilizes an XML format for serialization of testbed configurations. The XML format gives users a description language they can use to specify their own testbeds from scratch without any information collection on existing setups. The XML covers all elements of the data model and depicts *part of* relationships by nesting tags. XML is human-readable and can be easily validated based on the schema (see Appendix A) by standard libraries in most programming languages.

✓ *R3 Virtual and physical infrastructure*—The automated setup process is not limited to virtual environments. After machines are initially set up, the configuration phase of the automated setup does not differ for physical components and can, for the most part, utilize the same builder implementations. Section 5.8.3 covers how virtual and physical components can be combined in a testbed setup.

✓ *R4 Software agnostic design (extensibility)*—The design of the process does not rely on specific software, nor is it limited to certain platforms. This makes the design of the setup process software agnostic in the way its specified in the requirement. The separation of the generic setup plan from the implementation of setup steps make the design easily extensible. Switching between PDDL planners is supported by the design and only requires the implementation of a new output parser. All the application logic regarding PDDL is confined within a single module that can be replaced, if a solution for creating execution plans without PDDL is introduced.

✓ *R5 Open source and public availability*—The complete system including the tool implementing the automated setup process and the information collection component are made publicly available under the terms of the Apache License 2.0. The repository for the open-source implementation *INSALATA* is hosted on GitHub [41]. The planner fast-downward and Ansible are not included and have to be obtained if the respective modules are needed. Both tools, as well as all Python libraries used (see Table 6.1), are free software under either the GNU General Public License or the BSD License, making complete tool usable as free software.

✓ *R6 Separated persistent components*—The automated setup process creates separated persistent components by using virtual machine templates or physical components. Created components are part of the testbed until the removal is issued by all referencing users.

✓ *R7 Multi-user support*—Users can deploy infrastructure into a testbed without affecting others. Every configuration supplied by a user provides a unique identifier. All elements of this configuration are tagged with this identifier. If any updates or changes are made by applying a new configuration, only existing elements tagged with the identifier of the new configuration file are considered. If a new testbed user deploys completely new infrastructure without references to existing ones, the current configuration is essentially empty. Other components without this tag are not handed to the change detection mechanism in the first place.

✓ *R8 Sharing components*—In Section 6.2 we show how users can reference existing components defined in other configurations, in order to include them into their setup. Users are free to pick existing components. It is important to note that by referencing other components, including networks, users can affect each other when running experiments.

✓ *R9 Continuous updates*—Allowing continuous updates of the deployed testbed configurations is the central feature of the automated setup and configuration process. The change detection mechanism allows to consider existing components. This requirement is also the main reason for planning being part of the process. In our case, planning solves the problem of finding the correct steps for reconfiguration and putting them into order. By extending the granularity of the setup steps to even more detail, the feature can be further improved, as less configuration is redundantly applied with more fine granular steps.

7.2 Feature evaluation

In this section, we use the tool implementing the automated setup process to deploy a testbed configuration. We use a hypervisor server called *yggdrasil*, at the Chair of Network Architectures and Services and set up a testing infrastructure like it would be created by a student or researcher as a basis for network experiments. The example is designed to display all features of the automated setup and configuration process.

The feature evaluation is structured as follows:

1. Setup of a network infrastructure including time measurement and qualitative evaluation of the setup by testing its correctness.
2. Reconfiguration of the setup including time measurement and qualitative evaluation of the reconfigured setup by testing its correctness.

7.2.1 Setup of the initial network infrastructure

We start with a goal testbed configuration in form of an XML that has been written manually (see Appendix C.1). A visualization of this setup is shown in Figure 7.1.

The setup consists of two routers that form a total of three networks. The networks `test-net-1` and `test-net-2` contain two hosts each. An additional third network

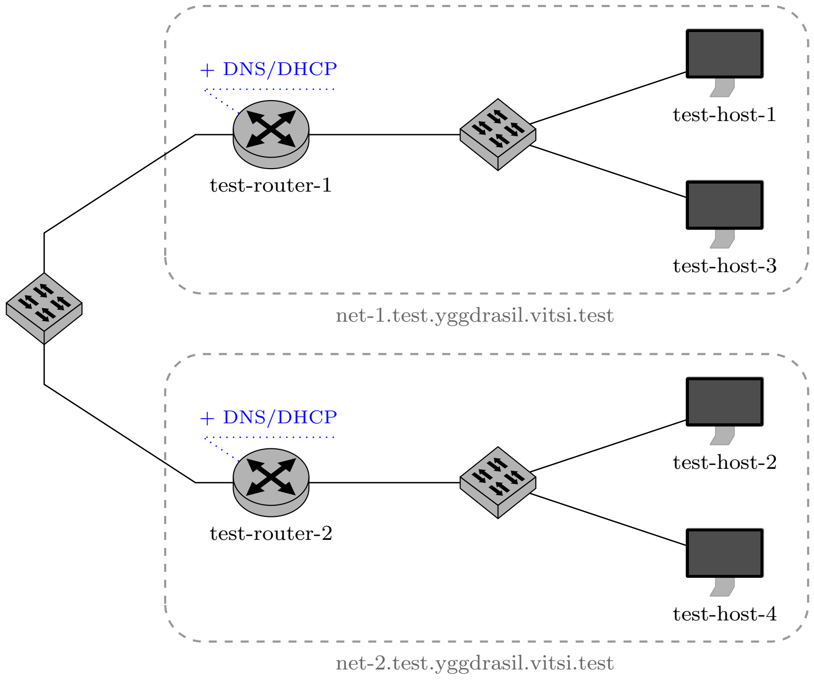


Figure 7.1: Visualization of the initial goal configuration in the feature evaluation.

`test-transport` connects both routers. The routers act as DHCP servers for the hosts in their direct subnetworks. They also host a DNS server which assigns a domain to the subnetworks. The transportation network between the routers has no DHCP or DNS server and is configured statically.

The terminal client is used to upload the XML file to the service and to trigger the automated setup and configuration process. As there are no elements tagged with the identifier of this configuration (`test`), the information collection module returns an empty set of network components. Therefore, the change detection mechanism marks every element of the configuration as being new.

7.2.2 Quantitative analysis of the setup process

The plan that is determined for setting up this initial configuration has 66 steps and is depicted in Appendix C.2. As heuristics are involved in creating the plan, it is not guaranteed to be ideal, nor is it the only valid solution. However, the plan is certain to take all constraints of the setup process given in Section 5.7.1 into account.

For this evaluation, we measure how much time the execution of the complete plan takes. To give an idea of how long each step takes proportionally to the whole setup, Figure 7.2 visualizes the execution process in regard to the duration of each step. Python's own `time.time()` function is used for time measurement.

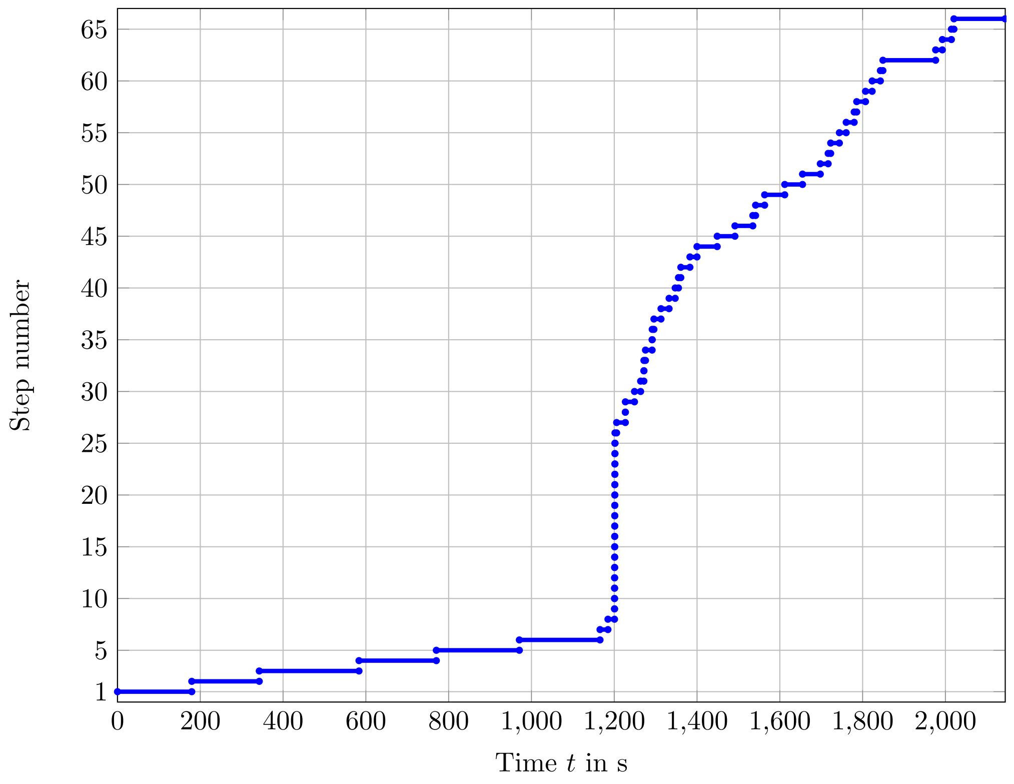


Figure 7.2: Time distribution of the initial setup steps in the feature evaluation.

Planning of the 66 setup steps takes the planner, fast-downward, 0.077 s. Building the testbed as visualized in Figure 7.2 takes 35 min 44 s. The time measurement was always repeated 4 times for this feature evaluation. In this case here, we observe a deviation of ± 3 min around the representative measurement displayed. These times can be put into perspective with the hardware specifications of the involved machines:

- Planning and orchestration on *fensalir*:
  4x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz, 8 GB RAM, Ubuntu 15.10, virtual
- Hypervisor server on *yggdrasil*:
  24x Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10 GHz, 128 GB RAM, Debian 7.6

Figure 7.2 shows that the majority of the time is consumed by creating hosts (steps 1–6). As this is a Xen-based setup, the respective building functions for this hypervisor are used. For Xen, creating a host encapsulates the process of cloning an existing template machine. This already includes that new hosts are equipped with an operating system. To create persistent hosts, a full clone is conducted. The time this takes depends on the size of the template's hard disk. In this scenario, each machine is provided with a 7 GB HDD. Observations during development showed that the time increases linear with larger disk sizes. Depending on the technology used for implementing the creation of hosts, the proportions of this step can be different. Nevertheless, creating a new host including the setup of an operating system is likely to take the most time out of all single steps in a setup. Step 62 and step 66, which took an extraordinary amount of time, are reboot operations regarding `test-host-3` and `test-host-4`. We observed this behavior on a few occasions and can ascribe it to Ubuntu 16.04 which is running as an operating system. The system sometimes runs into issues while shutting down a system service and takes longer than expected. This is not related to the automated setup and configuration process in any way and can be ignored.

7.2.3 Qualitative evaluation of the setup process

In order to assess whether or not the setup was successful, a few functional tests are conducted. These quick tests of functionality do of course not guarantee a completely correct setup, but they are a quick way of finding major flaws. We assess:

- Are all components created?
- Are hostnames and IP addresses (including DHCP) correctly configured?
- Is routing set up correctly?
- Does DNS work as expected?

After the setup finishes, all machines are running and reachable via SSH by their hostnames. Looking at the interface information of each host shows that static configuration for the routers and DHCP for the production hosts is working as intended. Table 7.1 lists all IP addresses. One can confirm that static IP addresses from the XML have been assigned to their respective interfaces, while all other interfaces are given an IP within the range specified by their DHCP server.

Table 7.1: IP addresses assigned to the hosts in the `test`-setup

Hostname	**Interface**	**MAC address**	**IP address/Prefix**
test-router-1	enxdeadbeef0101	de:ad:be:ef:01:01	172.16.244.1/24
test-router-1	enxdeadbeef0102	de:ad:be:ef:01:02	172.16.247.1/24
test-router-2	enxdeadbeef0201	de:ad:be:ef:02:01	172.16.245.1/24
test-router-2	enxdeadbeef0202	de:ad:be:ef:02:02	172.16.247.2/24
test-host-1	enxdeadbeef0301	de:ad:be:ef:03:01	172.16.244.111/24
test-host-2	enxdeadbeef0401	de:ad:be:ef:04:01	172.16.245.162/24
test-host-3	enxdeadbeef0501	de:ad:be:ef:05:01	172.16.244.112/24
test-host-4	enxdeadbeef0601	de:ad:be:ef:06:01	172.16.245.163/24

If routing has been setup correctly, the `test-host-1` should not only be able to reach `test-host-3` on its local network, but also `test-host-2` in the remote network. Figure 7.3 shows the reachability and displays that the correct routers appear as hops via `traceroute`.

```
root@test-host-1:~# ping 172.16.244.112 -c 1
  PING 172.16.244.112 (172.16.244.112) 56(84) bytes of data.
  64 bytes from 172.16.244.112: icmp_seq=1 ttl=64 time=0.494 ms

root@test-host-1:~# ping 172.16.245.162 -c 1
  PING 172.16.245.162 (172.16.245.162) 56(84) bytes of data.
  64 bytes from 172.16.245.162: icmp_seq=1 ttl=62 time=1.64 ms

root@test-host-1:~# traceroute 172.16.245.162
  traceroute to 172.16.245.162 (172.16.245.162), 30 hops max, 60 byte packets
  1  test-router-1.net-1.test.yggdrasil.vitsi.test (172.16.244.1)  0.740 ms  0.606 ms
       0.564 ms
  2  172.16.247.2 (172.16.247.2)  0.993 ms  0.947 ms  0.923 ms
  3  172.16.245.162 (172.16.245.162)  1.306 ms  1.271 ms  1.242 ms
```

Figure 7.3: Reachability local and remote hosts in the feature evaluation.

Figure 7.4 confirms that DNS is also working for both the local network and remote domains, making hosts available by their fully qualified domain name.

```
root@test-host-1:~# ping test-host-3 -c 1
  PING test-host-3.net-1.test.yggdrasil.vitsi.test (172.16.244.112) 56(84) bytes of data
  64 bytes from test-host-3.net-1.test.yggdrasil.vitsi.test (172.16.244.112):
        icmp_seq=1 ttl=64 time=0.456 ms

root@test-host-1:~# ping test-host-2.net-2.test.yggdrasil.vitsi.test -c 1
  PING test-host-2.net-2.test.yggdrasil.vitsi.test (172.16.245.162) 56(84) bytes of data
  64 bytes from 172.16.245.162: icmp_seq=1 ttl=62 time=0.863 ms
```

Figure 7.4: Reachability of hosts via DNS in the feature evaluation.

7.2.4 Reconfiguring the setup

Adjustments to the testbed are made by altering the initial XML to the version shown in Appendix C.3. This represents a new desired goal state of the testbed, visualized in Appendix C.4. Figure 7.5 visualizes the changes that have to be made to the original setup from Figure 7.1 in order to get to the goal state.

In the updated configuration, a new network is created and connected to the transport network via its own new router `test-router-3`. The existing host `test-host-3` is moved to the new network while `test-host-4` is removed entirely. To enable routing between the new and the existing networks, all routing tables are updated with a new rule that directs the correct traffic from and to the network `net-3`. In addition to that, `test-router-1` is configured with the firewall rule shown in Figure 7.6. This rule blocks incoming traffic from the new network to the network named `net-1`. Note that the rule rejects packets with an ICMP response stating that the packet has been *administratively prohibited.* This is deliberately configured for this evaluation to clearly

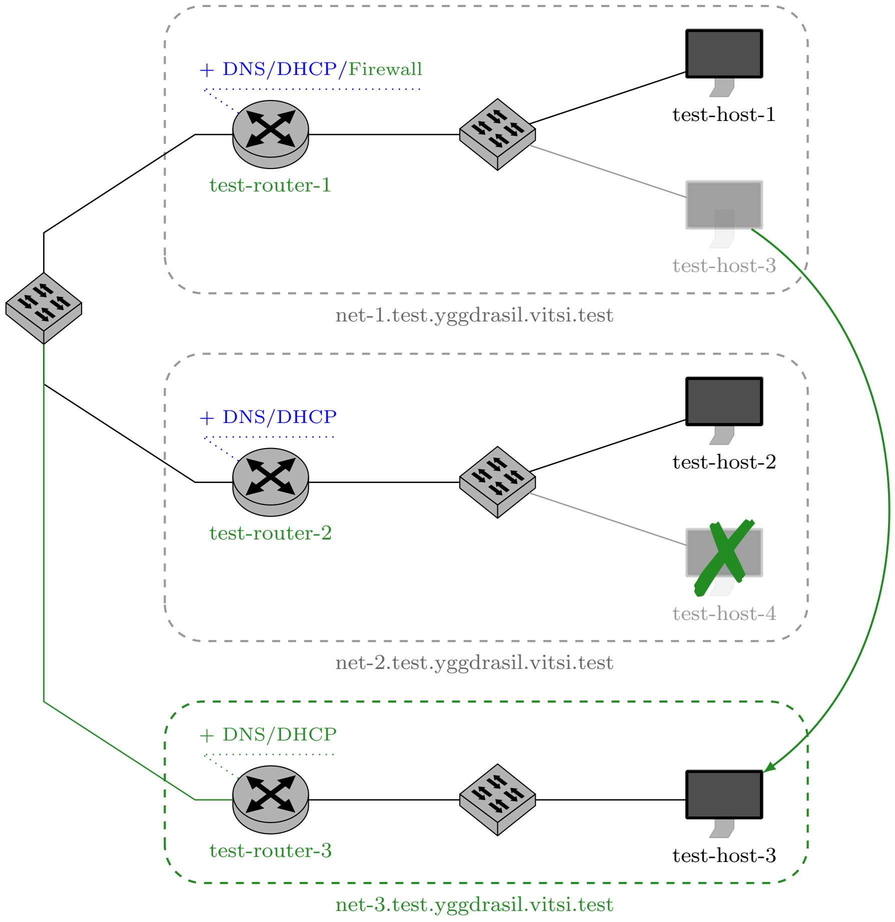


Figure 7.5: Visualization of the updated goal configuration in the feature evaluation.

verify the application of the rule afterwards. By simply dropping packets, one could not distinguish the packet loss from a faulty configuration.

```
iptables -A FORWARD -s 172.16.246.0/24 -j REJECT --reject-with icmp-admin-prohibited
```

Figure 7.6: The *iptables* firewall rule to prohibit traffic from `net-3` to `net-1`.

After uploading the new configuration and triggering the setup, the information collection method is able to find all elements from the initial setup tagged with the configuration's identifier. The change detection mechanism now gives detailed information about changes for the planner to process.

7.2.5 Quantitative analysis of the reconfiguration

The planner identifies a list of 30 steps after 0.050 s (see Appendix C.5). With only one new host and some minor changes, the update plan is considerably shorter than the previous one taking a total time of 11 min 7 s. Major changes are the removal of `test-host-4` in step 10 and step 11, as well as moving `test-host-3` to a new network in step 23. The latter is scheduled after the configuration of its new DHCP server in order to save time during the first reboot. Figure 7.7 once again visualizes the execution of the update plan in terms of the individual time taken by each step.

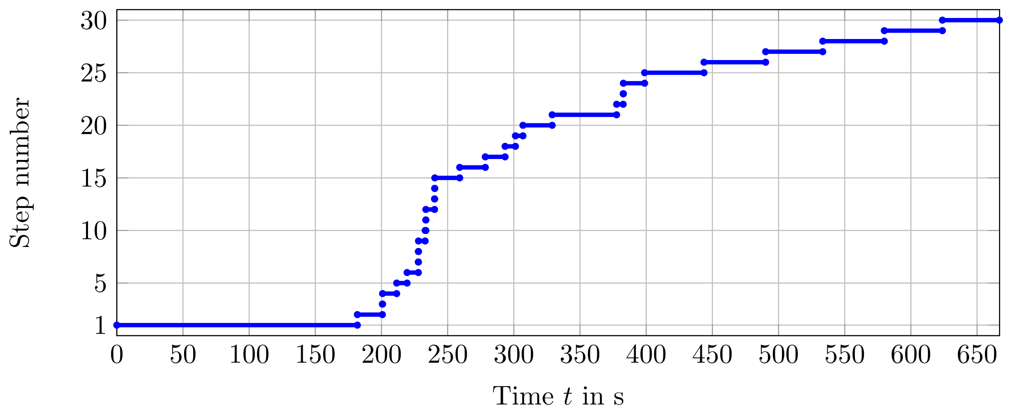


Figure 7.7: Time distribution of the update steps in the feature evaluation.

The creation of the new router `test-router-3` in step 1 consumes by far the most time out of all single steps. In contrast, the removal of `test-host-4` in step 10 is conducted very fast. The larger block of steps that makes up the second half of the update reconfigures the existing DNS interfaces. This is necessary for our enclosed setup to provide the other DNS servers with information about where the new DNS server running on `test-router-3` can be found.

7.2.6 Qualitative evaluation of the reconfiguration

As before, we conduct small tests, especially regarding the new network. A look at the hypervisor's database shows that both `test-host-4` and its associated hard disk have been successfully removed. As Table 7.2 shows, `test-host-3` is now part of the new network and received a new IP address via DHCP.

Table 7.2: IP addresses that changed after the update of the `test`-setup

Hostname	**Interface**	**MAC address**	**IP address/Prefix**
test-router-3	enxdeadbeef0701	de:ad:be:ef:07:01	172.16.246.1/24
test-router-3	enxdeadbeef0702	de:ad:be:ef:07:02	172.16.247.3/24
test-host-3	enxdeadbeef0501	de:ad:be:ef:05:01	172.16.246.112/24

By conducting the same simple tests via ICMP echo requests as in the initial setup, we can assess that the host is reachable from both its own network as well as remote ones (see Figure 7.8).

```
root@test-router-3:~# ping test-host-3 -c 1
  PING test-host-3 (172.16.246.112) 56(84) bytes of data
  64 bytes from test-host-3.net-3.test.yggdrasil.vitsi.test (172.16.246.112):
        icmp_seq=1 ttl=64 time=0.450 ms

root@test-host-2:~# ping test-host-3.net-3.test.yggdrasil.vitsi.test -c 1
  PING test-host-3.net-3.test.yggdrasil.vitsi.test (172.16.246.112) 56(84) bytes of data
  64 bytes from 172.16.246.112: icmp_seq=1 ttl=62 time=0.934 ms
```

Figure 7.8: Reachability of `test-host-3` after the update in the feature evaluation.

The firewall rule should prohibit traffic coming from `test-host-3` to `test-host-1`. The echo requests depicted in Figure 7.9 show that these packets are being filtered correctly.

```
root@test-host-3:~# ping test-host-1.net-1.test.yggdrasil.vitsi.test -c 3
  PING test-host-1.net-1.test.yggdrasil.vitsi.test (172.16.244.111) 56(84) bytes of data
  From 172.16.247.1 icmp_seq=1 Packet filtered
  From 172.16.247.1 icmp_seq=2 Packet filtered
  From 172.16.247.1 icmp_seq=3 Packet filtered
```

Figure 7.9: Application of the new firewall rule in the feature evaluation.

7.3 Case study: Mirroring an existing setup

The case study is based on the result of the information collection process running inside a physical network. The network used is a setup adapted from the lab course *iLab* offered by the Chair of Network Architectures and Services. The goal for this case study is to mirror a physical network setup into a virtual testbed.

Figure 7.10 displays the physical setup that is examined by the information collection process. In addition to what is shown in the figure, the hosts (PC1–PC4) and the Linux-Router are connected to the lab's internal management network. This network is also detected by the information collection process and is part of the testbed setup.

In contrast to the manually written XML files of the feature evaluation, the physical environment in which the information is gathered, results in some missing attributes. The location of all components has to be changed to the desired hypervisor server, in our case the Xen server *yggdrasil*. Hardware information like the CPU count or memory is not gathered as there is currently no scanner available. The preprocessor applies default values of 2 cores and 2 GB memory. This is done to reduce the resource consumption of the hypervisor server as the real machines in the scanned environment have quad-core workstation CPUs and 16 GB of memory each. The hostnames in our scenario are given as IP addresses which cannot be used for configuration via SSH. To resolve this issue, the preprocessor translates the addresses into a usable hostname by replacing the dots.

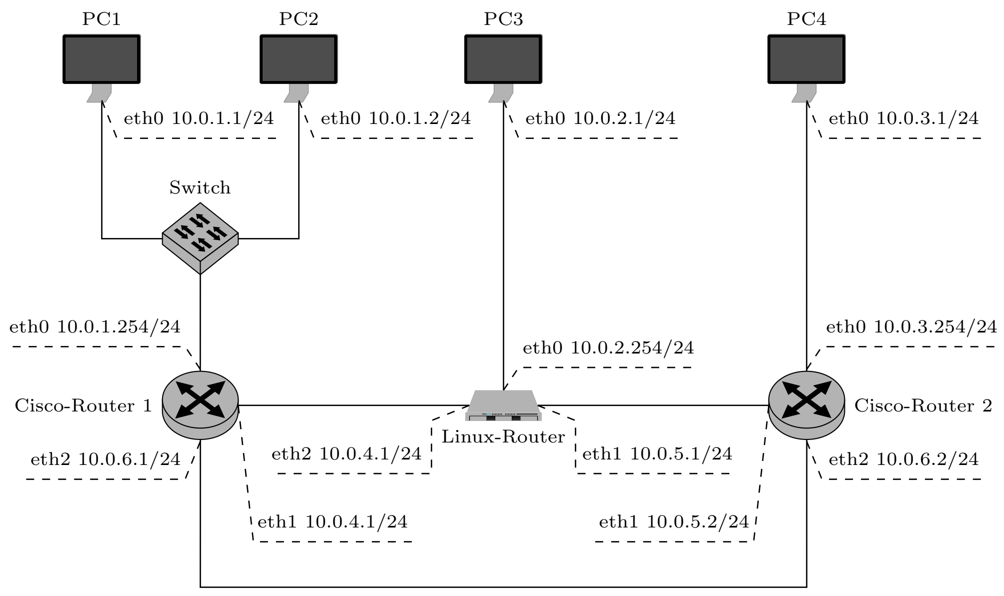


Figure 7.10: Visualization of the physical network being examined in the case study.

The completed information collection process exports its state to an XML in the known format (see Appendix D.1). What is noticeable about this XML is that interfaces that are not connected to any network but are physically present in the host, are listed as well. These interfaces are ignored by the setup process as they are not part of a network which is currently necessary for planning. The gathered information does also include the management network of the lab setup as 192.168.38.0/24.

With this XML, we can once again create a plan (see Appendix D.2) with 92 steps in 0.252 s. Figure 7.11 visualizes the setup and configuration process in regard to its execution time. The complete time for setup and configuration is 42 min 32 s. This includes two extraordinarily long reboot operations due to Ubuntu's issues which are not inherent part of the automated setup and configuration process. Like in the feature evaluation we can observe steps that complete almost instantaneously, in this case the steps 10–43. These steps create virtual networks and interfaces on the Xen server and do not have to copy large data like cloning from a template does.

Afterwards, we connect to each of the machines to make sure the assigned IP addresses match with the respective addresses of the original setup in Figure 7.10. By using `traceroute` we assess that the routing works as intended. In order to test all available routes, we trace packets from `PC1 (10.0.1.1)` to `PC4 (10.0.3.1)` which is supposed to use the direct link between the two Cisco routers. In addition to that, we trace packets from `PC3 (10.0.2.1)` to both `PC2 (10.0.1.2)` and `PC4 (10.0.3.1)` which are supposed to be sent over the two transport networks between the central Linux router. The results are shown in Figure 7.12, displaying that all routes are used as expected.

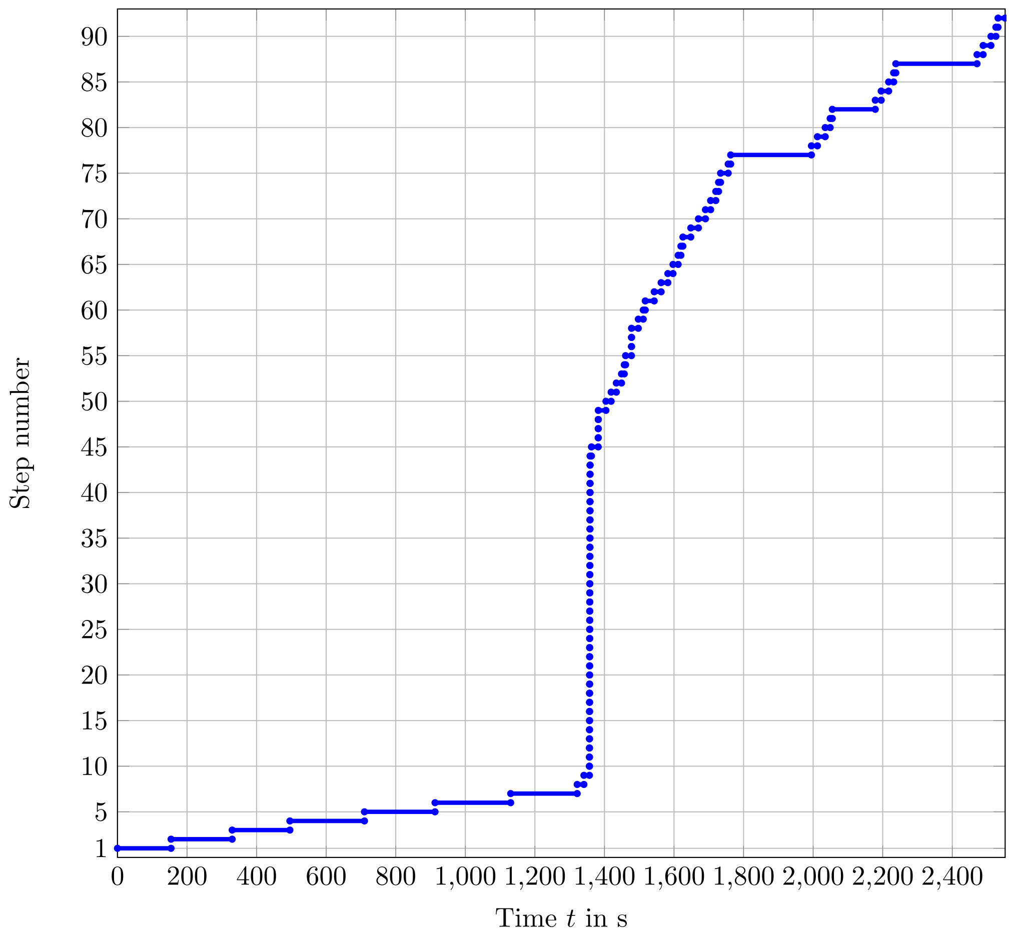


Figure 7.11: Time distribution of setup steps in the case study.

```
root@10-0-1-1:~# traceroute 10.0.3.1
  traceroute to 10.0.3.1 (10.0.3.1), 30 hops max, 60 byte packets
  1  10.0.1.254 (10.0.1.254)  0.808 ms  0.720 ms  0.687 ms
  2  10.0.6.2 (10.0.6.2)  1.070 ms  1.044 ms  1.013 ms
  3  10.0.3.1 (10.0.3.1)  1.000 ms  0.970 ms  0.939 ms

root@10-0-2-1:~# traceroute 10.0.1.2
  traceroute to 10.0.1.2 (10.0.1.2), 30 hops max, 60 byte packets
  1  10.0.2.254 (10.0.2.254)  0.737 ms  0.711 ms  0.697 ms
  2  10.0.4.1 (10.0.4.1)  1.267 ms  1.243 ms  1.196 ms
  3  10.0.1.2 (10.0.1.2)  1.499 ms  1.470 ms  1.432 ms

root@10-0-2-1:~# traceroute 10.0.3.1
  traceroute to 10.0.3.1 (10.0.3.1), 30 hops max, 60 byte packets
  1  10.0.2.254 (10.0.2.254)  0.501 ms  0.458 ms  0.399 ms
  2  10.0.5.2 (10.0.5.2)  1.518 ms  1.474 ms  1.446 ms
  3  10.0.3.1 (10.0.3.1)  1.802 ms  1.774 ms  1.735 ms
```

Figure 7.12: Traceroute results displaying routing in the case study.

Chapter 8

Conclusion

The thesis proposes a process for automating the setup and configuration of testbeds. The solution is subject to nine requirements initially defined in Section 3.3 based on needs of scientific testbed environments. The assessment of related work on the topic of automated testbed setups in Chapter 4 shows that there is no sufficient solution that fulfills all requirements. Therefore, we propose the design of a possible solution that relies on automated planning and scheduling as a key element. The practical applicability of the process is exhibited by an implementation of the process which is used to conduct a case study. To conclude the thesis, we refer back to the initial research questions and summarize the answers given by the thesis. As the field of automated setup and configuration of testbeds provides many additional interesting subtopics that expand on the work of this thesis, we give some outlook on possible future work.

8.1 Answering research questions

In Section 1.2 we introduced the central research question of how testbeds can be configured automatically. Subsequently, we summarize the answers to the sub questions arising from this central demand.

Q1 Which information is necessary in order to setup and configure a testbed up to a reasonable level of complexity and how should this information be structured? As an answer to this question, we proposed an object-oriented data model in Section 5.4. The data model provides classes and attributes for fundamental network components which cover aspects up until ISO/OSI layer 4 and basic network services like DNS and DHCP. Layer 7 services are covered in a generic way by modeling them as software products listening on a specified port. This model is sufficient for describing the configuration of a usable testbed with basic network services. Due to its object-oriented nature, the data model is easily extensible for additional elements.

Q2 What are the individual steps during the configuration of the network components? We found that the setup steps identified (see Table 5.1) for the automated setup and configuration process are strongly connected to the objects of the data model and their attributes. Setup steps are grouped in four different groups. Steps for *creating and removing elements* and steps *configuring components* create objects and deploy their attribute values into the real testbed. Additional steps regarding *power management* are necessary due to the domain revolving around computing devices. The group of steps for *managing configuration references* are a result of the requirements **R7 (Multi-user support)** and **R8 (Sharing components)**.

Q3 How to determine changes between testbed configurations in order to generate update plans? For this issue, we introduced two mechanisms, the *change detection* and the *problem parser*, both shown in Figure 5.2. Changes are determined by generic comparison over the object-oriented hierarchy. With two given testbed states in form of instance hierarchies, corresponding elements are matched by their identifier and compared regarding each of their attributes, recursively. Changes are noted in terms of whether they are new, removed or altered.

Q4 What is a generic approach to determine an execution plan consisting of the aforementioned steps? Determining the necessary steps based on the detected changes and arranging them into an execution plan is solved by using a PDDL planner. To do so, the output of the change detection is processed by the *problem parser* which generates a formal problem file. In this file, predicates are applied to objects based on how they have changed between testbed states using the *Planning Domain Definition Language (PDDL)*. The problem is solved by a PDDL planner. Additional background on planning is given in Section 3.4.3. Field-tested and sophisticated planner software is available as open-source solution and allows for a generic way of finding solutions to all update problems in a testbed.

8.2 Future work

The automated setup and configuration process proposed in this thesis and its open-source implementation can be further improved and extended with additional features.

Additional builder modules Due to the modular design of the building modules that conduct the setup and configuration, the implementation of additional modules is always possible and desirable in order to support more platforms. We further encourage the continuation by providing a public repository [41]. Further development efforts can be focused on adding support for additional hypervisors on top of Xen. In terms of operating systems, many currently implemented builders for configuration are Debian-based which leaves room for improvement. The possibilities for building modules for concrete services are practically unlimited.

Improved multi-user support One can expand on the requirements **R7** and **R8** by introducing rights management to prevent unwanted changes to machines in a large-scale testbed with many more users.

Time bottleneck As seen in Chapter 7, the building phase can take a considerable amount of time if setups get larger. Using concurrency could improve the performance. This is currently not implemented due to two reasons. The hypervisor infrastructure (Xen with the XAPI toolstack) does not support concurrent commands, especially not cloning a template which takes by far the most time. In addition to that, concurrency is not part of classical planning (see Section 3.4.3) and while there are extensions to PDDL to allow concurrent actions, finding a usable planner in the non-standardized environment of PDDL extensions remains an issue. Improving the performance of cloning templates without concurrency would also resolve the current time-bottleneck.

Template changes The steps determined in Section 5.6 are complete for what the process currently covers. In the special case of a host having its template changed, one could introduce new steps that are capable of transitioning from one template to another without reinstallation. This is theoretically possible if only certain template properties like a single installed software is missing.

Graphical client A larger addition to *INSALATA* would be a graphical user interface on the client side. Currently, only a terminal client that can execute commands declared by the service (see Section 6.1) is provided. While it is certainly possible to create very sophisticated terminal clients, further efforts should focus on a graphical frontend which can provide a richer user experience. With a graphical interface, editing of XML files already uploaded to the system can be achieved in an in-application editor. Viewing setups is a lot more convenient and can be extended to live network plans. In addition to that, changes conducted by the preprocessor after uploading an XML can be shown interactively and incorporate user input.

Appendix A

XML Schema

```xml
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="layer2network">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="id" use="required"/>
          <xs:attribute type="xs:string" name="location" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="layer3network">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="id" use="required"/>
          <xs:attribute type="xs:string" name="address" use="required"/>
          <xs:attribute type="xs:string" name="netmask" use="required"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="dhcp">
    <xs:complexType>
      <xs:simpleContent>
        <xs:extension base="xs:string">
          <xs:attribute type="xs:string" name="from" use="optional"/>
          <xs:attribute type="xs:string" name="to" use="optional"/>
          <xs:attribute type="xs:string" name="lease" use="optional"/>
          <xs:attribute type="xs:string" name="announcedGateway" use="optional"/>
        </xs:extension>
      </xs:simpleContent>
    </xs:complexType>
  </xs:element>
```

```xml
<xs:element name="dns">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="domain" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="services">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="dhcp" minOccurs="0"/>
      <xs:element ref="dns"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="layer3address">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="services"/>
    </xs:sequence>
    <xs:attribute type="xs:string" name="address" use="required"/>
    <xs:attribute type="xs:string" name="network" use="required"/>
  </xs:complexType>
</xs:element>
<xs:element name="interface">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="layer3address" minOccurs="0"/>
    </xs:sequence>
    <xs:attribute type="xs:int" name="rate" use="optional"/>
    <xs:attribute type="xs:short" name="mtu" use="optional"/>
    <xs:attribute type="xs:string" name="mac" use="required"/>
    <xs:attribute type="xs:string" name="network" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="route">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="destination" use="optional"/>
        <xs:attribute type="xs:string" name="gateway" use="optional"/>
        <xs:attribute type="xs:string" name="genmask" use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="rule">
  <xs:complexType>
    <xs:simpleContent>
```

```
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="chain"/>
        <xs:attribute type="xs:string" name="action"/>
        <xs:attribute type="xs:string" name="srcnet"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="firewallRules">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rule"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="raw">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="firewall"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="disk">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute type="xs:string" name="id" use="required"/>
        <xs:attribute type="xs:int" name="size" use="optional"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
<xs:element name="interfaces">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="interface" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="routes">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="route" maxOccurs="unbounded" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="firewall">
  <xs:complexType>
    <xs:sequence>
```

```
        <xs:element ref="firewallRules" maxOccurs="unbounded" minOccurs="0"/>
        <xs:element ref="raw" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="disks">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="disk"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="host">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="interfaces"/>
        <xs:element ref="routes" minOccurs="0"/>
        <xs:element ref="firewall" minOccurs="0"/>
        <xs:element ref="disks"/>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="cpus" use="optional"/>
      <xs:attribute type="xs:string" name="id" use="optional"/>
      <xs:attribute type="xs:long" name="memoryMax" use="optional"/>
      <xs:attribute type="xs:int" name="memoryMin" use="optional"/>
      <xs:attribute type="xs:string" name="powerState" use="optional"/>
      <xs:attribute type="xs:string" name="location" use="optional"/>
      <xs:attribute type="xs:string" name="template" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="layer2networks">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="layer2network" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="layer3networks">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="layer3network" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="hosts">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="host" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="config">
```

```
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="layer2networks"/>
        <xs:element ref="layer3networks"/>
        <xs:element ref="hosts"/>
      </xs:sequence>
      <xs:attribute type="xs:string" name="name"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix B

Testbed PDDL Domain

```
(define (domain testbed)
  (:requirements :strips :equality :typing :conditional-effects :action-costs)
  (:types router - host plain - host host network interface dhcp - service dns -
        service service disk)
  (:predicates (named ?x - host)
              (nameNotApplied ?x)
              (created ?x)
              (old ?x)
              (new ?x)

              (running ?x - host)
              (cpusConfigured ?x - host)
              (memoryConfigured ?x - host)
              (templateChanged ?x - host)
              (dnsConfigured ?x - dns)
              (dhcpConfigured ?x - dhcp)
              (serviceConfigured ?x - service)
              (routingConfigured ?x - router)
              (firewallConfigured ?x - host)
              (networkConfigured ?x - interface)
              (mtuConfigured ?x - interface)
              (rateConfigured ?x - interface)
              (interfaceConfigured ?x - interface)
              (static ?x - interface)

              (attached ?d - disk ?h - host)
              (part-of ?x ?y)
              (configNameAdded ?x)
  )

  (:action createNetwork
  :parameters (?n - network)
  :precondition (not (created ?n))
  :effect (and (created ?n)))

```

```
(:action boot
:parameters (?x - host)
:precondition (and (created ?x) (not (running ?x)) (named ?x)
       (not (nameNotApplied ?x)))
:effect (and (running ?x)))

(:action bootAndNamed
:parameters (?x - host)
:precondition (and (created ?x) (not (running ?x)) (named ?x) (nameNotApplied ?x))
:effect (and (running ?x) (not (nameNotApplied ?x))))

(:action bootUnnamed
:parameters (?x - plain)
:precondition (and (created ?x) (not (running ?x))
  (forall (?p - plain)
    (imply (or (not (named ?p)) (nameNotApplied ?p)) (not (running ?p)))))
:effect (and (running ?x)))

(:action bootUnnamed
:parameters (?x - router)
:precondition (and (created ?x) (not (running ?x))
  (forall (?r - router)
    (imply (or (not (named ?r)) (nameNotApplied ?r)) (not (running ?r)))))
:effect (and (running ?x)))

(:action shutdown
:parameters (?x - host)
:precondition (running ?x)
:effect (and (not (running ?x))))

(:action reboot
:parameters (?x - host)
:precondition (running ?x)
:effect (and (running ?x)))

(:action rebootAndNamed
:parameters (?x - host)
:precondition (and (running ?x) (nameNotApplied ?x))
:effect (and (running ?x) (not (nameNotApplied ?x))))

(:action createHost
:parameters (?x - host)
:precondition (and (not (created ?x)))
:effect (and (created ?x) (cpusConfigured ?x) (memoryConfigured ?x)))

(:action createInterface
:parameters (?x - interface)
:precondition (and (not (created ?x))
  (forall (?h - host) (imply (part-of ?x ?h) (and (created ?h) (not (running ?h)))))
  (forall (?n - network) (imply (part-of ?x ?n) (created ?n))))
:effect (and (created ?x) (networkConfigured ?x) (mtuConfigured ?x)
       (rateConfigured ?x)))
```

```
(:action deleteHost
:parameters (?h - host)
:precondition (and (created ?h) (not (running ?h)))
:effect (and (not (created ?h)) (not (named ?h)) (not (templateChanged ?h))
  (not (configNameAdded ?h))
  (forall (?i - interface)
    (and
      (when (part-of ?i ?h) (and (not (created ?i)) (not (interfaceConfigured ?i))))
      (forall (?s - service)
        (when (part-of ?s ?i)
          (and (not (created ?s)) (not (dnsConfigured ?s)) (not (dhcpConfigured ?s))
           (not (serviceConfigured ?s)))
        )
      )
    )
  )
))

(:action removeNetwork
:parameters (?n - network)
:precondition (and (old ?n)
  (forall (?i - interface) (imply (part-of ?i ?n) (not (created ?i)))))
:effect (and (not (created ?n))))

(:action removeHost
:parameters (?h - host)
:precondition (and (old ?h) (not (running ?h)))
:effect (and (not (created ?h))
  (forall (?i - interface) (when (part-of ?i ?h) (not (created ?i))))))

(:action addDisk
:parameters (?d - disk ?h - host)
:precondition (and (created ?h) (not (attached ?d ?h)) (part-of ?d ?h))
:effect (and (attached ?d ?h)))

(:action removeDisk
:parameters (?d - disk ?h - host)
:precondition (and (old ?d) (attached ?d ?h))
:effect (and (not (attached ?d ?h))))

(:action removeInterface
:parameters (?i - interface)
:precondition (and (old ?i)
  (forall (?h - host) (imply (part-of ?i ?h) (not (running ?h)))))
:effect (and (not (created ?i)) (not (mtuConfigured ?i)) (not (rateConfigured ?i))
      (not (networkConfigured ?i))))

(:action name
:parameters (?x - host)
:precondition (running ?x)
:effect (and (named ?x) (nameNotApplied ?x)))
```

```

(:action configureRouting
:parameters (?r - router)
:precondition (and (running ?r)
  (forall (?i - interface) (imply (part-of ?i ?r) (interfaceConfigured ?i))))
:effect (and (routingConfigured ?r)))

(:action configureFirewall
:parameters (?h - host)
:precondition (and (running ?h)
  (forall (?i - interface) (imply (part-of ?i ?h) (interfaceConfigured ?i))))
:effect (and (firewallConfigured ?h)))

(:action configureCpus
:parameters (?x - host)
:precondition (and (created ?x) (not (running ?x)))
:effect (and (cpusConfigured ?x)))

(:action configureMemory
:parameters (?x - host)
:precondition (and (created ?x) (not (running ?x)))
:effect (and (memoryConfigured ?x)))

(:action configureInterface
:parameters (?x - interface)
:precondition (and
  (created ?x) (networkConfigured ?x) (rateConfigured ?x) (mtuConfigured ?x)
  (forall (?h - host) (imply (part-of ?x ?h) (running ?h)))
  (forall (?d - dns) (dnsConfigured ?d))
  (forall (?d - dhcp) (dhcpConfigured ?d))
)
:effect (and (interfaceConfigured ?x)))

(:action configureInterface
:parameters (?x - interface)
:precondition (and (static ?x)
  (created ?x) (networkConfigured ?x) (rateConfigured ?x) (mtuConfigured ?x)
  (forall (?h - host) (imply (part-of ?x ?h) (running ?h)))
)
:effect (and (interfaceConfigured ?x)))

(:action unconfigureInterface
:parameters (?x - interface)
:precondition (and
  (not (created ?x)) (interfaceConfigured ?x) (old ?x)
  (forall (?h - host) (imply (part-of ?x ?h) (running ?h)))
)
:effect (and (not (interfaceConfigured ?x))
  (forall (?h - host) (when (part-of ?i ?h) (not (part-of ?i ?h))))
))

```

```
(:action configureDns
:parameters (?s - dns)
:precondition (and (not (dnsConfigured ?s))
  (forall (?i - interface)
    (imply (part-of ?s ?i)
      (and (created ?i) (interfaceConfigured ?i)
        (forall (?h - host)
          (imply (part-of ?i ?h) (and (created ?h) (named ?h) (running ?h)))
        )
      )
    )
  )
)
:effect (and (created ?s) (not (new ?s)) (dnsConfigured ?s) (serviceConfigured ?s)
  (forall (?d - dns)
    (when (new ?s) (and (not (dnsConfigured ?d)) (not (new ?d))))
  )
))

(:action unconfigureDns
:parameters (?s - dns)
:precondition (old ?s)
:effect (and (not (created ?s)) (not (dnsConfigured ?s))
       (not (serviceConfigured ?s))))

(:action configureDhcp
:parameters (?s - dhcp)
:precondition (and (not (dhcpConfigured ?s))
  (forall (?i - interface)
    (imply (part-of ?s ?i)
      (and (created ?i) (interfaceConfigured ?i)
        (forall (?h - host)
          (imply (part-of ?i ?h) (and (created ?h) (named ?h) (running ?h)))
        )
      )
    )
  )
)
:effect (and (created ?s) (dhcpConfigured ?s) (serviceConfigured ?s)))

(:action unconfigureDhcp
:parameters (?s - dhcp)
:precondition (old ?s)
:effect (and (not (created ?s)) (not (dnsConfigured ?s))
       (not (serviceConfigured ?s))))

(:action configureNetwork
:parameters (?x - interface)
:precondition (and (created ?x) (forall (?h - host) (imply (part-of ?x ?h)
       (not (running ?h)))) (forall (?d - dhcp) (dhcpConfigured ?d)))
:effect (and (networkConfigured ?x)))

```

```
  (:action configureMtu
  :parameters (?x - interface)
  :precondition (created ?x)
  :effect (and (mtuConfigured ?x)))

  (:action configureRate
  :parameters (?x - interface)
  :precondition (created ?x)
  :effect (and (rateConfigured ?x)))

  (:action addConfigNameNetwork
  :parameters (?x - network)
  :precondition (and (created ?x) (not (old ?x)))
  :effect (and (configNameAdded ?x)))

  (:action addConfigNameHost
  :parameters (?x - host)
  :precondition (and (created ?x) (not (old ?x)))
  :effect (and (configNameAdded ?x)))

  (:action addConfigNameDisk
  :parameters (?x - disk)
  :precondition (and (created ?x) (not (old ?x)))
  :effect (and (configNameAdded ?x)))
)
```

Appendix C

Feature evaluation

C.1 XML of the students desired testbed configuration

```xml
<?xml version="1.0"?>
<config name="test">
  <layer2networks>
    <layer2network id="test-net-1" location="yggdrasil"/>
    <layer2network id="test-net-2" location="yggdrasil"/>
    <layer2network id="test-transport" location="yggdrasil"/>
  </layer2networks>
  <layer3networks>
    <layer3network id="test-net-1" address="172.16.244.0" netmask="255.255.255.0"/>
    <layer3network id="test-net-2" address="172.16.245.0" netmask="255.255.255.0"/>
    <layer3network id="test-transport" address="172.16.247.0" netmask="255.255.255.0"/>
  </layer3networks>
  <hosts>
  <host cpus="2" id="test-router-1" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="router-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:01:01"
        network="test-net-1">
          <layer3address address="172.16.244.1" network="test-net-1">
            <services>
              <dhcp from="172.16.244.100" to="172.16.244.200" lease="8h"
        announcedGateway="172.16.244.1"/>
              <dns domain="net-1.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:01:02"
        network="test-transport">
          <layer3address address="172.16.247.1" network="test-transport">
            <services>
              <dns domain="net-1.test.yggdrasil.vitsi.test"/>
            </services>
        </layer3address>
      </interface>
    </interfaces>
    <routes>
      <route destination="172.16.245.0" gateway="172.16.247.2"
      genmask="255.255.255.0" />
    </routes>
    <firewall />
    <disks />
  </host>
  <host cpus="2" id="test-router-2" memoryMax="2G" memoryMin="512M"
      powerState="Running" location="yggdrasil" template="router-base">
    <interfaces>
      <interface rate="125000" mtu="1500" mac="de:ad:be:ef:02:01"
      network="test-net-2">
        <layer3address address="172.16.245.1" network="test-net-2">
          <services>
            <dhcp from="172.16.245.100" to="172.16.245.200" lease="8h"
      announcedGateway="172.16.245.1"/>
            <dns domain="net-2.test.yggdrasil.vitsi.test"/>
          </services>
        </layer3address>
      </interface>
      <interface rate="125000" mtu="1500" mac="de:ad:be:ef:02:02"
      network="test-transport">
        <layer3address address="172.16.247.2" network="test-transport">
          <services>
            <dns domain="net-2.test.yggdrasil.vitsi.test"/>
          </services>
        </layer3address>
      </interface>
    </interfaces>
    <routes>
      <route destination="172.16.244.0" gateway="172.16.247.1"
      genmask="255.255.255.0" />
    </routes>
    <firewall />
    <disks />
  </host>
  <host cpus="2" id="test-host-1" memoryMax="2G" memoryMin="512M"
      powerState="Running" location="yggdrasil" template="host-base">
    <interfaces>
      <interface rate="125000" mtu="1500" mac="de:ad:be:ef:03:01"
      network="test-net-1"/>
    </interfaces>
    <disks />
  </host>
  <host cpus="2" id="test-host-2" memoryMax="2G" memoryMin="512M"
      powerState="Running" location="yggdrasil" template="host-base">
    <interfaces>
```

```
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:04:01"
         network="test-net-2"/>
      </interfaces>
      <disks />
    </host>
    <host cpus="2" id="test-host-3" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:05:01"
         network="test-net-1"/>
      </interfaces>
      <disks />
    </host>
    <host cpus="2" id="test-host-4" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:06:01"
         network="test-net-2"/>
      </interfaces>
      <disks />
    </host>
  </hosts>
</config>
```

C.2 Plan for setting up the initial setup

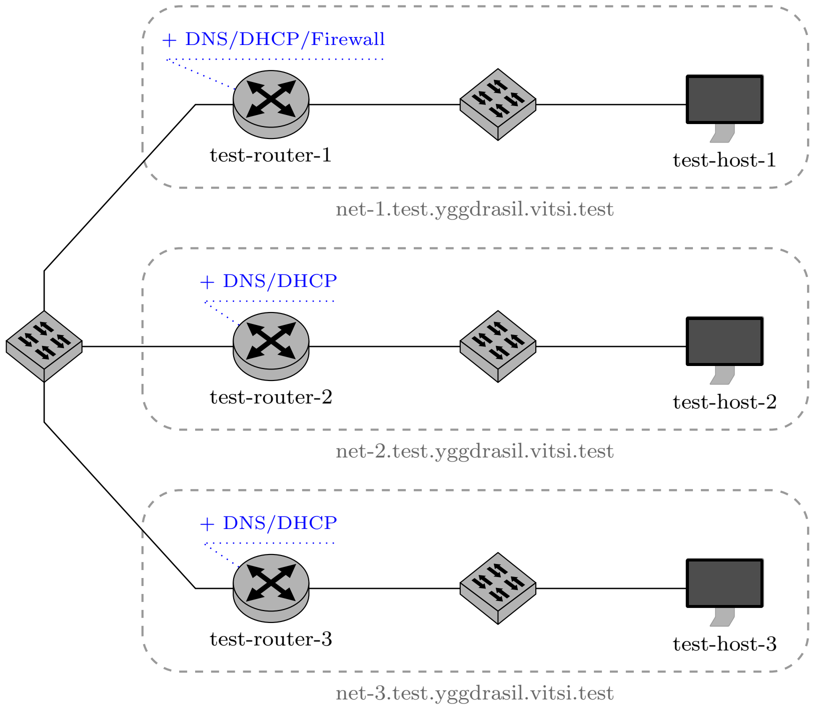
```
(createhost test-router-1)
(createhost test-router-2)
(createhost test-host-1)
(createhost test-host-2)
(createhost test-host-3)
(createhost test-host-4)
(bootunnamed test-router-1)
(bootunnamed test-host-1)
(createnetwork test-transport)
(addconfignamenetwork test-transport)
(createinterface enxdeadbeef0202)
(addconfignamehost test-router-1)
(addconfignamehost test-router-2)
(createnetwork test-net-2)
(createinterface enxdeadbeef0201)
(addconfignamenetwork test-net-2)
(createinterface enxdeadbeef0601)
(createinterface enxdeadbeef0401)
(createnetwork test-net-1)
(addconfignamenetwork test-net-1)
(createinterface enxdeadbeef0501)
(addconfignamehost test-host-1)
(addconfignamehost test-host-2)
(addconfignamehost test-host-3)
(addconfignamehost test-host-4)
(shutdown test-router-1)
(bootunnamed test-router-2)
(createinterface enxdeadbeef0102)
(configureinterface enxdeadbeef0202)
(configureinterface enxdeadbeef0201)
(configurerouting test-router-2)
(createinterface enxdeadbeef0101)
(shutdown test-host-1)
(bootunnamed test-host-2)
(createinterface enxdeadbeef0301)
(shutdown test-router-2)
(bootunnamed test-router-1)
(configureinterface enxdeadbeef0102)
(configureinterface enxdeadbeef0101)
(configurerouting test-router-1)
(name test-router-1)
(rebootandnamed test-router-1)
(bootunnamed test-router-2)
(configuredhcp 172.16.244.1:67_udp_dhcp)
(configuredns 172.16.244.1:53_udp_dns)
(configuredns 172.16.247.1:53_udp_dns)
```

```
(name test-router-2)
(rebootandnamed test-router-2)
(configuredns 172.16.245.1:53_udp_dns)
(configuredhcp 172.16.245.1:67_udp_dhcp)
(configuredns 172.16.247.2:53_udp_dns)
(configureinterface enxdeadbeef0401)
(name test-host-2)
(rebootandnamed test-host-2)
(bootunnamed test-host-1)
(configureinterface enxdeadbeef0301)
(name test-host-1)
(rebootandnamed test-host-1)
(bootunnamed test-host-3)
(configureinterface enxdeadbeef0501)
(name test-host-3)
(rebootandnamed test-host-3)
(bootunnamed test-host-4)
(configureinterface enxdeadbeef0601)
(name test-host-4)
(rebootandnamed test-host-4)
```

C.3 XML of the updated testbed

```xml
<?xml version="1.0"?>
<config name="test">
  <layer2networks>
    <layer2network id="test-net-1" location="yggdrasil" />
    <layer2network id="test-net-2" location="yggdrasil" />
    <layer2network id="test-net-3" location="yggdrasil" />
    <layer2network id="test-transport" location="yggdrasil"/>
  </layer2networks>
  <layer3networks>
    <layer3network id="test-net-1" address="172.16.244.0" netmask="255.255.255.0"/>
    <layer3network id="test-net-2" address="172.16.245.0" netmask="255.255.255.0"/>
    <layer3network id="test-net-2" address="172.16.246.0" netmask="255.255.255.0"/>
    <layer3network id="test-transport" address="172.16.247.0" netmask="255.255.255.0"/>
  </layer3networks>
  <hosts>
  <host cpus="2" id="test-router-1" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="router-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:01:01"
        network="test-net-1">
          <layer3address address="172.16.244.1" network="test-net-1">
            <services>
              <dhcp from="172.16.244.100" to="172.16.244.200" lease="8h"
        announcedGateway="172.16.244.1"/>
              <dns domain="net-1.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:01:02"
        network="test-transport">
          <layer3address address="172.16.247.1" network="test-transport">
            <services>
              <dns domain="net-1.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
      </interfaces>
      <routes>
        <route destination="172.16.245.0" gateway="172.16.247.2"
        genmask="255.255.255.0" />
        <route destination="172.16.246.0" gateway="172.16.247.3"
        genmask="255.255.255.0" />
      </routes>
      <firewall />
      <disks />
    </host>
    <host cpus="2" id="test-router-2" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="router-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:02:01"
        network="test-net-2">
          <layer3address address="172.16.245.1" network="test-net-2">
            <services>
              <dhcp from="172.16.245.100" to="172.16.245.200" lease="8h"
        announcedGateway="172.16.245.1"/>
              <dns domain="net-2.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:02:02"
        network="test-transport">
          <layer3address address="172.16.247.2" network="test-transport">
            <services>
              <dns domain="net-2.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
      </interfaces>
      <routes>
        <route destination="172.16.244.0" gateway="172.16.247.1"
        genmask="255.255.255.0" />
        <route destination="172.16.246.0" gateway="172.16.247.3"
        genmask="255.255.255.0" />
      </routes>
      <firewall />
      <disks />
    </host>
    <host cpus="2" id="test-host-1" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:03:01"
        network="test-net-1"/>
      </interfaces>
      <disks />
    </host>
    <host cpus="2" id="test-router-3" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="router-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:07:01"
        network="test-net-3">
          <layer3address address="172.16.246.1" network="test-net-3">
            <services>
              <dhcp from="172.16.246.100" to="172.16.246.200" lease="8h"
        announcedGateway="172.16.246.1"/>
              <dns domain="net-3.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:07:02"
        network="test-transport">
          <layer3address address="172.16.247.3" network="test-transport">
            <services>
              <dns domain="net-3.test.yggdrasil.vitsi.test"/>
            </services>
          </layer3address>
        </interface>
      </interfaces>
      <routes>
        <route destination="172.16.244.0" gateway="172.16.247.1"
        genmask="255.255.255.0" />
        <route destination="172.16.245.0" gateway="172.16.247.2"
        genmask="255.255.255.0" />
      </routes>
      <firewall />
      <disks />
    </host>
    <host cpus="2" id="test-host-1" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:03:01"
        network="test-net-1"/>
      </interfaces>
      <disks />
    </host>
    <host cpus="2" id="test-host-2" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:04:01"
        network="test-net-2"/>
      </interfaces>
      <disks />
    </host>
    <host cpus="2" id="test-host-3" memoryMax="2G" memoryMin="512M"
        powerState="Running" location="yggdrasil" template="host-base">
      <interfaces>
        <interface rate="125000" mtu="1500" mac="de:ad:be:ef:05:01"
        network="test-net-3"/>
      </interfaces>
      <disks />
    </host>
  </hosts>
</config>
```

C.4 Visualization of the updated testbed



C.5 Plan for updating the setup

```
(createhost test-router-3)
(bootunnamed test-router-3)
(addconfignamehost test-router-3)
(configurefirewall test-router-1)
(configurerouting test-router-1)
(configurerouting test-router-2)
(createnetwork test-net-3)
(addconfignamenetwork test-net-3)
(shutdown test-host-4)
(removehost test-host-4)
(removedisk test-host-4-hdd test-host-4)
(shutdown test-router-3)
(createinterface enxdeadbeef0702)
(createinterface enxdeadbeef0701)
(bootunnamed test-router-3)
(configureinterface enxdeadbeef0702)
(configureinterface enxdeadbeef0701)
(configurerouting test-router-3)
(name test-router-3)
(rebootandnamed test-router-3)
(configuredhcp 172.16.246.1:67_udp_dhcp)
(shutdown test-host-3)
(configurenetwork enxdeadbeef0501)
(boot test-host-3)
```

```
(configuredns 172.16.246.1:53_udp_dns)
(configuredns 172.16.244.1:53_udp_dns)
(configuredns 172.16.247.1:53_udp_dns)
(configuredns 172.16.245.1:53_udp_dns)
(configuredns 172.16.247.3:53_udp_dns)
(configuredns 172.16.247.2:53_udp_dns)
```

Appendix D

Case study

D.1 XML output of the information collection component

```xml
<?xml version="1.0"?>
<config name="case2">
  <hosts>
    <host id="10.0.1.2" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:7e:71:91" mtu="1500" />
        <interface mac="90:e2:ba:7e:71:93" mtu="1500" network="10.0.1.0" rate="1000000">
          <layer3address address="10.0.1.2" gateway="10.0.1.254" netmask="255.255.255.0" network="10.0.1.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="48:5d:60:76:bc:eb" mtu="1500" />
        <interface mac="d0:50:99:52:cc:80" mtu="1500" network="192.168.38.0" rate="1000000">
          <layer3address address="192.168.38.2" netmask="255.255.255.0" network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.2.1" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:82:52:2d" mtu="1500" />
        <interface mac="00:25:d3:39:ec:10" mtu="1500" />
        <interface mac="90:e2:ba:82:52:2f" mtu="1500" network="10.0.2.0" rate="1000000">
```

```xml
          <layer3address address="10.0.2.1" gateway="10.0.2.254"
        netmask="255.255.255.0" network="10.0.2.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="d0:50:99:52:cd:2e" mtu="1500" network="192.168.38.0"
        rate="1000000">
          <layer3address address="192.168.38.3" netmask="255.255.255.0"
        network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.2.254" location="physical" template="router-base">
      <interfaces>
        <interface mac="90:e2:ba:7f:78:49" mtu="1500" network="10.0.4.0" rate="100000">
          <layer3address address="10.0.4.2" netmask="255.255.255.0"
        network="10.0.4.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="90:e2:ba:7f:78:4a" network="10.0.5.0" mtu="1500" rate="100000">
          <layer3address address="10.0.5.1" netmask="255.255.255.0"
        network="10.0.5.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="48:5d:60:77:32:47" mtu="1500" />
        <interface mac="d0:50:99:52:cd:15" mtu="1500" network="192.168.38.0"
        rate="1000000">
          <layer3address address="192.168.38.5" netmask="255.255.255.0"
        network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="90:e2:ba:7f:78:4b" mtu="1500" network="10.0.2.0" rate="1000000">
          <layer3address address="10.0.2.254" netmask="255.255.255.0"
        network="10.0.2.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes>
```

```xml
        <route destination="10.0.1.0" gateway="10.0.4.1" genmask="255.255.255.0" interface="enx90e2ba7f7849"/>
        <route destination="10.0.3.0" gateway="10.0.5.2" genmask="255.255.255.0" interface="enx90e2ba7f784a"/>
        <route destination="10.0.6.0" gateway="10.0.5.2" genmask="255.255.255.0" interface="enx90e2ba7f784a"/>
      </routes>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.1.1" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:7f:78:4f" mtu="1500" network="10.0.1.0" rate="1000000">
          <layer3address address="10.0.1.1" gateway="10.0.1.254" netmask="255.255.255.0" network="10.0.1.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="48:5d:60:77:5b:db" mtu="1500" />
        <interface mac="90:e2:ba:7f:78:4d" mtu="1500" />
        <interface mac="d0:50:99:52:cc:f3" mtu="1500" network="192.168.38.0" rate="1000000">
          <layer3address address="192.168.38.1" netmask="255.255.255.0" network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.1.254" location="physical" template="router-base">
      <interfaces>
        <interface mac="64:f6:9d:12:db:7c" network="10.0.1.0">
          <layer3address address="10.0.1.254" netmask="255.255.255.0" network="10.0.1.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="64:f6:9d:12:db:7e" network="10.0.6.0">
          <layer3address address="10.0.6.1" netmask="255.255.255.0" network="10.0.6.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="64:f6:9d:12:db:7d" network="10.0.4.0">
```

```xml
          <layer3address address="10.0.4.1" netmask="255.255.255.0" network="10.0.4.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes>
        <route destination="10.0.3.0" gateway="10.0.6.2" genmask="255.255.255.0"/>
        <route destination="10.0.2.0" gateway="10.0.4.2" genmask="255.255.255.0"/>
        <route destination="10.0.5.0" gateway="10.0.4.2" genmask="255.255.255.0"/>
      </routes>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.3.254" location="physical" template="router-base">
      <interfaces>
        <interface mac="a8:9d:21:86:d0:ea" network="10.0.6.0">
          <layer3address address="10.0.6.2" netmask="255.255.255.0" network="10.0.6.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="a8:9d:21:86:d0:e8" network="10.0.3.0">
          <layer3address address="10.0.3.254" netmask="255.255.255.0" network="10.0.3.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="a8:9d:21:86:d0:e9" network="10.0.5.0">
          <layer3address address="10.0.5.2" netmask="255.255.255.0" network="10.0.5.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
      </interfaces>
      <disks/>
      <routes>
        <route destination="10.0.4.0" gateway="10.0.5.1" genmask="255.255.255.0"/>
        <route destination="10.0.1.0" gateway="10.0.6.1" genmask="255.255.255.0"/>
        <route destination="10.0.2.0" gateway="10.0.5.1" genmask="255.255.255.0"/>
      </routes>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.1.3" location="physical" template="host-base">
      <interfaces/>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
    <host id="10.0.3.1" location="physical" template="host-base">
      <interfaces>
        <interface mac="90:e2:ba:82:51:6f" mtu="1500" network="10.0.3.0" rate="100000">
          <layer3address address="10.0.3.1" gateway="10.0.3.254"
        netmask="255.255.255.0" network="10.0.3.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="48:5d:60:76:e1:ca" mtu="1500" />
        <interface mac="d0:50:99:52:cb:e4" mtu="1500" network="192.168.38.0"
        rate="1000000">
          <layer3address address="192.168.38.4" netmask="255.255.255.0"
        network="192.168.38.0/24" static="True">
            <services/>
          </layer3address>
        </interface>
        <interface mac="90:e2:ba:82:51:6d" mtu="1500" />
      </interfaces>
      <disks/>
      <routes/>
      <firewall>
        <firewallRules/>
      </firewall>
    </host>
  </hosts>
  <layer2networks>
    <layer2network id="10.0.6.0" location="physical" />
    <layer2network id="10.0.1.0" location="physical" />
    <layer2network id="192.168.38.0" location="physical" />
    <layer2network id="10.0.3.0" location="physical" />
    <layer2network id="10.0.2.0" location="physical" />
    <layer2network id="10.0.4.0" location="physical" />
    <layer2network id="10.0.5.0" location="physical" />
  </layer2networks>
  <layer3networks>
    <layer3network address="10.0.2.0" id="10.0.2.0/24" netmask="255.255.255.0"/>
    <layer3network address="10.0.1.0" id="10.0.1.0/24" netmask="255.255.255.0"/>
    <layer3network address="192.168.38.0" id="192.168.38.0/24" netmask="255.255.255.0"/>
    <layer3network address="10.0.3.0" id="10.0.3.0/24" netmask="255.255.255.0"/>
    <layer3network address="10.0.6.0" id="10.0.6.0/24" netmask="255.255.255.0"/>
    <layer3network address="10.0.4.0" id="10.0.4.0/24" netmask="255.255.255.0"/>
    <layer3network address="10.0.5.0" id="10.0.5.0/24" netmask="255.255.255.0"/>
  </layer3networks>
</config>
```

D.2 Plan for setup and configuration

```
(createhost 10-0-1-254)
(createhost 10-0-2-254)
(createhost 10-0-3-254)
(createhost 10-0-1-1)
(createhost 10-0-1-2)
(createhost 10-0-2-1)
(createhost 10-0-3-1)
(bootunnamed 10-0-1-254)
(bootunnamed 10-0-1-1)
(createnetwork 192.168.38.0)
(addconfignamenetwork 192.168.38.0)
(createinterface enxd0509952cd2e)
(createinterface enxd0509952cd15)
(createinterface enxd0509952cc80)
(createinterface enxd0509952cbe4)
(createnetwork 10.0.6.0)
(addconfignamenetwork 10.0.6.0)
(createinterface enxa89d2186d0ea)
(createnetwork 10.0.5.0)
(addconfignamenetwork 10.0.5.0)
(createinterface enxa89d2186d0e9)
(createinterface enx90e2ba7f784a)
(createnetwork 10.0.4.0)
(addconfignamenetwork 10.0.4.0)
(createinterface enx90e2ba7f7849)
(createnetwork 10.0.3.0)
(addconfignamenetwork 10.0.3.0)
(createinterface enxa89d2186d0e8)
(createinterface enx90e2ba82516f)
(createnetwork 10.0.2.0)
(addconfignamenetwork 10.0.2.0)
(createinterface enx90e2ba82522f)
(createinterface enx90e2ba7f784b)
(createnetwork 10.0.1.0)
(addconfignamenetwork 10.0.1.0)
(createinterface enx90e2ba7e7193)
(addconfignamehost 10-0-1-254)
(addconfignamehost 10-0-2-254)
(addconfignamehost 10-0-3-254)
(addconfignamehost 10-0-1-1)
(addconfignamehost 10-0-1-2)
(addconfignamehost 10-0-2-1)
(addconfignamehost 10-0-3-1)
(shutdown 10-0-1-254)
(bootunnamed 10-0-2-254)
(createinterface enx64f69d12db7e)
(createinterface enx64f69d12db7d)
(createinterface enx64f69d12db7c)
(configureinterface enxd0509952cd15)
(configureinterface enx90e2ba7f784b)
(configureinterface enx90e2ba7f784a)
(configureinterface enx90e2ba7f7849)
(configurerouting 10-0-2-254)
(shutdown 10-0-1-1)
(bootunnamed 10-0-1-2)
(createinterface enxd0509952ccf3)
(createinterface enx90e2ba7f784f)
(configureinterface enxd0509952cc80)
(configureinterface enx90e2ba7e7193)
(name 10-0-2-254)
(rebootandnamed 10-0-2-254)
(bootunnamed 10-0-1-254)
(configureinterface enx64f69d12db7e)
(configureinterface enx64f69d12db7d)
(configureinterface enx64f69d12db7c)
(configurerouting 10-0-1-254)
(name 10-0-1-254)
(rebootandnamed 10-0-1-254)
(bootunnamed 10-0-3-254)
(configureinterface enxa89d2186d0ea)
(configureinterface enxa89d2186d0e9)
(configureinterface enxa89d2186d0e8)
(configurerouting 10-0-3-254)
(name 10-0-3-254)
(rebootandnamed 10-0-3-254)
(name 10-0-1-2)
(rebootandnamed 10-0-1-2)
(bootunnamed 10-0-1-1)
(configureinterface enxd0509952ccf3)
(configureinterface enx90e2ba7f784f)
(name 10-0-1-1)
(rebootandnamed 10-0-1-1)
(bootunnamed 10-0-2-1)
(configureinterface enxd0509952cd2e)
(configureinterface enx90e2ba82522f)
(name 10-0-2-1)
(rebootandnamed 10-0-2-1)
(bootunnamed 10-0-3-1)
(configureinterface enxd0509952cbe4)
(configureinterface enx90e2ba82516f)
(name 10-0-3-1)
(rebootandnamed 10-0-3-1)
```

List of Figures

List of Tables

List of Listings

Bibliography

[1] P. Owezarski, P. Berthou, Y. Labit, and D. Gauchard, “LaasNetExp: A Generic Polymorphic Platform for Network Emulation and Experiments,” in *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, ser. TridentCom ’08. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 24:1–24:9.

[2] K. Ali, “Algorizmi: A configurable virtual testbed to generate datasets for offline evaluation of Intrusion Detection Systems,” Master’s thesis, University of Waterloo, 2010.

[3] T. R. Henderson, M. Lacage, G. F. Riley, C. Dowell, and J. Kopena, “Network simulations with the ns-3 simulator,” *SIGCOMM demonstration*, vol. 15, p. 17, 2008.

[4] ns-3 documentation: Internet Stack. Accessed: 2016-08-17. [Online]. Available: https://www.nsnam.org/docs/release/3.14/models/html/internet-stack.html

[5] S. Peach, B. Irwin, and R. van Heerden, “An Overview of Linux Container Based Network Emulation,” in *ACPI 15th European Conference on Cyber Warfare & Security (ECCWS)*, vol. 15, July 2016, pp. 253–259.

[6] X. Jiang and D. Xu, “vBET: A VM-based Emulation Testbed,” in *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, ser. MoMeTools ’03. New York, NY, USA: ACM, 2003, pp. 95–104.

[7] K. Wehrle, M. Gnes, and J. Gross, *Modeling and Tools for Network Simulation*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[8] M. Dorfhuber, “Information Collection for Temporal Variation Analysis on Networks,” Bachelor’s Thesis, Technische Universität München, 2016.

[9] K. Morris, *Infrastructure as Code: Managing Servers in the Cloud.* Sebastopol, CA: O’Reilly Media, 2016.

[10] Puppet Labs, “Puppet,” https://puppet.com/, 2005–2016.

[11] M. DeHaan, “Ansible,” https://www.ansible.com/, 2012–2016.

[12] D. Harrington, R. Presuhn, and B. Wijnen, “An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks,” RFC 3411, Internet Engineering Task Force, December 2002, updated by RFCs 5343, 5590.

[13] J. Schoenwaelder, “Overview of the 2002 IAB Network Management Workshop,” RFC 3535 (Informational), Internet Engineering Task Force, May 2003. [Online]. Available: http://www.ietf.org/rfc/rfc3535.txt

[14] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, “Network Configuration Protocol (NETCONF),” RFC 6241 (Proposed Standard), Internet Engineering Task Force, June 2011, updated by RFC 7803. [Online]. Available: http://www.ietf.org/rfc/rfc6241.txt

[15] J. Hoffmann, “Everything you always wanted to know about planning – (but were afraid to ask),” in *KI*, ser. Lecture Notes in Computer Science, vol. 7006. Springer, 2011, pp. 1–13.

[16] D. Nau, M. Ghallab, and P. Traverso, *Automated Planning: Theory & Practice.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004.

[17] M. Fox and D. Long, “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains,” *J. Artif. Int. Res.*, vol. 20, no. 1, pp. 61–124, December 2003.

[18] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins, “PDDL – The Planning Domain Definition Language,” 1998.

[19] J. van der Ham, F. Dijkstra, R. Łapacz, and J. Zurawski, “Network Markup Language Base Schema Version 1,” Muncie, IN: Open Grid Forum, report GFD-R-P.206, 2013.

[20] M. Ghijsen, J. van der Ham, P. Grosso, C. Dumitru, H. Zhu, Z. Zhao, and C. de Laat, “A Semantic-Web Approach for Modeling Computing Infrastructures,” *Computers & Electrical Engineering*, vol. 39, no. 8, pp. 2553–2565, 2013.

[21] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, “PlanetLab: An Overlay Testbed for Broad-coverage Services,” *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, pp. 3–12, July 2003.

[22] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, “GENI: A Federated Testbed for Innovative Network Experiments,” *Computer Networks*, vol. 61, pp. 5–23, Mar. 2014.

[23] B. Lantz, B. Heller, and N. McKeown, “A Network in a Laptop: Rapid Prototyping for Software-defined Networks,” in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.

[24] J. Ahrenholz, “Comparison of CORE Network Emulation Platforms,” in *Proceedings of the IEEE Military Communications Conference*, November 2010, pp. 864–869.

[25] S. Kreuzer, "Automation of Virtual Machine Setups for Network Experiments," Bachelor's Thesis, Technische Universität München, 2015.

[26] R. Bifulco, G. D. Stasi, and R. Canonico, "NEPTUNE for fast and easy deployment of OMF virtual network testbeds [Poster Abstract]," ser. ACM WiNTECH, 2010.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *OSDI02*. Boston, MA: USENIXASSOC, December 2002, pp. 255–270.

[28] T. Benzel, "The Science of Cyber Security Experimentation: The DETER Project," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 137–148.

[29] C. Diekmann, L. Hupel, and G. Carle, "Semantics-Preserving Simplification of Real-World Firewall Rule Sets," in *20th International Symposium on Formal Methods*. Springer, June 2015, pp. 195–212.

[30] M. Vukovic and J. Hwang, "Cloud Migration using Automated Planning," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 96–103.

[31] Python Software Foundation, "Python," https://www.python.org/, 2001–2016.

[32] S. Behnel and M. Faassen, "lxml - XML and HTML with Python," http://lxml.de/index.html, 2005–2016.

[33] M. Foord, N. Larosa, R. Dennis, and E. Courtwright, "ConfigObj," https://github.com/DiffSK/configobj, 2014–2016.

[34] D. P. D. Moss, "netaddr – A network address manipulation library for Python ," https://github.com/drkjam/netaddr/, 2008–2016.

[35] Fast Downward Homepage. Accessed: 2016-10-15. [Online]. Available: http://www.fast-downward.org/

[36] P. Fennell, "Schematron – More useful than you'd thought," in *XML London 2014 Conference Proceedings*. XML London, 2014, pp. 103–112.

[37] xmllint — command line XML tool. Accessed: 2016-10-14. [Online]. Available: http://xmlsoft.org/xmllint.html

[38] M. Helmert, "The Fast Downward Planning System," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191–246, 2006.

[39] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *Proceedings of the Nineteenth*

ACM Symposium on Operating Systems Principles, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 164–177.

[40] E. Mellor, D. Scott, and R. Sharp, "Xen Management API (API Revision 1.0.6)," https://downloads.xenproject.org/Wiki/XenAPI/xenapi-1.0.6.pdf, 2008, Accessed: 2016-11-11.

[41] M. Dorfhuber and C. Rudolf, "INSALATA," https://github.com/tumi8/INSALATA, December 2016, GitHub Repository.