Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3481

A Generic Artifact-Driven Approach for Provisioning, Configuring, and Managing Infrastructure Resources in the Cloud

Severin Leonhardt

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Frank Leymann

Supervisor: Dipl.-Inf. Johannes Wettinger

Commenced: May 31, 2013

Completed: November 30, 2013

CR-Classification: H.3.5, D.2.12, K.6

Abstract

Provisioning, configuration, and management of infrastructure resources in the cloud is difficult due to diverse APIs offered by cloud providers. Because approaches for a common API are still in an early stage and may not be broadly accepted, individual artifacts can be used to interact with different providers. They require generic properties to describe the configuration of infrastructure resources and combine them with provider-specific information provided by the user. Such generic properties are determined in this thesis by looking at the infrastructure offerings of 14 different providers. The artifacts can be made available in public repositories similar to configuration management scripts originating in the DevOps community. However, trust in their good nature is a challenge because in contrast to configuration management scripts they are executed in a shared management environment. To control and restrict the actions they are performing in this shared environment, a method to confine their execution has been developed. The Linux security module Tomoyo has been chosen as a foundation for this. A policy associated with each artifact describes the artifact's permissions in detail. The artifacts are used in the context of the OASIS Topology and Orchestration Specifiction for Cloud Applications (TOSCA), an emerging standard supported by a number of industry partners. This standard allows to model a topology of resources to be provisioned at a provider. Each infrastructure resource, such as a virtual machine, gets an artifact assigned for provisioning purposes. Based on this standard, two simple tools as well as artifacts for four providers were developed. They show the viability of this artifact-driven approach.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Cloud computing has been a trend in the IT industry for years. A large range of competitors provide resources on demand, ranging from specific software (Software as a Service) to basic infrastructure (Infrastructure as a Service). Along with these services, every provider offers an API to manage resources, for example to create new virtual machines. Most providers have developed their own API. This leads to vendor lock-in and difficulties in using multiple providers, thus preventing the customer from using the best provider for each task.

Several attempts have been made to achieve a common API for Cloud providers. Unfortunately, most major providers keep their proprietary API instead of switching to an open standard. A remarkable exception is Rackspace: their service is fully based on the open standard OpenStack, to which they are a major contributor.

Managing resources at, or even across several providers requires more than just a common API. As suggested in [FRC+13], a model-driven approach is reasonable. The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) is such an approach, allowing to model the required entities ranging from servers to databases and applications.

For each of these entities (called nodes in TOSCA) some activities need to be performed to install, start, or stop it. These activities can be implemented by artifacts assigned to nodes. For nodes representing virtual machines or other infrastructure resources, this is currently not done. They are abstract, without an implementation. The actual logic must be within the TOSCA container, which is the application that accepts a Cloud Service Archive (CSAR) and allows to run the artifacts contained therein. The goal of this thesis is to show that this logic can be moved into artifacts assigned to infrastructure nodes, requiring no provider-specific knowledge by the TOSCA container. Artifacts implemented as part of this thesis are simple script files. These scripts are executed in a shared environment, shared between all artifacts and the TOSCA container itself.

These artifacts are meant to be shared among people to prevent everyone from having to write their own. This raises the issue of trustworthiness, since these artifacts are nothing less than code being executed to perform important tasks for the user. Part of this thesis is concerned with the issue of ensuring that artifacts only perform actions they are supposed to perform.

1.1 Outline

The basis for this thesis are generic properties for infrastructure resources and secure execution of applications. Integrating the findings with TOSCA by developing some tools is the main

part. The results are then evaluated and conclusions are presented, as well as some ideas on how to further improve the tools. All of this is divided into the following chapters:

Chapter 2 – Background about the thesis' topics and foundation for the following work.

Chapter 3 – Interfacing with different Cloud providers using a generic interface.

Chapter 4 – Confinement of executed provisioning artifacts.

Chapter 5 – Integration of the provisioning artifacts with TOSCA.

Chapter 6 – Evaluation of the developed artifacts, the confinement, and the integration.

Chapter 7 – Conclusions drawn from the work done in this thesis.

Chapter 8 – Future Work to improve the developed prototype.

1.2 Abbreviations

Several abbreviations are used in this thesis. Except for the very common ones, all are listed in here:

Abbreviation	Full Name
CSAR	Cloud Service Archive
EC2	Amazon Elastic Compute Cloud
ERB	Embedded Ruby (Template Engine)
iSCSI	internet Small Computer System Interface
LSM	Linux Security Module
OCCI	Open Cloud Computing Interface
TOSCA	Topology and Orchestration Specification for Cloud Applications

1.3 Typography

Within this thesis, source code, command-line tools, configuration files, and other files are discussed many times. Without appropriate typography to show the context of words, ambiguities are inevitable and readability will suffer. Therefore, some typographical conventions are introduced here which will be used throughout the document.

Listings contain important or longer pieces of shell sessions, XML markup, or any content of files. Shell sessions show a shell prompt, the command to input, and the output. The shell prompt is # for the root user and $ for other users. An example of a shell session is shown in Listing 1.1.

Pieces of a longer Listing within the text are either shown inline (`date +%B`) or as follows, if they are longer:

```
groups | tr ' ' '\n' | sort | head -n 1
```

Listing 1.1 Example Shell Session

```
$ date +%B
November
$ su
Password:
# groups | tr ' ' '\n' | sort | head -n 1
# audio
```

URIs, XML namespaces, and names of command-line applications and their invocation are shown in the same way. It is clear from the context to which the text belongs. File and directory names are shown in the same font as well. They can be recognized by their extension, if they are a file (`config.xml`), or by the trailing slash, if they are a directory (`/usr/bin/`).

Contrary to XML namespaces, XML element names are in the same font as normal text. Most of the time the element names are from TOSCA, for example NodeTemplate. When such an element name is used, it refers not only to the actual XML element but also to the object defined by that element. To stay with our example, a NodeTemplate is not just an XML element, it is a template for a node which is later provisioned. Because of this, the element names seamlessly fit into the text and are therefore not highlighted in any way.

2 Background

This chapter provides the foundation for the following chapters. First, it shows the current state of the art in generic provisioning of infrastructure resources. Next, we introduce abstraction libraries for Cloud APIs that reduce the hassle to interact with different Cloud providers. Taking the abstraction even further, we also introduce model-driven Cloud management.

2.1 Cloud API Abstraction Libraries

There is a massive number of Cloud providers, each differing in several aspects. Almost all offer their services through their own proprietary API. This has been identified as a major drawback [FGJ+09] and a lot of work is being done in standardizing each aspect of the Cloud [Bur10]. OpenStack is one of the already progressed standardization efforts and in use by a few Cloud providers as their main and only API.

Despite many standardization efforts in Cloud computing, most providers still maintain their own API. This makes it difficult to interact with and migrate between different providers. Several abstraction libraries have been developed that try to mitigate this issue. While researching the topic, four abstraction libraries were found which support a broad range of providers and are still in active development. These are deltacloud[1], fog[2], libcloud[3], and jclouds[4]. The development and targeted programming language for each is shown in Table 2.1.

These libraries allow to interface with different providers without having to know the API of each individual provider. The degree to which individual APIs are combined and the degree of

Library	**Programming Language**	**API for**
fog	Ruby	Ruby
Delta Cloud	Ruby	Ruby, C, any REST capable
jclouds	Java	Java, Closure
libcloud	Python	Python

Table 2.1: Cloud API Abstraction Libraries

[1] `http://deltacloud.apache.org/`
[2] `http://fog.io/`
[3] `http://libcloud.apache.org/`
[4] `http://jclouds.incubator.apache.org/`

support of all the provider's features varies among the libraries. There is Deltacloud, which has its own well documented API. It is the only way to interact with the library. If any Cloud provider offers features not reflected in this API these features will not be available. On the other side, there is the fog library. This library does not have a specific API, only some general abstraction concepts. For simple use cases, this gives common method names and parameters across different providers. But when one wants to use functionality specific to a provider, the library allows to do so by offering methods that reflect this specific functionality of that provider.

2.2 Bare Metal and the Cloud

Most of the time, Cloud computing is about infrastructure resources in the Cloud, meaning the actual physical hardware is transparent to the user. But especially in private Clouds, the physical resources can be visible to the user. After all, it is hardware physically known to the user. It might even be desired, because some hardware could be specialized for certain tasks. Take a machine connected to a DCF77[5] device as an example, which will have the most accurate clock.

Provisioning this physically existing hardware is called bare metal provisioning. Instead of some virtual resource, a concrete physical device is managed. The management itself does not differ much from managing virtual resources, because the latter still follow the principles of physical resources. They are just not implemented that way. For example, a compute instance still looks like a computer with an operating system, a CPU, memory, etc.

Several tools exist to provision ones own hardware. The Foreman[6] is one of them. Although its main purpose is to manage physical machines it can, in turn, manage resources at Cloud providers as well. Within this thesis, the Foreman will also be considered, not to provision Cloud resources but actual physical hardware.

The artifact-driven approach to provision resources developed in this thesis is not limited to resources in the Cloud. It works for any kind of resource, as long as an artifact exists to provision these resources. The Foreman is a good example to show this.

2.3 Model-Driven Cloud Management

Using abstraction libraries to provision infrastructure resources can be labeled imperative. Commands are given to the library in the form of function calls. When doing so, we always assume a certain configuration (no deployed resources yet) and write code to transform to another configuration (all resources are deployed).

[5] German long wave signal transmitting the accurate time.
[6] `http://theforeman.org/`

Imperative configuration is commonly contrasted with declarative configuration. In our case, this means we merely describe the desired configuration (how which resources should be deployed) and it is the job of some tool to ensure this configuration. In [FRC+13] a case is made for using model-driven engineering methods to describe resources required from a Cloud provider. The result is the suggested Cloud Modeling Language (CloudML). But there are more of these projects that aim at describing resources in the Cloud. Based on these descriptions the resources can be provisioned. The TOSCA standard [TOSa], which is used in this thesis, is such a project.

Imperative and declarative are in no way exclusive to each other in this context. The differentiation is merely described from an end user's perspective, how he interacts with a tool. A tool which allows us to describe our configuration in a declarative manner will most likely use API calls of the provider which are imperative. This is how both concepts come together.

3 Interfacing

Many attempts have been made at creating a common API or at least concepts to interact with different Cloud providers [Bur10]. This chapter will not describe yet another one of those attempts. Its focus is quite different.

Within this thesis, there is no need for a common API because provider-specific artifacts will be employed. These artifacts communicate with a provider in whatever means are appropriate. But some input needs to be provided to these artifacts, some information about the properties of the resource to deploy. This can be the size of a disk, the name of a server, etc. Together with as little provider-specific additional information as possible, these generic properties must allow the artifact to deploy a resource.

Finding such generic properties was the goal of this chapter. These properties are different for every type of infrastructure resource. Three common types of infrastructure resources are handled in this thesis:

1. compute resources (also called virtual machines or servers)
2. block storage resources
3. firewalls and their rules

If there were a common API or framework which requires a set of values to deploy a resource at any provider, these values could be used as our generic properties. In fact, there are a few APIs and frameworks attempting to provide a common way to interface with multiple providers. At first, an overview of these and their support of Cloud providers is given in Section 3.1.

To anticipate the outcome, none of the APIs or frameworks did offer a truly generic interface applicable for many different providers. The approaches taken by these solutions were very different. A comparison of these approaches is given in Section 3.2. This should give some insight why no universal API or framework exists and what this means for our attempt at finding generic properties for resources.

The actual work of figuring out generic properties for the infrastructure resources is described in Section 3.3. For each type of infrastructure resource, the possible candidates for generic properties were examined. A short discussion about the candidates led to a set of properties which can be considered generic.

The results and some general findings are summarized in Section 3.4. Based on these, prototypes for a few selected providers are developed, which is described in Section 3.5.

API	Supported Providers
individual	42
EC2	4
OCCI	1
OpenStack	6
vCloud	2

Table 3.1: Number of Supported Providers per API

3.1 APIs and Frameworks for Cloud Providers

There are quite a few readily available APIs and frameworks that allow one to access multiple providers. Four major frameworks have already been presented in Section 2.1. During the research for this topic, two major APIs and two less used APIs were found as well. The major APIs were OpenStack[1] and the EC2 API[2]. The less used APIs were OCCI[3] and vCloud[4]. These APIs and frameworks were evaluated with regard to the number of Cloud providers they can be used with. Only compute resources have been considered for brevity, no storage or other types of resources. This already gives a good impression of the extent of supported providers though. A total of 49 providers have been considered. They all result from the list of supported providers of each frameworks.

In Table A.1 a checkmark indicates for each provider through which API he can be accessed. The first column, called "individual", is used when the provider runs his own API, which is not shared with any other provider. The supported API of a provider has been determined from the available documentation. A summary of the number of checkmarks per API is shown in Table 3.1.

What is most obvious from the summary table: close to all providers have their individual API instead of a common one. This rules out the possibility to derive generic properties from a single API, which would have made things quite simple.

In Table A.2 the same comparison is made for libraries. A checkmark indicates that a provider can be accessed through this API. The table is based on the documentation or list of available plugins of the library. A summary of the number of checkmarks per library is shown in Table 3.2.

The summary table shows nicely how fog and libcloud support more than half of the Cloud providers. One can argue this is because the list is based on what the libraries supports. But looking at the list of providers, all the major and not so major providers are there. It is difficult to find more providers in addition to what is already there.

[1] http://www.openstack.org/
[2] http://aws.amazon.com/de/ec2/
[3] http://occi-wg.org/
[4] http://pubs.vmware.com/vcd-51/index.jsp?topic=%2Fcom.vmware.vcloud.api.doc_51%2FGUID-86CA32C2-3753-49B2-A471-1CE460109ADB.html

Framework	Supported Providers
fog	26
Deltacloud	13
jclouds	15
libcloud	33

Table 3.2: Number of Supported Providers per Library

On a final note, a checkmark was only set if an API offers direct access to a provider. It is possible that a provider might be indirectly accessible. For example, it is possible to use OCCI to interface with OpenStack, thus OCCI can be used to interface with any provider that supports OpenStack. These indirections always require some intermediate service which translates the API calls. The provider is not accessed via a simple API anymore, it is a different basic operation model.

3.2 Basic Operation Models

To summarize the previous sections: A diverse range of solutions has already been presented to communicate with several Cloud providers in a common way, at least in key aspects.

This sentence sounds very vague, which is due to huge differences between presented solutions. To grasp the range of differences each vague part of this sentence is first explained. The word "solution" was used because it can either be an application, a library, or an API. Only the generic term "solution" fits all of them. The way how data is sent to and received from the Cloud provider is different, too. The word "communicate" implies a direct communication less than the word "interface", which is typically used. In fact, communication with the provider does not necessarily have to be direct, but might go through some intermediate. The data transferred during the communication can even be modified. The last part of the sentence is the "common way", which is even limited to "key aspects". These key aspects refer to basic operations on resources. Some presented solutions only have a common way of interacting with different providers for very basic operations, for example listing all virtual machines. Therefore it is limited to key aspects.

The section's title is called "basic operation models", even though the chapter is about interfacing. The term "operation model" highlights the fact that the differences are not only in the interface but in the actual way the solutions work. Some solutions have to do some work before a request or response passes from one side to the other.

In [LKBT11] the directions of many attempts to achieve a common interface are shown. They are presented in three groups:

- standardization initiatives
- Cloud computing interoperability frameworks
- semantically interoperable Cloud solutions

The classification presented in this chapter is roughly based on these groups. Only the focus in each group is more user oriented and formulated in simpler terms.

3.2.1 Standardized API

The first class of solutions presented in this thesis are APIs standardized among different Cloud providers. Those need not be standardized by a consortium but merely used by different providers thus being a de facto standard.

A standard developed by a consortium is OpenStack. There are a few providers using it as their main API. It supports core concepts like compute resources and storage and it is extensible. Security groups, as an example, are implemented as an extension to the core OpenStack API.

A de facto standard is the API for Amazon's EC2. It is implemented by a few other providers to allow easy migration away from Amazon to their service. This gives a common API for different providers, too.

It has already been shown that most providers use their own API. This means there is currently no single API one can use and still be able to use a great range of Cloud providers. The reasons are manifold, just to name a few: It would require current providers to change their API, a huge burden for existing customers and the provider's backend. The customer has the opportunity to more easily switch to another provider, which is bad for the current provider. Finally, some features, maybe the killer feature of the provider, might not be accessible through the API or the API is difficult to extend in that direction.

3.2.2 Mediation

There is no standardized API the majority of providers have agreed upon. Instead of requiring the providers to implement such an API an intermediate tool can be utilized to map a standardized API to provider-specific APIs. The result is the same: one API can be used to interface with many different providers. This is what the mediation operation model is about. A service between the end user and the provider mediates requests and responses, meaning they are being translated from one format to another format. Maybe even more logic is applied where one requests results in multiple outgoing requests or vice versa.

An implementation of this operation model is Deltacloud. It offers its own API through which users can communicate with different Cloud providers to provision infrastructure resources. This requires an instance of Deltacloud to be running somewhere. It should also be a trusted instance since credentials are passed through it as well, a publicly available installation might not be the best choice. Leaving this aside, there is another shortcoming. Because the mediation service has its own API, this API limits what can be done. Only if the API supports a feature will it be available to the user. Judging from what has been seen so far, it is difficult to develop a common API for many Cloud providers, the API of the mediation service has the same problem. It allows to interface with other Cloud providers, but only within the scope of its

own interface. Special features, only one or a few Cloud providers offer, might not make it into the common API and are thus unavailable.

3.2.3 Partial Generic

The mediation operation model decouples the common API from the Cloud provider. Going one step further this operation model decouples the common parts of an API from the specific parts, leading to an API that is two-parted. One part offers methods that work the same across many or all providers. These will typically be basic functionality like creating and destroying resources. The other part offers individual methods for each provider to access its individual functionality. These will be very specific features like ordering a backup plan for a server.

The fog library follows this approach. It has models and collection of models for most resource types. The way to access and list these models is always the same. But individual fields of a compute resource model differ from provider to provider.

All features of the providers are available, but it is not a single common API anymore. This means some provider-specific logic is necessary to use some features.

3.3 Deducing A Solution

Since it is clear there is no single API or framework usable with most Cloud providers, we have no single source to derive generic properties from. But we need these generic properties to be able to describe an infrastructure resource in a provider-independent way. Only as little information as possible should be required in addition when a specific provider is selected. For that reason, this section is dedicated to finding generic properties of infrastructure resources at different providers.

A simple comparison of parameters required for API calls at each provider was performed. When comparing the parameters, commonalities become apparent and lead to a list of generic properties. This was done by looking at the requests for authentication, creating compute resource, creating and attaching storage resources, and creating and assigning security groups at each provider. A subsection is dedicated to every request type.

The number of existing Cloud providers is overwhelming, our list contains 49 providers. Initially, it was planned for the comparison to consider all providers found in the previous section. But this is impossible with the time available in this thesis. Since time is the limiting factor, it was easy to find a criteria to filter out providers. Some providers offer very simple and concise documentation, while others offer vast amounts of text to read through. Therefore the simplicity and clarity of the documentation was the criteria whether to include a provider in this analysis or not. Although this is a somewhat subjective criteria it still leaves 14 providers to be considered in the comparison. Instead of Ninefold, which uses the CloudStack API, CloudStack itself was considered.

The detailed list of properties per provider and their mapping to generic names take up a lot of space. Therefore, they are not shown here but in Appendix A.1 to not hinder the flow of reading.

3.3.1 Authenticating

The first step when interacting with a Cloud provider is authentication. Table A.3 shows what information is needed to authenticate with different Cloud providers. As one might expect, most providers use a combination of username and password. Only Linode allows using just a password, more precisely a secret key generated by the user beforehand.

Some providers offer their services in different regions or, as with OpenStack, the API is not located at the same address in all cases. Therefore, it is necessary to connect to an endpoint that identifies the region or system one wants to work in. While not actually being part of the authentication, this endpoint must be known before anything else can be done. Since authentication must also be done before anything else can be done both were combined.

The resulting generic properties were:

- Endpoint: a region or URL to connect and authenticate to.
- Username: a name which does not need to be kept secret and identifies the user.
- Password: a secret phrase that identifies the user (in combination with the username).

Semantically, these are generic properties. Some providers just give these values quite different names where the correspondence to username and password is not always obvious. Take AWS' access key ID and secret access key as an example. It can be difficult for an end user to figure out what he as to use as username and what as password. In addition, the actual values are inherently provider specific. You do not share the same credentials with several provider but have a unique username and password combination for each. For these two reasons, it is not appropriate to generalize the authentication. The properties are similar among providers, but they are not generic.

3.3.2 Creating Virtual Machines

While authentication is pretty much the same across all providers, it is quite the opposite with creating virtual machines. There was not one generic property that is applicable across all providers, as can be seen in Appendix A.2.2. A closer look at the differences was necessary.

Taken from Table A.4, the most used generic names were:

1. Flavor (10 times)
2. Location (8 times)
3. Image (7 times)
4. Hostname (5 times)

5. DisplayName (5 times)

Most providers offer different flavors (also called plans or configurations) for their servers. Only a few (GoGrid and Serverlove) allow to specify concrete values for memory and disk size. To describe the desired resource in a way that works with both, an idea from [FRC+13] was borrowed and improved. It had been suggested to specify a minimum and maximum value to define in which range the value has to be. This was extended by a third value, the desired value. When provisioning a resource, a value as close to the desired value as possible must be used. The range limits how far the search can go. This ensures the value is not too small to be unusable but also ensures it is not too high leading to unnecessarily high costs. Only the desired value is required, if no limiting minimum and maximum values are given the search range is not limited.

The location is different than the region mentioned for authentication. The region is an endpoint which cannot be switched while being connected. The location is within a region and can be chosen for every request. Providers have different names for the location property, for example calling it availability zone (Amazon), facility (Voxel) or data center (Linode).

Location and region are similar which is why the generic applicability was in question here. The property name, i.e. the meaning, might be similar, the actual values are not. Every Cloud provider uses his own names to address his data centers around the globe. Some use city names in the location name (BareMetalCloud: miami-fl-usa), others use numbers within continents (Amazon: eu-west-1a). To be usable, some generic values are necessary which must be mapped to provider-specific values. This is not an easy task. There might be multiple close locations or no location on a continent at all. No method to easily find and maintain such mappings was found. Therefore, the location was considered provider specific.

Images contain a complete filesystem which is written to the disk of the virtual machine when it is created. These images contain a basic operating system, sometimes preconfigured for certain tasks. With all providers, the images have descriptive names to indicate what they contain. But the pattern these images follow can be different, maybe even for a single provider. Using a search pattern with the right words, an image with desired properties can be found across different providers. As an example the wildcard pattern “Ubuntu * 11” matches all images with Ubuntu in version 11. With some bad luck this can also match the Image “Ubuntu 10, NOT 11!”. Well named images need to go together with a user who reviews the actually selected image. Then the image of a virtual machine can well be a generic property by using a wildcard pattern.

Not many providers support setting a hostname. But the hostname is a very important aspect of a virtual machine. For that reason it was considered to be a generic property anyway.

The display name is the name of the server shown in the administrative interface of the Cloud provider. This was also deemed to be a generic property, even if not supported by many. It allows to give the resource a name, which can be used to easily refer to that resource.

A description is supported by even less providers, but still it was considered to be a generic property. Being able to write down some notes about a resource can always be helpful, regardless whether that description will be visible at the Cloud providers interface, or not.

This gave rise to the following generic properties:

- CpuRange: a range in which the number of CPU cores must be.
- DiskRange: a range in which the disk size must be.
- MemoryRange: a range in which the memory size must be.
- Description: a description of the server.
- DisplayName: a name to identify the machine. (Different from the hostname.)
- Hostname: The hostname of the server.
- Image: a filesystem image to write to the server's disk.

The CpuRange, DiskRange and MemoryRange are required ranges with desired values. Only these desired values are required for a range, the minimum and maximum are optional. The Description, DisplayName, Hostname, and Image are all optional because not all providers require them.

3.3.3 Creating Block Storage Devices

Block Storage Devices store blocks of binary data, like your typical hard drive. In most cases, they are transparently attached to virtual machines looking like directly connected block devices. In some cases the connection is less transparent, as with BareMetalCloud. They call their storage devices targets and use iSCSI to make them accessible by the servers. The virtual machine needs to set up iSCSI on its own to access the device. Using the API, only a permission is granted to access a target.

Taken from Table A.5, the most used generic names were:

1. Size (8 times)
2. Location (6 times)
3. Snapshot (4 times)
4. Name (3 times)

Obviously the size is used by all Cloud providers. Without doubt, this is a generic property for a block storage device.

As with virtual machines, it is possible to specify the location of the block storage device. For the same reasons as mentioned before, this is a provider-specific property.

Some providers allow to take snapshots of block storage devices. When creating a new block storage device, it can be based on a previously created snapshot. This means the content of the new device is exactly the same as contents of another device when the snapshot was created. Snapshots are addressed by an identifier, but this is by no means generic. Snapshots also have a descriptive name. Using a pattern matching approach, as was suggested for the image of virtual machines, a snapshot can be found based on its name. This gives a generic

way of addressing a snapshot, allowing us to include it as a generic property. Creating a block storage device does not require a snapshot so the generic property is optional.

Yet another similarity to virtual machines is the name property. It is not supported by many providers but for the same reasons as before, this property was also considered being generic.

For block storage devices, these are the generic properties:

- Size: the size of the block device.
- Name: the name of the block storage device.
- Snapshot: an optional snapshot from which to clone.

The Size is required. The Name and Snapshot are both optional because they are not supported by all providers.

3.3.4 Attaching Block Storage Devices

Most Cloud providers separate the creation of storage devices and virtual machines from attaching them. Using some API call, an existing device is attached to a running virtual machine. At Serverlove, this is different. They require all devices to be attached to the machine before it is started.

There is not much information to provide, when attaching block storage devices. The identifier of a virtual machine instance and the identifier of a block storage device must always be used. At most providers, the device name to use within the virtual machine can be set as well. But in our understanding of a block device, which was established before, it might also be iSCSI. For iSCSI a device name is not applicable, therefore the device name was not considered a generic property.

To attach a block storage device to a virtual machine these are the generic properties:

- Instance: the instance to which to attach the device.
- Volume: the block storage device to attach.

Both of these properties are required.

3.3.5 Creating Security Groups

Security groups are a way to group several network filtering rules. Sometimes security groups are also referred to as firewalls, they group firewall rules. Grouping rules allows easy assignment of several rules to one virtual machine.

Not all providers support creating network filtering rules but almost all who do allow grouping these rules. Joyent is the exception here. Instead of using security groups to enable several rules for some servers all rules are global for the account. For OpenStack, security groups are an extension and not part of the core specification.

If security groups are supported by a provider they always have a name and almost always a description. Both were strong candidates for being generic properties. As always, some providers allow more. As an example, GoGrid allows enabling and disabling security groups.

As discussed before, having a name and a description for a resource is helpful in any case. The resulting generic properties for security groups were therefore:

- Name: the name of the security group.
- Description: a description of the security group.
- Rules: a set of rules of the security group.

The name is required to identify a security group but the description is optional. This is to stay in common with the other infrastructure resource types where a description is also optional. If it is required for a specific provider it can always be queried from the user. A security group must contain a list of rules, but that list can be empty.

3.3.6 Creating Security Group Rules

Security group rules specify from which network source access is allowed to which network target. These can be inbound (ingress) or outbound (egress) connections.

AWS supports multiple sources per rule. This is not the case for any other provider. Instead of a single rule with multiple sources the same can be described using multiple rules with a single source. The generic rules therefore only allow a single source.

We have already seen with block storage devices that resources might work differently across providers. This is very much the case for security group rules. Common is only the possibility to filter by single TCP ports or ranges of TCP ports. Fundamentally different are the ways how the source and the target are being matched.

- **Unidirectional Rule:** An outside location and a direction are specified. Outside location means an IP address or network outside of the current security group. With some providers even another security group can be used as the outside location. Depending on the direction, either traffic coming from that location or going to that location is matched.
- **Bidirectional Rule:** A local address and a remote address are specified. This is similar to the unidirectional rule, but here the direction is implicit. Traffic from the local to the remote address or in the reverse direction is allowed.
- **Source-Target Rule:** A source address and a target address are specified. In this case the source or target can be either remote or local. Traffic going from the source to the destination is matched.

Due to these very different types, no abstraction was found that still provides the full potential of each solution. The only possibility was to use a subset of the functionality provided by each provider. Typically, a firewall protects incoming network access and only allow traffic through certain ports. This can be described in each rule type. For the unidirectional rule, the direction is incoming and the port to allow is specified. For the bidirectional rule only used by Zerigo, a remote IP can be left out giving the desired behavior. For the source-target rule used by BrightBox and Joyent, a source of 0.0.0.0/0 matches all IP addresses.

The resulting "generic" properties for security group rules were therefore:

- Protocol: protocol to allow (TCP, UDP, ICMP)
- PortRange: traffic to these ports is allowed.

The protocol is required and at least the first port of the range. The last port can be omitted, it will then be interpreted to be the same as the first port. This makes the rule applicable for a single port.

3.4 Consequences

To sum it up, it depends very much on the type of the infrastructure resource whether there are commonalities among providers. These commonalities do not necessarily lead to generic properties. It sometimes is more appropriate to have a provider-specific property than to force it being generic.

A good example for this is authentication. It is similar among different providers, but due to its provider-specific nature having generic properties is not appropriate. Attaching block storage devices and joining security groups is very similar among providers, too. In many cases these can be used by only having generic properties. Creating block storage devices does not differ much among providers. The most important aspects can be put into generic properties. For virtual machines it is more diverse, but the most important aspects can be put into generic properties as well. The highest diversity exists among security group rules, which work fundamentally different among providers. Generic properties only allow to use a small subset of the possible functionality the rules can offer.

The previous sections indicate that a truly generic solution to access many providers is not possible, most likely due to the high diversity among them. The differences in the provided functionality, and thus their API, can even be seen as a competitive advantage [LKBT11]. Still, the attempt at finding generic properties for different types of infrastructure resources was successful. Even though provider-specific information is needed as well, the generic properties are a foundation. Based on this foundation, artifacts for individual providers can deduce a lot of required information and only request a few additional pieces of information, if necessary at all.

3.5 Prototypes and Future Artifacts

Even with a framework at hand, it was too time consuming to implement an artifact for each of the 14 presented provider. As long as a library is used, there is not even much gain in doing so since the complex and therefore interesting logic is already in the library. Still, as part of this thesis a few artifacts for different providers were implemented to show the broad support of the developed concepts and system. Four providers were selected. **Amazon** is the first one, because it is a, if not the, major player in offering public Cloud computing. **OpenStack** was also targeted, since it is a standardized and broadly supported API. **BareMetalCloud** was included, because it is different from a typical, fully virtualized Cloud infrastructure. Finally, **the Foreman** was targeted, which is not a provider but a piece of software. It manages actual bare metal provisioning and is thus an interesting addition to Cloud based providers. These very different targets are diverse enough to show the general applicability of the developed system.

Only the fog library provided access to three of the selected providers. By using fog to interface with these providers, the artifact development is greatly simplified. Ruby was the only choice as the programming language of the artifacts because fog can only be used with Ruby. To interface with the Foreman, a different road was taken. The artifact was implemented using Bash scripts, thus further demonstrating the diversity allowed by the solution developed in this thesis.

A prototype was created for each provider, showing how to interface with the provider. They were simple Ruby scripts, but they already performed all necessary work to have an infrastructure resource deployed. The artifacts, which were developed later, are based on these prototypes. While writing the prototype for BareMetalCloud it was discovered that the support in fog was missing some requests, collection and model classes. These classes greatly simplify using the API. The missing pieces had to be implemented first.

4 Confinement

The result from the previous chapter were some scripts that interface with providers. These scripts must be executed somewhere, and it can obviously not be the machine to provision. They need to be run in a shared management environment, all artifacts share this environment with other artifacts and with the TOSCA container. Either their execution is performed on a dedicated machine, or on the same machine which hosts the TOSCA container. Especially in the second case, we need to be concerned with the capabilities these artifacts have. They can, in a typical setup, access most of the local system, write files, kill processes, access the local network, and the whole Internet. Since the source of these artifacts and their inner workings may not be known, we should not blindly trust them and limit what they can do.

The goal of this chapter was to find means to confine artifacts, making sure they behave only in expected ways. First we defined our requirements from which we then derived applicable existing solutions. Finally we implemented a prototype showing how to utilize the confinement solution.

4.1 Requirements

The most important requirement is, of course, that artifacts can be prevented from harming our systems or leaking information to the outside world. This basically means we want to restrict access to the local system and to the network. Even when an artifact was written with malicious intent to circumvent our security measures, it should not be able to do so.

The artifacts can make use of any scripting language or even contain binary applications. The confinement solution must therefore be able to confine any type of application, without requiring the application to be modified.

In addition we want to have as little complexity as possible. Neither configuring the solution nor defining what is allowed and what not should take much time or require a lot of knowledge. The effort required from artifact developers must be low enough for it to still be worthwhile.

The overhead of the confinement must also be low. It should not require a long setup or teardown time and the execution duration should not be increased significantly.

4.2 Finding a Solution

There is quite a range of options to choose from to confine applications [SMP13]. But their requirements and capabilities are very different. Most of the solutions did not fulfill all of our requirements. A major limiting factor was the requirement to confine any application without modifications to it.

There were basically two options. One of them was container virtualization using tools like LXC[1]. A lightweight virtual machine is set up in which the artifact is being run. This at least allows arbitrary applications to be run and provides safety for the local system. But on the downside, we have a huge overhead and at least some complexity in managing this virtualized container. Files need to be copied, the container needs to be set up, etc. In addition, this offers no way to restrict access to the network. To achieve our required level of confinement a firewall must be used as well, increasing the level of complexity.

The other option was enforcing security restrictions on the operating system level. This option looked very promising, it allows for fine-grained control and has a low overhead compared to container virtualization. The only downside is that it makes the solution operating system dependent. During the research, no solution was found that works across different operating systems and it seems plausible there will be none. Operating systems work differently, and thus a so tightly integrated solution cannot provide the full set of security features available for each operating system. Therefore, we had to consider the operating system we wanted to use as well.

Linux offers an interface to load different security modules (LSMs) that enforce restrictions on a kernel level. It is possible to confine applications in a very fine-grained manner. Well known LSMs are AppArmor and SELinux. But we also had a look at the less known but still very capable Tomoyo.

Windows offers Mandatory Access Control since Vista, calling it Windows Integrity Control [WIC]. It allows to define several integrity levels of which each process and file is a member of. Processes with a lower integrity level cannot access processes or files with a higher level. This is a very coarse-grained security mechanism. Our artifact can have the lowest integrity level and thereby be disallowed to harm the system much. To restrict network access a personal firewall must be used.

Due to more comprehensive confinement options available on Linux these were further investigated. Each of the three mentioned above were judged with regard to our requirements.

[1] `http://linuxcontainers.org/`

4.3 Comparison of Linux Security Modules

To compare the different LSMs we used the following setup:

One Ruby script which, when well behaving, accessed one IP address to perform some HTTP requests. This behavior was allowed. The script was then replaced with some malicious scripts. The first malicious script downloaded the Google main page to the local file system. This must be prevented, since our well behaving script neither required accessing Google nor writing to the local file system. The second and third malicious scripts tried to remove the home directory or the root directory respectively. This must be prevented as well, no file must get removed.

The comparison considered the following aspects:

- **Installation**: The Linux security module must be installed and configured to work. Although it is only done once, this step should not be too complex.
- **Policy creation**: When an artifact is developed a policy must be written. The process of creating a policy must be as straight-forward as possible. Simple syntax and easy-to-understand concepts are an advantage.
- **Policy installation**: Before an artifact is executed the policy must be installed. This should be fast and possible in an automated fashion.
- **Supported restriction options**: The module must allow confinement of file and network access. The possibilities to limit or allow access should be fine-grained but easy to understand at the same time.

All of these tools use the word “policy” in the same meaning. It refers to the combined set of all rules which apply to the system. There is only one policy, which is loaded into the kernel or stored on disk. This is not well suited for our context, where many different artifacts have their policy rules associated with them. Therefore we slightly redefine the term. From here on a policy is not the combined set of all rules but simply a set of rules. Several policies can be combined to obtain one large policy, which ultimately leads to the policy containing all rules. This allows us to refer to the set of rules associated with each artifacts as the policy of the artifact.

4.3.1 AppArmor

The Wiki of the AppArmor project offers a good self-description of AppArmor:

> “ AppArmor is an effective and easy-to-use Linux application security system. AppArmor proactively protects the operating system and applications from external or internal threats, even zero-day attacks, by enforcing good behavior and preventing even unknown application flaws from being exploited. AppArmor security policies completely define what system resources individual applications can access, and with what privileges. [Appa] ”

Listing 4.1 AppArmor Policy File

```
/home/user/scripts/aws-deploy.rb {
  #include <abstractions/base>
  #include <abstractions/nameservice>
  #include <abstractions/ruby>

  /home/user/scripts/aws-deploy.rb r,
  /usr/bin/ruby1.9 ix,
  /usr/lib{,32,64}/** mr,
}
```

It originates from SubDomain, which was acquired and rebranded by Novel in 2005. Ubuntu and OpenSUSE integrated it in their default installation, but AppArmor is available for many more distributions.

A detailed administration guide for AppArmor is available [Appb]. The following steps and explanations are loosely based on it.

Installation

AppArmor is available for many distributions. In most cases the installation consists of installing the AppArmor packages and enabling AppArmor. The kernel boot parameter `security` is used to enable any LSM by passing the name of the security module. To enable AppArmor, `security=apparmor` must be added to the list of kernel boot parameters.

The OpenSUSE distribution installs and enables AppArmor by default. In addition there is a YaST module to easily manage AppArmor and its profiles. For those two reasons we will evaluate AppArmor on an OpenSUSE installation.

Policy Creation

Policies are plain text files. They are stored in a special configuration directory. By convention, their name matches the path to the executable. For example, the file at `/etc/apparmor.d/home.user.scripts.aws-deploy.rb` is the policy file for the script at `/home/user/scripts/aws-deploy.rb`.

Policy files can automatically be created by the `aa-genprof` command-line tool or the graphical YaST module. AppArmor is put into learning mode, logging any activity of the application but not denying it. A policy file is created from this log and can be edited further. To edit policy files, a simple text editor can also be used. It is even possible to create policy files from scratch using a text editor, but this can be tedious. The recommended way is to use the learning mode of AppArmor.

A simple policy file is shown in Listing 4.1. It describes what is allowed for the script `/home/user/scripts/aws-deploy.rb`. The “include” statements load rules from other files, which keeps the policy short and easy to write. In addition, only access to the script, the ruby

interpreter (for some reason that is not part of abstractions/ruby), and all libraries on both architectures is given. Each character after the paths stands for one permission, `r` for reading, `ix` for execute and inherit the current profile, `m` for allowing memory mapping.

Policy Installation

To actually enable the policy, it must be loaded into the kernel. This will be done automatically if the command-line or graphical tool is used to edit a policy. In case the policy file was edited manually or copied from somewhere else, all policies must be reloaded by issuing the command `rcapparmor restart`.

Supported Restriction Options

Access to files and directories can be restricted to read, write, and execute access. When granting the execute permission it can be specified how the executed application should be confined. Paths can be expressed using wildcards thus allowing to combine similar rules. This also improves readability of the policy.

Network access can be restricted based on the network protocol but not based on the target address.

4.3.2 SELinux

SELinux has originally been developed by the NSA to enforce confidentiality and access restrictions within their organization. Applications are in domains and are restricted by what is allowed within that domain. Every object (application, file, network port, etc.) in the system must be labeled. Based on these labels, policies specify what is allowed and what not [Run04]. Over time, SELinux has become part of standard Linux distributions and can be regarded as the most comprehensive solution to enforce security on Unix-based systems.

Installation

SELinux is available for many distributions and can be installed via a package manager. As described in the AppArmor installation section, this LSM must also be enabled by adjusting the kernel boot parameter.

Listing 4.2 Simple SELinux Module

```
policy_module(aws_deploy,1.0.1)

########################################
#
# Declarations
#

type deploy_script_t;
type deploy_proc_t;

require {
        type kernel_t;
}

#============= transitions ==============
type_transition kernel_t deploy_script_t : process deploy_proc_t;

#============= ROLES ==============
role system_r types deploy_proc_t;
```

Policy Creation

The concepts in SELinux are quite thorough and complicated. Every subject (processes, etc.) and object (files, ports, etc.) has to be labeled. Based on these labels, rules are created that allow subjects with a certain label to perform an action on objects with a certain label.

A learning mode is available by changing to permissive mode and then using audit2allow to parse the logged errors. In permissive mode, access violations are only logged but the application is allowed to undertake them normally.

First, a simple module that introduces new types for our script was created. This is shown in Listing 4.2 The transition enforces the executed script to be run under the `deploy_proc_t` type. Since nothing is allowed for this type the audit process gave us a complete policy that exactly allows what our scripts needs to do. The result is shown in Listing A.1 and is quite extensive.

Unfortunately this policy did not compile when we tried to load it. The error was due to a "noallow" rule defined elsewhere. These types of rules forbid certain allow rules to be written to prevent writing bad rules. Also, looking at the policy, it looks by no way specific to limit the Ruby script to only access Ruby libraries. As an example, consider this rule:

```
allow deploy_proc_t file:t:file {
  execute getattr read open ioctl execute_no_trans
};
```

It allows reading any standard file. The argument made by SELinux is that it all comes down to correct labeling of objects. Remember that SELinux does not use path names but labels to determine which actions are allowed. Our goal is to confine a single script. Having to label

all objects in the whole system is too much overhead to confine a single script. This makes SELinux the least best option to choose for our use case.

Policy Installation

Typically, changes to the one policy present in the kernel are kept in modules. These modules must first be compiled and then inserted into the loaded policy. This is done via the command `semodule -i my_module.pp`. It ran for a long time, 4-5 minutes on two very different machines. The duration seemed to be unrelated to CPU or disk drive speed. Neither was utilized in a considerable amount. A bug report[2] for Debian covers this issue and apparently has been resolved in the meantime.

Supported Restriction Options

There is a wide range of actions that SELinux can allow or deny. Ranging from the usual read, write, and execution of files to binding to ports and accessing file attributes. To restrict network access, packets have to be labeled as well. This is done with the help of iptables for which rules need to be written that label packets as desired.

4.3.3 Tomoyo

Tomoyo offers this self-description on its website:

> " TOMOYO Linux is a Mandatory Access Control (MAC) implementation for Linux that can be used to increase the security of a system, while also being useful purely as a system analysis tool. (...)
>
> TOMOYO Linux focuses on the behavior of a system. (...) TOMOYO Linux allows each process to declare behaviors and resources needed to achieve their purpose. When protection is enabled, TOMOYO Linux acts like an operation watchdog, restricting each process to only the behaviors and resources allowed by the administrator. [Tomc] "

In Mandriva Linux 2010.0, Tomoyo has replaced AppArmor as the default LSM [Toma]. A very detailed and illustrated documentation about all aspects of Tomoyo is available on their website [Tomb]. The following steps and explanations are loosely based on it.

[2] `http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=724999`

Listing 4.3 Tomoyo Exception Policy

```
initialize_domain /home/user/scripts/aws-deploy.rb from any
```

Installation

Tomoyo is also available for several distributions and can be installed using a package manager. It must be enabled in the same way as the previous LSMs by adjusting the kernel boot parameter.

Policy Creation

Similar to AppArmor, the policies are plain text files, which are loaded into the kernel. There are several configuration files, but we only needed `exception_policy.conf` and `domain_policy.conf`. The command-line tool `tomoyo-editpolicy` can be used to edit stored configuration files or the policy in the kernel.

A process is identified not only by its executable but by its execution history. This means, if `/usr/bin/ls` is executed by `/usr/bin/bash` it can have a different policy than if it was executed by `/usr/bin/ruby`. The execution history, when executed by the Ruby interpreter, is `<kernel> /usr/bin/ruby /usr/bin/ls`. It is very common that applications are launched by other applications to only perform a very specific task. The execution history allows to confine launched applications to exactly what they are supposed to do.

In some cases, the execution history does not matter. For example, no matter by whom the TOSCA container is run, the confinement of its launched scripts must always be the same, i.e. the domain must always be the same. Using the `initialize_domain` directive an application can be put directly under `<kernel>`. This is done for the script itself, shown in Listing 4.3. The `from any` part states that it does not matter from where the application is executed, it will always have its new domain initialized. Starting from this domain, the execution history can be used to confine launched applications.

Tomoyo has a learning mode, which is basically the same as the other learning modes. The learned rules are only stored in the kernel and must be saved to files afterwards. As with the other learning modes, these rules are very specific and thus verbose. A domain policy for our example is shown in Listing 4.4. It was simplified using wildcard expressions. Tomoyo supports many types of wildcards that can be used in paths. This can be less secure but has reasons to exist. Temporary files are often created with random names. To allow access to them, wildcards must be used. It also greatly shortens the policy, which helps auditing it for correctness.

This Listing contains rules for two domains. Each block begins with the execution history, which always starts with `<kernel>`. The first block is the actual script file, which causes the Ruby interpreter to be started. The Ruby executable is another domain. The execution history must be used since these rules only apply to the interpreter, if it was run to execute this script.

Listing 4.4 Tomoyo Domain Policy

```
<kernel> /home/user/scripts/aws-deploy.rb
use_profile 3
use_group 0

misc env \*

file read /usr/bin/env
file execute /usr/bin/ruby exec.realpath="/usr/bin/ruby1.9" exec.argv[0]="ruby"
file read /usr/lib/locale/\*/\*
file read /usr/lib/locale/de_DE.utf8/LC_MESSAGES/SYS_LC_MESSAGES

<kernel> /home/user/scripts/aws-deploy.rb /usr/bin/ruby
use_profile 3
use_group 0

misc env \*
file read /dev/urandom
file read /home/user/scripts/aws-deploy.rb
network unix stream connect /var/run/nscd/socket
network inet stream connect 192.168.0.207 35357
file read /usr/lib/ruby/1.9.1/\{\*\}/
file read /usr/lib/ruby/gems/1.9.1/gems/\{\*\}/
file read /usr/lib/ruby/gems/1.9.1/specifications/\*
file read /usr/lib/locale/de_DE.utf8/LC_CTYPE
file read /usr/lib/ruby/1.9.1/\*
file read /usr/lib/ruby/1.9.1/\{\*\}/\*
file read /usr/lib/ruby/gems/1.9.1/gems/\{\*\}/\*
```

The following two lines indicate which profile to apply to the domain and which group to use. A profile decides how Tomoyo behaves. There is a profile telling Tomoyo to just allow everything the application wants to do. Another profile, the learning mode, has already been presented. The most important profile is number 3, the enforcing profile. It will deny operations not allowed by the policy and log these violations.
A group is a set of rules. If applications share common rules these are put into a group. In the domain, the group is referenced and all its rules apply. This greatly shortens policy files for similar applications. The current implementation of Tomoyo has one downside regarding groups, though. Just one group can be used by a domain at the same time.

Each of the following lines is a rule that grants access. Access must be granted explicitly or it will be denied. Most of the rules in a Tomoyo policy are self-explanatory.

Policy Installation

Policies can be installed from any location using the `tomoyo-loadpolicy` command-line tool. These are only present until the next reboot. A configuration to load during boot can be stored in `/etc/tomoyo/`.

Supported Restriction Options

The Tomoyo documentation has a complete list of all supported directives in the domain policy. To summarize, it is possible to restrict:

- File operations like read, write, execute, create, chown, symlink, etc.
- Access to environment variables
- Network and Unix socket operations by address and port.

4.4 Consequences

AppArmor, while being relatively simple to use, does not provide well enough means to limit network access. Only in combination with iptables we could achieve our intended level of confinement.

SELinux is complex to use and create policies for. It is capable of providing our intended level of confinement, but at a very high cost of maintainability. Especially the requirement to label all objects prevents SELinux from being usable here. It would require some agreement how to label operating system files, libraries, ports, etc. This is too much overhead to confine simple artifacts.

By the process of elimination this leaves Tomoyo. But Tomoyo actually does offer exactly what is needed. It is easy to install, easy to write policies for, and still allows fine-grained control over what applications can do locally and over the network. For that reason, Tomoyo was selected to confine the execution of artifacts.

4.4.1 Path-based vs. Label-based

SELinux gives each file a label to define its security context, AppArmor and Tomoyo simply use the path of the file. This is often criticized by SELinux supporters as not being mandatory access control, even being a security issue [Lei06]. The issue arises when a simple symbolic or hard link from a forbidden file is created to a file to which access is allowed. Access is then granted to the file to which access was originally forbidden.

Tomoyo has a web page[3] explaining in detail how this got solved. To summarize, Tomoyo prevents the application from creating such links in the first place.

[3] http://tomoyo.sourceforge.jp/wiki-e/?cmd=read&page=WhatIs&p=0#g6a56098

4.4.2 Unresolved Issues

There is one drawback all LSMs have. To limit network operations, IP addresses must be given, domain names cannot be used. This is unavoidable, because network connections are always established to IP addresses, not to domain names. But domain names are used extensively today, which makes writing policies difficult. It is necessary to obtain all IP addresses the script might connect to beforehand. In case of load balancing mechanisms which return different IPs for every DNS request this is very difficult. If the policy is created long before it is used, the obtained IP addresses might even be outdated.

The policy writer must be allowed to use domain names. These can be resolved to IP addresses right before the policy is loaded. In most cases this will work. Details on how this was implemented can be found in Section 5.3.4.

4.5 Implementation

As shown above, Tomoyo is a simple and feature rich solution to confine scripts. A simple framework for running scripts in a confined environment was developed. The confinement is described in a policy file. The framework itself is a single Ruby script, called confinement controller, which requires Tomoyo to be available.

The basic functionality is very simple:

1. Put the script and complementary files into place.
2. Adjust paths in the policy for the script.
3. Load the policy into the kernel.
4. Run the script.
5. Unload the policy from the kernel.

The first step must actually be performed by the caller of the confinement controller, only he knows which files need to be present. The directory which contains all these files and the name of the script to execute must be passed to the confinement controller. Each of the following steps is then performed by the confinement controller.

These individual steps are first explained in detail in the following subsections. The last subsection puts these steps together into a single executable script.

4.5.1 Adjust Policy

A domain policy consists of blocks, each beginning with `<kernel>` followed by the execution history. All paths in the execution history must be absolute so Tomoyo knows which file they refer to. Similarly, the policy file might contain references to other files, again via absolute paths. The creator of the policy file would need to know where the files will be placed. This can be done by convention or by determining the paths before loading the policy. We chose the latter because it allows for much more flexibility.

A placeholder string was used to identify the path where the scripts are stored. This placeholder is later replaced by the actual path before the policy is loaded. Ideally, the placeholder is a sequence of characters that usually does not occur in a policy file. We chose `$SCRIPTDIR` for this. It looks like a variable name and can only occur in a policy file, if a directory or file contains this in its name, which was deemed very unlikely.

```
SCRIPTDIR=‘pwd‘
sed -i "s~\$SCRIPTDIR~$SCRIPTDIR~g" domain_policy.conf > modified_domain_policy.conf
```

It should be noted that this gives the possibility to inject arbitrary rules into the policy. Simply append a newline followed by a few rules and a comment sign and complete is the code injection. Therefore, the content of `$SCRIPTDIR` must come from a trusted source, i.e. the confinement controller itself.

4.5.2 Load Policy

The updated policy file must be loaded into the kernel to be active. In addition, the exception policy must be updated to initialize a new domain for our script. As described above, one lines must be added to move the script into a new domain. The full path to the script that is executed must be used. This path is known to the confinement controller since it later needs to execute that file. In our case this path consists of the path to the directory containing the script and the actual name of the script.

The exception policy is created and loaded by this command:

```
echo "initialize_domain $SCRIPTDIR/$SCRIPTFILE from any" | tomoyo-loadpolicy -e
```

4.5.3 Run Script

The actual execution of the script is very simple. It is run like any other application. This can require to pass command-line arguments and environment variables on to the script as specified by the caller of the confinement controller.

Listing 4.5 Confinement Controller Script

```
#!/bin/bash

# ARGV[1] the script's root directory
# ARGV[2] the relative path to the script to run
# ARGV[3..] are the arguments for the script
# environment variables are passed through

cd $ARGV[1]

# Adjust policy
sed "s~\$SCRITDIR~$ARGV[1]~g" domain_policy.conf > modified_domain_policy.conf

# Load policy
tomoyo-loadpolicy -d < modified_domain_policy.conf
echo "initialize_domain $ARGV[1]/$ARGV[2] from any" | tomoyo-loadpolicy -e

# Run script
./$ARGV[2] ${@:3}

# Unload policy
echo "delete initialize_domain $ARGV[1]/$ARGV[2] from any" | tomoyo-loadpolicy -e
grep "^<kernel>" modified_domain_policy.conf | sed "s/^/delete /" | tomoyo-loadpolicy -d
```

4.5.4 Unload Policy

Adding a policy file again and again can increase the memory footprint and cause conflicts with previous rules, if an updates policy file is used. This requires us to unload the policy for the script after its execution finished.

The documentation of Tomoyo somewhat hides the possibility to unload a policy. To delete an exception policy rule, simply prepend the keyword `delete` to the rule and add that using the `tomoyo-loadpolicy` tool. The exception rule is then deleted. Similarly, complete domains can be unloaded by loading a line consisting of the `delete` keyword followed by the execution history of the domain. As an example, `delete <kernel> /bin/myscript.sh` removes the domain `<kernel> /bin/myscript.sh` and all associated rules.

4.5.5 The Complete Confinement Controller

Tomoyo must be installed on the system that runs the confined artifact. This can easily be achieved through the distribution's own package manager. Some additional setup steps might be necessary as described in Section 4.3.3.

To run a script from the artifact in a confined manner, only a very simple wrapper script was necessary. This wrapper script, the complete confinement controller, is shown in Listing 4.5.

5 Integration

At this point, we had three things: A concept how to describe infrastructure resources through generic properties, some code to provision the resources, and a prototype to safely execute this code. The next step was to combine these with the TOSCA standard and integrate them into a CSAR. In simple terms, a CSAR is an archive which ties together a description of resources to be deployed with the artifacts required for the deployment.

For all further development, the standard SugarCRM example introduced in the TOSCA primer [TOSb] was used. This example had to be modified based on the findings from Section 3.3. The modifications and newly introduced elements are described in Section 5.1.

Getting from a CSAR to a set of deployed resources is a two step process. First, the user has to select a provider for each resource. Based on the selected provider more information is requested, for example authentication information. This additional information and the artifacts that interface with the provider are put into the CSAR as other information and artifacts are already present in it. The tool performing this step is called enricher, because it enriches the CSAR with more information and artifacts. It is described in Section 5.3.

The second step is to deploy the resources describe in the enriched CSAR. This is done by another tool which executes artifacts associated with resources. These artifacts are executed in a confined environment according to the policy associated with them. Since this tools executes artifacts, its name is executor. It is described in Section 5.5.

To test the whole process, a simple test script was developed. It uses `expect`[1] to automate the interactive enricher and calls the executor with the generated CSAR. This results in a single script to test the whole process for different providers.

5.1 Modifications to TOSCA

This sections explains not only modifications to the actual standard, but also modifications to the SugarCRM example CSAR[2] and to the implementer's recommendation [LS] for invoking bash script artifacts. Each following subsection reflects a logical group of modifications.

In the course of developing the enricher and the executor a few modifications to TOSCA were required to make it work. These modifications are minor, but they still touch what is specified by the standard or proposed in other documents. Each following subsection describes one

[1] `http://expect.sourceforge.net/`
[2] `https://www.oasis-open.org/committees/document.php?document_id=50158`

modification. Most of them do not touch the actual standard, and where they do changes are minimal.

5.1.1 Artifact Input

The TOSCA standard does not specify how data is passed to artifacts that are being run. The implementer's recommendation does for invoking bash scripts. It suggests that each property of the NodeTemplate or RelationshipTemplate is passed as an environment variable. For this to work, the properties must have a flat structure, i.e. no attributes and nested elements. This is a very simple and limited method.

The artifacts developed as part of this thesis are mostly written in Ruby and the properties are more complex - they make use of attributes and nested elements. For these two reasons, the recommendation was not followed regarding how to pass data to scripts. Instead, a new way was developed.

Ruby Script Input

The recommendation for bash scripts suggests an ArtifactType called ScriptArtifact, which has a property called ScriptLanguage. Depending on the script language, different bindings can be used. These bindings define how artifacts are being called. The recommendation only described such a binding for shell scripts. A new binding for Ruby scripts was developed and is described here.

The binding is actually very simple. In case of an artifact for a NodeTemplate, the complete XML structure of the NodeTemplate is passed into the script via the standard input channel. The artifact then utilizes an XML library part of the Ruby installation to process the properties. In case of an artifact for a RelationshipTemplate, a new XML document is build. The root element "Root" contains three child elements. The first is called "Source" and contains the complete XML of the source NodeTemplate. The second is called "Target" and contains the complete XML of the target NodeTemplate. The third is the complete XML of the RelationshipTemplate itself.

This gives the called script access to all properties and even more information, which might come in handy. One such additional piece of information is the name of a NodeTemplate, which can directly be read and used.

Shell Script Input

The goal for shell scripts is to pass the same complete XML structure, as is passed to Ruby scripts. This is achieved by flattening the structure into a set of environment variables. For XML attributes, the environment variable name is the name of the attribute prepended by the element name of all parents, each separated by an underscore. For XML elements which only contain text, the environment variable name is the name of the element prepended by

Listing 5.1 XML to be Transformed to Environment Variables

```
<NodeTemplate name="My Server" id="s1">
  <Properties>
    <ServerProperties>
      <Memory desired="2"/>
      <Description>my first server</Description>
    </ServerProperties>
    <AuthenticationProperties>
      <Username>hans</Username>
    </AuthenticationProperties>
  </Properties>
</NodeTemplate>
```

Listing 5.2 Environment Variables Generated from XML

```
NodeTemplate_name="My Server"
NodeTemplate_id="s1"
NodeTemplate_Properties_ServerProperties_Memory_desired="2"
NodeTemplate_Properties_ServerProperties_Description="my first server"
NodeTemplate_Properties_AuthenticationProperties_Username="hans"
```

the name of all parent elements, again separated by an underscore. The environment variable value is the content of the attribute or element. Namespaces are ignored. In Listing 5.1 a simple NodeTemplate XML element is shown. Its transformation into environment variables is shown in Listing 5.2.

Obviously, there are some requirements the XML structure must fulfill for this to work. For one, there must not be any conflicts in element names when namespaces are removed. There must also be no attribute with the same name as a child element. Mixed content of XML elements is not supported, meaning an element either only contains child elements or text. Finally, underscores in element and attribute names are forbidden. This is not so much an issue for the input but when the output of the script is parsed (see below) ambiguities must be prevented.

The suggested binding still has its limitations, but they are far less. At least for this thesis the binding allowed for all that was required.

5.1.2 Artifact Output

Artifacts not only need input but they also produce output which might become input for other artifacts. For example, when deploying a virtual machine and a storage device, the artifact to connect the two needs to know the identifiers of both resources. Using the standard output channel, such output can be passed back to the caller. If the script needs to communicate warnings, errors, or other information, it can use the standard error channel and thus follows Unix conventions.

Listing 5.3 Demonstration of New Shell Binding

```
$ export NodeTemplate_name="My Server"
$ export NodeTemplate_Properties_ServerProperties_Description="my first server"
$ ...
$ ./my_script.sh
NodeTemplate_Properties_ServerProperties_Description="my first server deployed"
...
```

Ruby Script Output

The input to Ruby scripts is passed as an XML string. The most obvious way to return data is for the Ruby script to output an XML string again. This XML string is the input with modifications done by the script. The caller can then simply take this XML string and replace the original document with the new one. But this would allow the script to modify any information of the NodeTemplate, including its ID. This is deemed unnecessary and more likely to cause problems than to solve them. Therefore, only the properties element is taken from the output of the script.

The input for scripts used with RelationshipTemplates consists of three parts, the output will therefore too. Although not used in this thesis, this output must be parsed as well. Analog to the previous paragraph, modifications to the properties are copied. The only difference is that three different groups of properties must be copied. The properties of the source, the properties of the target, and the properties of the relationship itself.

Shell Script Output

The previously mentioned implementer recommendation states that “return parameters may have been written to environment variables by the script itself” [LS]. But environment variables are always propagated downwards to sub-processes, never back to the calling process. There is only one possibility to see changes of environment variables when calling shell scripts. The caller himself is a shell, and the script is not run in a subshell but sourced by the caller. This is far from ideal since statements like “exit” in the script will terminate the caller without him being able to evaluate the modified environment variables.

To be able to get output from shell scripts, a similar approach to before is suggested. Using the output channel, the script returns changes in a format close to the input. To illustrate the communication with the script a manual invocation of such a script is depicted in Listing 5.3. The output consists of key value pairs, which directly reflect the environment variables. For every pair that is being output the caller modifies the source XML element accordingly. Looking at the previous Listing, the description property of the NodeTemplate must be changed.

It may be the case that a key is output for which there is no XML element yet. In that case the caller ignores the key. No new elements should be created because the namespace for those is not known.

Listing 5.4 XML Schema for the ExecutionLocation Property

```
<xs:element name="ExecutionLocation" default="target"/>
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="management"/>
      <xs:enumeration value="target"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

5.1.3 Execution Location of Artifacts

ImplementationArtifacts must be executed somewhere. This can be on the targeted machine or they can be executed in the management environment. The latter is necessary for infrastructure resources because no target machine is present yet.

The TOSCA standard provides no means to define where an artifact is to be run. Therefore a new property is added to ArtifactTemplate called ExecutionLocation. The XML schema is shown in Listing 5.4. The ExecutionLocation can be “management” or “target”. “Management” meaning the artifact is to be run on the management side. “Target” meaning the artifact is to be run on the targeted resource, typically a virtual machine.

5.1.4 Properties of Server NodeType

The SugarCRM example suggests two properties for virtual machines (called servers): Number of CPUs (either 1, 2, or 4) and Size of the Memory. This only allows very narrow specification of the required features of a virtual machine. It does not fit to the very broad possibilities discovered for all the different Cloud providers in Section 3.3.2. This subsection describes how the generic properties are best reflected as properties for the “server” NodeType.

A shortened XML schema of the ServerProperties element is shown in Listing 5.5. Memory, Disk, and number of CPUs are required elements for which a range and a desired value can be given. The resource which is created must have its value between min and max and should be as close to the desired value as possible. The name of an Image, the Hostname and the Description are optional because not all providers need them.

Even though the property DisplayName has been mentioned in Section 3.3.2, it is not mentioned here. Instead of using a property for the name, the name of the NodeTemplate itself should be used. This applies to volumes and security groups as well, which follow now.

5.1.5 Block Storage

For block storage devices, a new NodeType and accompanying properties are suggested. The NodeType is called BlockStorage and shown in Listing A.2. It is derived from the RootNodeType and has a PropertyDefinition for one element called BlockStorageProperties.

Listing 5.5 XML Schema for the ServerProperties Type

```
<xs:complexType name="tServerProperties">
  <xs:sequence>
    <xs:element name="Memory" type="xs:int">
      <xs:complexType>
        <xs:attribute name="min" type="xs:positiveInteger"/>
        <xs:attribute name="max" type="xs:positiveInteger"/>
        <xs:attribute name="desired" type="xs:positiveInteger" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="Disk" type="xs:int">
    ... same as Memory ...
    </xs:element>
    <xs:element name="Cpus" type="xs:int">
    ... same as Memory ...
    </xs:element>
    <xs:element name="Image" type="xs:string" minOccurs="0"/>
    <xs:element name="Hostname" type="xs:string" minOccurs="0"/>
    <xs:element name="Description" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

This property element is designed similar to the ServerProperties element. It is shown in Listing A.3. The disk size is mandatory, it is a range with a desired value. All other properties are not always necessary, but most providers need some of them. Therefore Snapshot, Description and Offering are optional.

The name is also listed as a generic property in Chapter 3. But instead of using an additional property the name of the NodeTemplate can be used.

Next to the NodeType, a new RelationshipType to connect block storage to virtual machines is suggested. This RelationshipType is called AttachStorage whose source is a Server NodeType and target is a BlockStorage NodeType. A new interface is also introduced here for attaching storage to a virtual machine. It is called `http://www.example.com/thesis/Interfaces/AttachTo` with a single operation called attachTo.

5.1.6 Security Groups

The SugarCRM example is not covering security groups. Therefore a suggestion on how to model these is made here.

Security Groups are different from other types of infrastructure resources, they have no physical equivalent. Is it even appropriate to model them as infrastructure resources? Instead of being a resource they could be properties of virtual machines, describing what traffic can go into and out of the machine. But then the definition of a single security group might be spread over multiple virtual machines. These rules need to be merged, complex logic is necessary to not create duplicate security groups and it is an open question where to store the artifacts

for creating and modifying security groups. Security groups are created and destroyed using artifacts in the same way as any other resource. They are connected to other types of resources, meaning these are part of the security group. All of this comes naturally, when security groups are treated as infrastructure resources.

A new NodeType, shown in Listing A.4, describes security groups. The properties element is shown in Listing A.5, the XML schema for the rules type is shown in Listing A.6. Rules associated with a security group are described in the properties associated with the NodeType. The difficulties in finding a common set of rules that work for all providers have been described in Section 3.3.6. Each rule is represented as an XML element and contains a protocol name and a port range. These define which type of incoming traffic is allowed.

Analog to the BlockStorage NodeType this one also needs a RelationshipType. It is called JoinSecurityGroup, the source is a Server and the target is a SecurityGroup. The interface is the same as for block storage for ease of implementation of the prototype.

5.1.7 Extended SugarCRM Example

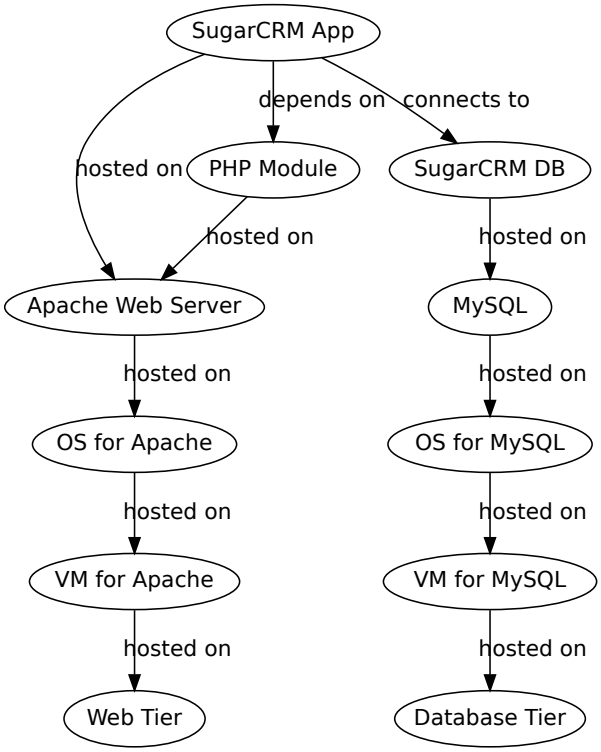

Figure 5.1: Topology of the Original SugarCRM CSAR

The topology of the existing SugarCRM example for TOSCA is shown in Figure 5.1. It describes a very basic topology of two virtual machines hosting Apache and MySQL for the

SugarCRM web application. This example CSAR had to be extended for this thesis. The XML schemas for properties shown before and the new NodeTypes were added. To be able to use them some NodeTemplates were added as well. Finally, the existing NodeTemplates were changed to fit the new schemas and to have better fitting values.

New NodeTemplates were added for a block storage devices that is being attached to the MySQL server, and a security group is created for both servers. This gave us an example CSAR to work with, which utilizes different types of infrastructure resources.

The security group allows incoming traffic on TCP port 80 to both servers. The storage for MySQL must be at least 1 GB, desired are 2 GB but it may go as high as 100 GB. This was necessary to allow deployment at BareMetalCloud where the smallest target has 100 GB.

The Apache server memory must at least be 64 MB, 512 MB are desired, and it may go as high as 1 GB. For MySQL the server memory must at least be 64 MB, but 1 GB is desired. There is no upper limit. Both servers desire one CPU and no disk, but Apache at least allows a disk to be present.

The complete topology of the updated SugarCRM example's service template is shown in Figure 5.2. The nodes shaded in gray are infrastructure resources. These are treated by the tools developed as part of this thesis, the other nodes are ignored.

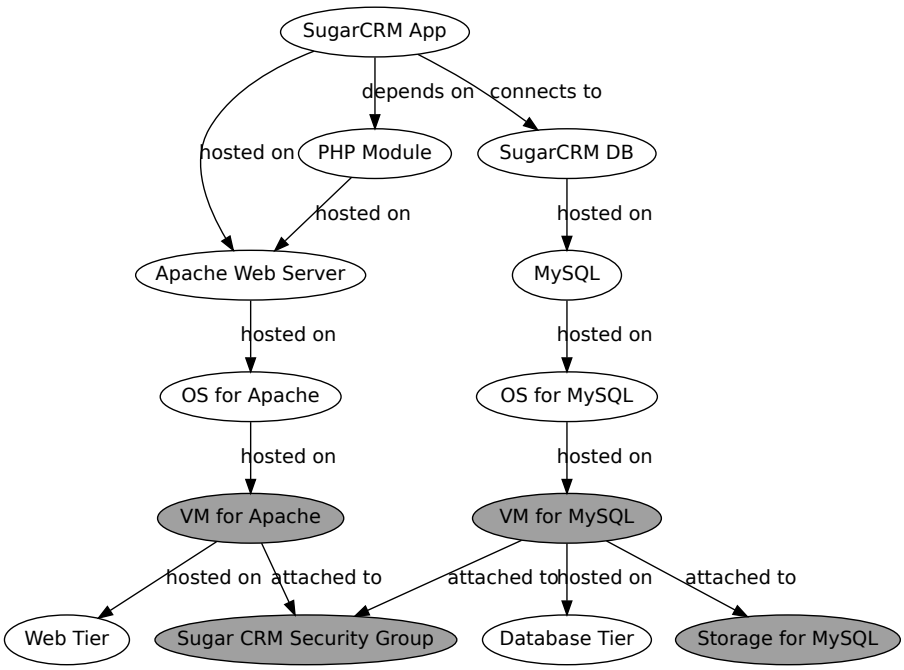

Figure 5.2: Topology of the Modified SugarCRM CSAR

5.2 Ruby Module for TOSCA

Both, the enricher and the executor, need to read a CSAR, interpret the XML files, find all templates, types, etc. This common logic for processing a CSAR and handling TOSCA elements was put in its own module called "TOSCA". The important parts of the module are explained in this section.

5.2.1 IDs and References

All elements of the TOSCA standard have an attribute called ID (sometimes also called name) to identify the element. To prevent name clashes, namespaces are used leading to fully qualified attribute values. This means an attribute value can be prepended with a namespace prefix. These values must thus be interpreted correctly.

The correct way to interpret these values is to use the namespace associated with the prefix, if one is used. If none is used not the default namespace but the targetNamespace must be used. That must be set for the Definitions XML element but can also be set for others.

A class in the TOSCA module called `ID` represents such an ID with its namespace and its value. It retrieves the correct namespace and value for an ID or a reference to an ID from the attribute of an XML element. It also allows to write an ID back into an attribute value while maintaining the namespace. This means it uses the correct prefix or adds a new namespace for the element.

5.2.2 Loading

The largest part of the module is for loading a CSAR. First, the metadata is scanned for files containing TOSCA Definitions. These must be parse. For each definition the imports are also scanned and added to the list of files to process. When processing a definitions document all NodeTemplates, RelationshipTemplates, NodeTypeImplementations, RelationshipTypeImplementations, and ArtifactTemplates are read and kept in arrays or hashes for later retrieval. After all these elements are read, the NodeTemplates are connect according to the RelationshipTemplates. This creates a graph, which can later be traversed.

The loading step is limited to what is needed for the purposes of this thesis. It does not cover all TOSCA elements and only provides easy access to information needed by the enricher and executor.

5.2.3 Iterate through the Topology

After the CSAR was loaded, there are two ways to access its content. Either via one of the lists of found elements, or using the `iterate_topology` method. This method traverses the graph of NodeTemplates and RelationshipTemplates using a topological sort. To be precise, the order is a bit different than the normal topological sort. A NodeTemplate is processed if the sources of

all of its incoming relationships have been processed. An outgoing relationship is only processed if the target NodeTemplate already has been processed. So, before a RelationshipTemplate is processed, its source and target NodeTemplates must have been processed first. The reason is that the artifact for a RelationshipTemplate can only connect its source and target resource, if both are present.

A Ruby block must be passed to the `iterate_topology` method, which is then called for every NodeTemplate and RelationshipTemplate. Using introspection, it is possible to detect the class and thus if it is a NodeTemplate or RelationshipTemplate. The processing logic of the two will most likely differ.

5.3 Enricher Tool

To reiterate, the enricher takes an existing CSAR and add data and artifacts to provision the infrastructure resources described therein. That CSAR must be a modified CSAR, modified according to what has been described in Section 5.1. The enricher only considers NodeTypes which describe infrastructure resources. These are identified by their names. Other NodeTypes not describing infrastructure resources are left alone by the enricher.

The NodeTemplates in the existing CSAR must already have properties as described in Section 5.1. Based on these generic values specific values for the selected Cloud provider are determined.

The enricher is meant to be run right before the CSAR is given to the executor, i.e. resources are actually being created. Some preparations done by the agents might become invalid if too much time has passed. No examples imposing a hard time limit were found. But the image and flavor are typically derived from the generic properties and might not be available when trying to use them much later.

The work of the enricher can be summarized in simple terms. For every NodeTemplate and RelationshipTemplate the user is queried for the provider agent to use. Depending on the selected provider, more information is queried from the user. Together with artifacts, this information is written back to the CSAR. It is behind the scenes, where a lot of complexity resides in correctly modifying XML documents, saving files in the CSAR without name clashes, etc. The details, that must be done right, are explained in this section.

To keep the complexity low, no graphical interface was developed. The enricher is an interactive command-line application which expects the CSAR as an argument and uses the standard input and output for interaction with the user. It was written in Ruby, the language of choice in this thesis. To be non-destructive, the enricher outputs the modified CSAR with '-enriched' appended to the filename. The input CSAR must be valid, no validation is performed regarding the correctness of the input CSAR.

Support for different providers was achieved through a plugin system. A single plugin represents a single provider and is called provider agent. Within a provider agent, there are multiple plugins again, each for different NodeTypes and RelationshipTypes. Since they map to resource

types, these plugins are called resource agents. The term plugin is reserved for the context of actual plugin logic. In all other cases the term provider agent or resource agent is used, to avoid confusion about which type of plugin is meant.

To summarize, a CSAR and the agents are input to the enricher, the modified CSAR is the output. This is depicted in Figure 5.3.

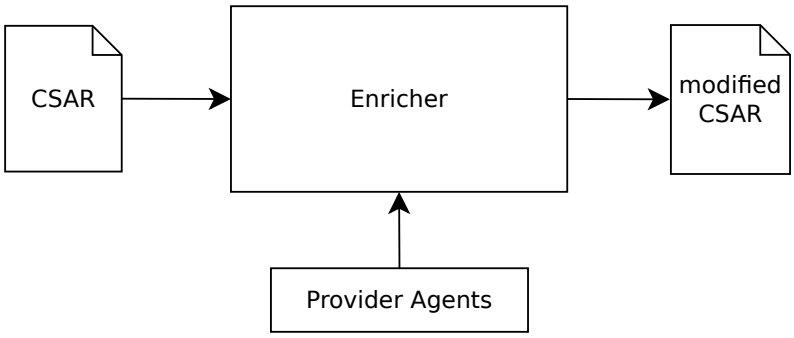

Figure 5.3: Enricher Overview

The following subsections explain all aspects of the enricher. At first, an overview of the architecture is given. After that, the detailed explanations begin with the agents and move up to the enricher's main script.

5.3.1 Overall Architecture

The architecture of the enricher is shown in Figure 5.4. A special notation has been utilized to be able to show all the important elements that make up the architecture. Every element is shown as a rectangle with a type, which is shown in the upper right corner between angle brackets. A type of "code" denotes Ruby code in the form of a class, mixin or script. A type of "xsd" denotes an XML schema. A type of "directory" or "file" denotes a directory or file. Files can be configuration files, script files or any other kind of file. A type of "tosca" denotes a TOSCA element stored in an XML file. Elements without a type group all contained elements and give them a name. There are three types of dependencies between elements. An element can contain exactly one element of some type ($\rightarrow\!\!\blacktriangleright$), one or more elements of some type ($\blacktriangleright\!\!\blacktriangleright$) or not contain but use another element ($\rightarrow\!\!\triangleright$). If the resource agent is for RelationshipTypes it will contain a RelationshipType and RelationshipTypeImplementation instead of the NodeType and NodeTypeImplementation.

To read from, traverse through and write to the CSAR file, the TOSCA module was utilized. The actual modification of XML files is done by the enricher itself, not by the TOSCA module nor by agents. They merely provide the data to write into the CSAR. Excluded from this are properties, they are directly modified by the agents.

All provider agents must derive from a base class defined by the enricher. This base class also implements the plugin mechanism which was taken from [Rub]. For the enricher to be able to

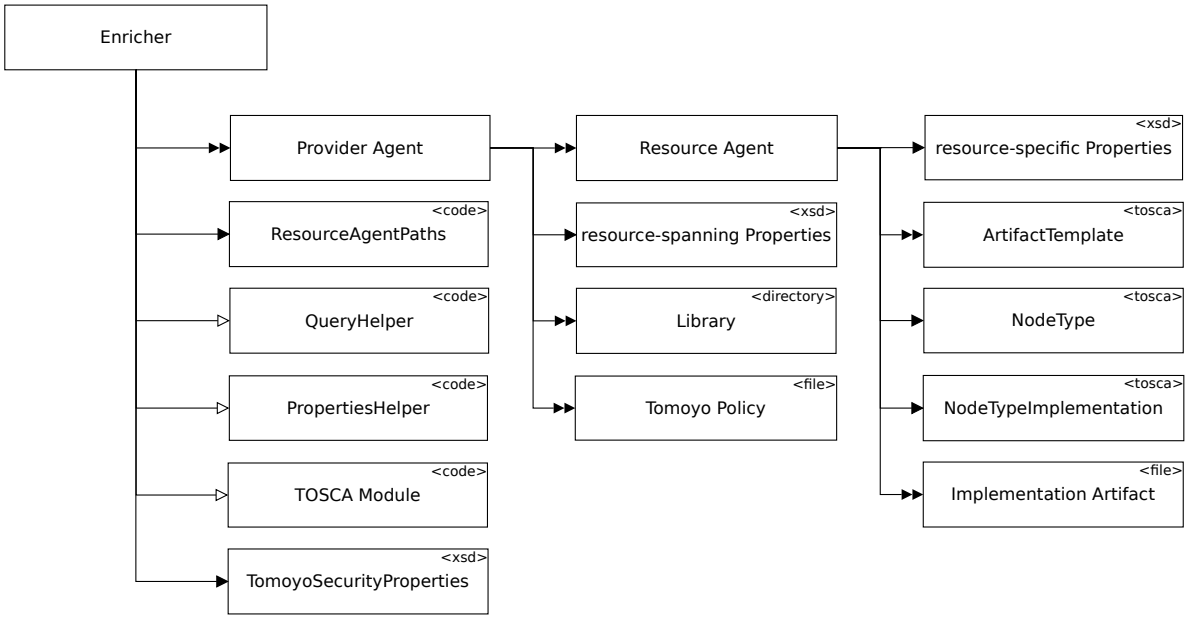

Figure 5.4: Architecture of the Enricher

find and load all agents, they must be named and stored correctly. Details regarding this are explained in Section 5.3.4.

The resource agents, although being similar to plugins, are loaded statically by hard coding their names in the provider agent. They also have no base class because they only live within the provider agent. The enricher does not care about their implementation, only the provider agent must be able to use them.

From a code perspective, the agents are simple Ruby classes. But there are much more files belonging to each agent. These files are the main reason why a separation into provider and resource agent was done. These additional files include XML schemata for properties, policy templates, and artifacts. All files of the agents are explained in the agents' subsection.

Next to the main script and the agents, there are also helper classes. One to help with adding and deleting properties and one to help querying the user.

The above describes the static architecture of the enricher. The following describes the behavior of the enricher and the agents.

After all provider agents are loaded, the enricher loads the CSAR. For every NodeType and RelationshipType, it checks if any provider agent claims to support it. The user is offered a list of these providers from which he can select one. The control flow is then handed off to the provider agent two times. First, to preprocess the NodeTemplate or RelationshipTemplate, then to actually process it. The reason behind this separation is explained in the provider agent's subsection. The provider agent, in turn, determines the resource agent that can handle the NodeTemplate or RelationshipTemplate for preprocessing and processing. Still, the

provider agent has the possibility to perform his own work in the preprocessing and processing stage. Authentication information for the provider (required for every resource type at that provider) is typically queried from the user in the processing stage, before handing it off to the resource agent. The resource agents will also query the user for missing information and then return. When the control flow is returned to the main script, the selected agent has modified some properties and returned some data. Based on the returned data, the main script will then add files to the CSAR or modify existing files.

5.3.2 Helpers

All helpers are implemented as mixins. Their methods can be imported into a class using the `include` statement.

The `PropertiesHelper` offers a few methods implementing common behavior among agents when dealing with TOSCA properties. The first common behavior is to remove unsupported properties and report their removal to the user. The second common behavior is to ensure an optional property exists when it is required by the agent. The helper has two methods for this, which are utilized by the agents.

The `QueryHelper` offers a few methods to query the user for information. This includes a simple method to query a single value from the user, which is then cached. Using this for authentication information prevents the user from having to enter complex passwords multiple times. The `QueryHelper` also includes methods to help with selecting values from a list. One, suitable for small lists, prints a list with numbered entries from which the user enters one number. The other, suitable for larger lists, allows the user to search within the list using a wildcard pattern.

5.3.3 Resource Agents

Resource agents are part of provider agents. They are stored in subdirectories of the provider agent directory. Every resource agent supports one or more NodeTypes or ResourceTypes.

A resource agent subdirectory contains these entries:

- `MyProviderAgent_MyResourceAgent.rb` The main class containing logic for the resource agent.
- `artifact-templates.xml` For each artifact, an ArtifactTemplate element is required by TOSCA. The elements for all artifacts of the agent are within this file.
- `ImplementationArtifacts/` A directory containing the ImplementationArtifacts to actually create, destroy, etc. the resource for this NodeType / RelationshipType.
- `properties.xsd` XML schema for properties specific to this NodeType / RelationshipType.
- `type.xml` The NodeType / RelationshipType element and its Implementation element.

Main Class

Although the implementation of resource agents is completely up to the provider agent, a layout is suggested here which fits well with the rest of the architecture.

The class is named exactly as the file (without the suffix of course) and offers five methods.

The first is the constructor, which accepts an object. This object can carry all information the provider agent wants to pass to the resource agent. In our implementation, this is an instance of the Fog library for which the provider agent has already performed the authentication. The resource agent has thus access to the provider.

Next, two class instance methods (also called static methods) must be present. The method `self.types` returns an array of names of supported types (NodeTypes or RelationshipTypes). Since TOSCA qualifies the name of types with a namespace, this must be reflected in the returned name as well. A common pattern is utilized here which was originally suggested by James Clark[3]. The full namespace URI is written in curly braces before the name of the type. For example: `{http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes}Server`

The other class method is `self.name`, which simply returns the name of the resource agent. This name is appended to the provider agent name to create a unique name for artifacts belonging to this resource agent. All of them will be copied into a directory whose name is the provider agent's name followed by an underscore and the result of this method.

The `preprocess(element)` method takes a NodeTemplate or RelationshipTemplate element and does some preprocessing on its properties. Two main aspects are covered here: reporting and removing unsupported properties and adding required but missing properties. For both, the PropertiesHelper class offers methods to do so. They just need to be called with the correct parameters from the preprocess method.

It might not be obvious why properties should be deleted. This is done to prevent any inconsistencies in the properties. Let's say a server has a hostname property, but the provider does not allow defining a hostname. It would be very misleading if the CSAR contained a hostname which will not be the one the deployed resource will get.

The `preprocess` method must only handle properties that are part of the TOSCA standard. Additional properties are better handled in the next method.

In `process(element)`, the main work of the resource agent is done. In contrast to preprocessing, where just some preparations are being done, this is where data for the actual deployment is queried, calculated, and stored in properties. The user is queried for missing data in a way appropriate for the artifact. Typically the QueryHelper will be used, but it is also possible to implement a separate way of asking the user for information. This gives great flexibility, when it comes to querying the user in the best way possible. Next to querying the user, the resource agent can also use the generic properties to derive provider-specific values. As an example, the best matching image and flavor in an OpenStack environment can be searched for instead of

[3] `http://www.jclark.com/xml/xmlns.htm`

asking the user. The return value of the `process` method is a single properties element or a set of properties elements. These are typically properties specific to the provider and resource agent, while the preprocess step works on generic properties.

The `process` method is called immediately after the `preprocess` method. Technically both methods could be combined into one. Still, the separation was deliberately retained to emphasize the difference in semantics. The `preprocess` method solely works on generic properties and prepares them for the real processing. The `process` method then performs interactive and more complex processing and data preparation.

Figure 5.5 shows both calls to these methods, originating from the enricher. The methods of the `ProviderAgent` have to select the correct `ResourceAgent` to forward the call to and can do preparations and cleanup.

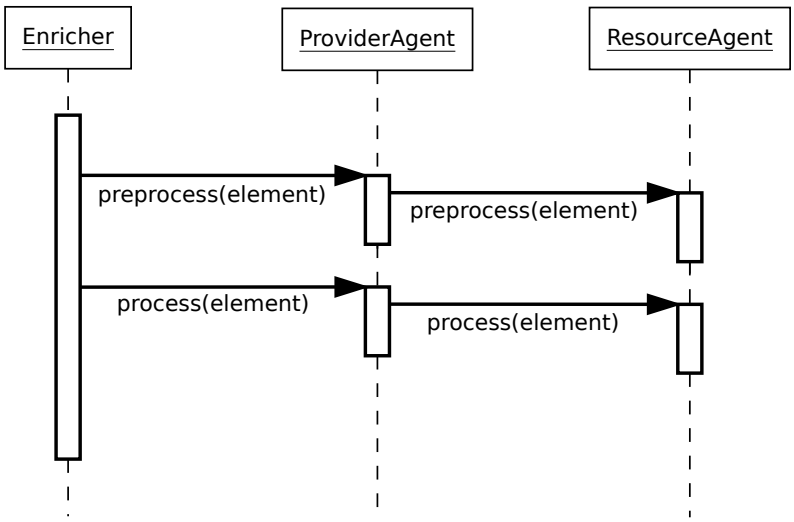

Figure 5.5: Sequence Diagram for Preprocessing and Processing

Artifacts

TOSCA has two different kinds of artifacts. ImplementationArtifacts and deployment artifacts. The former are concerned with implementing the resource, e.g. create and delete it. The latter are deployed into the resource to perform some tasks. Within this thesis, only ImplementationArtifacts were regarded because they are the artifacts being run to manage resources. DeploymentArtifacts are targeted at the managed resource, they are not in the scope of this thesis. The ImplementationArtifacts are stored in the `ImplementationArtifacts/` directory together with auxiliary files. While the organization of the artifacts is not prescribed per se, a specific pattern was used for artifacts created as part of this thesis. There is one artifact for every operation. Its name consists of the operation name followed by "at" followed by the plugin name. In case of artifacts written in Ruby, an auxiliary file called `toscaio.rb` is present. It handles communication between the artifacts and the execution environment via

the standard input and output channels. XML is read from the standard input channel and all properties used by the artifact are made available via getter methods. Where artifacts might change values a setter method is also present. When the artifact finished its job it can tell the `toscaio` class to output the modified XML to the standard output channel.

Every artifact in the CSAR has an ArtifactTemplate XML element which describes it. All ArtifactTemplate elements of the agent are kept in the `artifact-templates.xml` file. The paths to the files will be adjusted by the enricher when copying this file into the CSAR. For details see Section 5.3.5.

New NodeType and RelationshipType

Up until now, a small but important detail has been not been mentioned yet. Artifacts can only be added to a NodeTemplate through its NodeType. The connection from an artifact to a NodeTemplate is shown in Figure 5.6. All NodeTemplates of the same NodeType, e.g. a virtual machine, will use the same artifact.

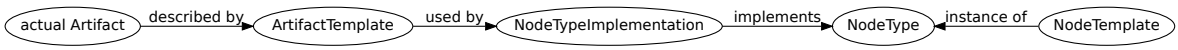

Figure 5.6: Dependency Chain from Artifact to NodeTemplate

This prevents us from deploying virtual machines at different providers, which might be desired. To solve it, the enricher creates a new NodeType for the resource agent. This new NodeType is derived from the original NodeType of the NodeTemplate. For ease of implementation in the prototype, a new NodeType is always created. More intelligent implementations can only do so if actually two providers are required for the same type. For every NodeType, a NodeTypeImplementation must also be created. The ArtifactTemplate, on the other hand, is always the same and shared by all NodeTypeImplementations.

A skeleton for this new NodeType is provided by the `type.xml` file. It contains a NodeType element and its accompanying NodeTypeImplementation. The content of this file is also adjusted by the enricher when copying this file into the CSAR. For details see Section 5.3.5.

5.3.4 Provider Agents

Provider agents reside in their own directory within the `agents/` directory, which is next to the enricher script. Each provider agent is for a single Cloud provider. While in theory a provider agent could support multiple providers, and the same provider could be supported by multiple provider agents, this mapping makes the structure and user interaction easy to understand. It is clear to the user which provider is used when a provider agent is selected. In addition, the provider agent can reuse information if the same provider is used multiple times. For example, the authentication information can be cached such that the user must not enter it again.

The directory of a provider agent contains these entries:

- `MyProviderAgent.rb` The main class that contains the logic of the agent.
- `libraries/` A set of libraries required by the artifacts in this agent.
- `policies/` Templates to create Tomoyo policy files for the artifacts.
- `properties.xsd` XML schema for properties common among all NodeTypes and RelationshipTypes. Most likely authentication information.
- Additional subdirectories contain the resource agents for NodeTypes and RelationshipTypes.

Main Class

While the resource agent's main class only follows a specific interface by convention, the interface for a provider agent's main class is prescribed by the enricher. Every provider agent's main class must derive from a base class and implement all methods that raise a `NotImplementedError`. The base class is shown in Listing 5.6. The first four methods can be overwritten by the provider agent. The fifth method must not be overwritten but can be used to add properties to the NodeTemplate. The last method must be ignored, it is used by the main script to get the provider agent for a template.

The first method, `self.display_name`, simply returns a user friendly name for the agent. This name is displayed to the user in the list of providers to choose from.

The other class method, `self.type_names`, returns a list of all NodeType and RelationshipType names which are supported by this agent. That is, all type names supported by the resource agents of the provider agent.

The `preprocess(element)` method is called to preprocess an element, as described in Section 5.3.3. A default implementation is provided which determines the correct resource agent and calls its preprocess method. This will only work if the resource agents follow the structure explained in Section 5.3.3. Otherwise the provider agent must overwrite this method.

The `process(element)` mainly needs to determine the correct resource agent and hand of the element to it. Before doing so, the provider agent can do some work common to all resource agents. The most common work will be to add new properties which contain authentication information. Typically, this information is required by every artifact of resource agents. The main benefit is that the provider agent can remember these values and suggest them the next time. For further processing, the caller needs to know the resource agent that did the processing. Its name is returned by this method, which it gets from the resource agent's method `self.name` (see Section 5.3.3).

The properties common within the provider agent are described in `properties.xsd`. These properties are being created by the provider agent itself and are available to its resource agents.

Listing 5.6 Enricher ProviderAgent Base Class

```
class ProviderAgent
  include Plugin

  include QueryHelper

  def self.display_name
    raise NotImplementedError.new()
  end

  def self.type_names
    raise NotImplementedError.new()
  end

  def preprocess(element)
    ...
  end

  def process(element)
    raise NotImplementedError.new()
  end

  def add_properties(element, properties_element_set)
    ...
  end

  def add_property(propertiesXml, name, value)
    ...
  end

  def self.for_template(element)
    ...
  end
end
```

Policy Templates

The policy templates are referenced in the `artifact-templates.xml` files of the individual resource agents. For every reference to a policy file, the template is run and produces policy files adjusted specifically for the artifact and the properties provided by the user. This allows for very specific confinement. For example, the exact IP of the endpoint to connect to can be added, thereby preventing any other internet access. Details about the way policy templates are evaluated and used can be found in Section 5.3.5.

Libraries

Artifacts of resource agents usually require some library to interact with a provider. In this thesis this is the fog library, but any kind of additional code that needs to be available next to the artifacts is supported by the resource agent. Required libraries reside in the subdirectory

`libraries/`. These can be used by the agents themselves and will also be copied into the CSAR. The libraries could simply be part of the artifact and reside next to it. But this means each resource agent has a copy of the library. Not only within the provider agent but also within the archive, which will cause an explosion of archive size. Therefore, libraries are kept in their own directory and are, first of all, shared within a provider agent. But they can also be shared within a CSAR, if multiple provider agents use the same library. For this to work, a simple schema is used. The library directory name uniquely identifies the exact version of the library. It consists of the name of the library followed by a dash and the version of the library. If such a directory already exists in the CSAR it is assumed that it is the same library in the same version and not copied again. Thereby multiple artifacts from different provider agents can use the same library, if they require the same version of the library.

The fog library used in this thesis has been modified for better support of BareMetalCloud, see Section 3.5. The version number can no longer be used now, instead the fog library directory was called "fog-thesis-modified".

5.3.5 Main Script

Now that the TOSCA module and the agents are known, it all comes together in the main script of the enricher.

To keep track of the paths of all files and directories in the agent and the CSAR, a class called `ResourceAgentPaths` was created. It provides the correct path to each file in the agent and the CSAR. There are also two helper methods called `ensure_file_present` and `ensure_dir_present`, which make sure a file or a directory with its contents is present in the CSAR.

The CSAR to enrich is specified as a command-line argument with the `-c` option. The enricher starts by loading all provider agents it can find in the `agents` subdirectory. After all provider agents are loaded, a copy of the CSAR is created with the name "-enriched" appended to it. All the following modifications are done in that file.

Figure 5.7 shows the sequence of the activities performed by the enricher and its agents. Using the TOSCA module, the enricher goes through all NodeTemplates and RelationshipTemplates in the CSAR. It queries the user which provider agent to use for each of them and then hands off the control flow to that agent. After the agent returns, the main script has all necessary data to update the CSAR. This is the major part of the main script. Each task performed to update the CSAR is explained in sequence now.

Copy Provider Agent's Properties

At first, the XML schema for the properties of the provider are copied into the CSAR using `ensure_file_present`. To prevent name clashes the `properties.xsd` file is prepended with the provider agent's name. The method will do nothing, if the file is already present.

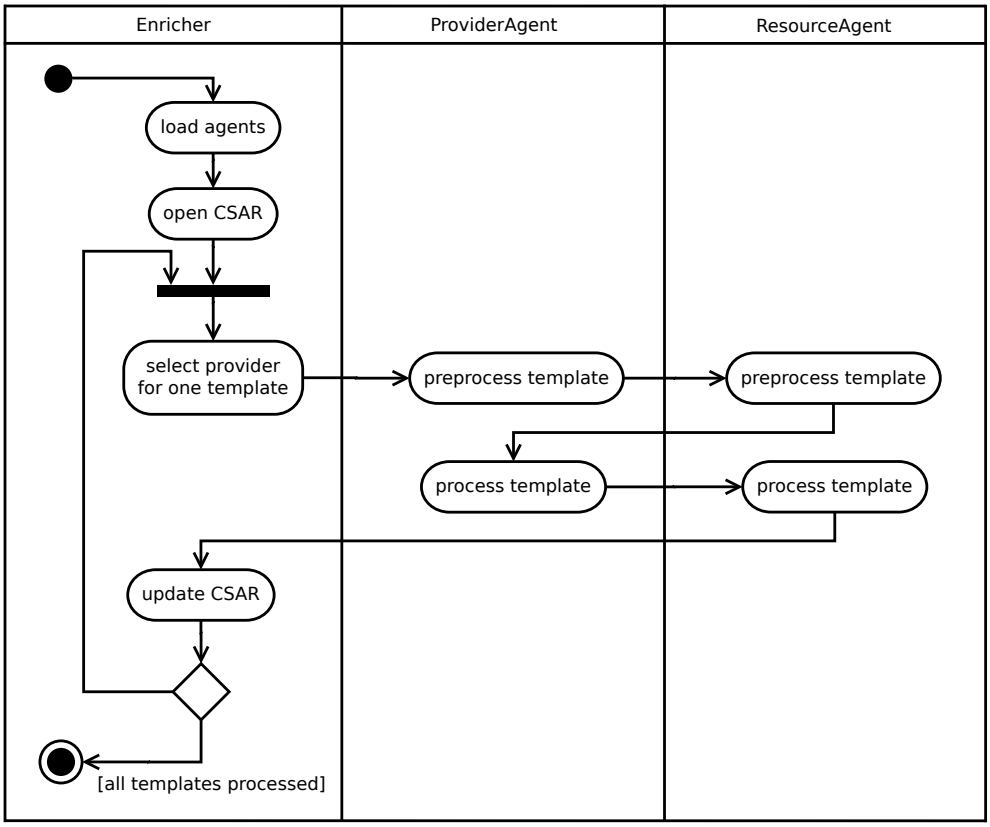

Figure 5.7: Activities Performed by the Enricher

Copy Libraries

All libraries of the provider agent are stored in the `libraries/` subdirectory. As described in Section 5.3.4, it is checked for every library if it needs to be copied into the CSAR. If so, the library's directory must be copied into the CSAR. All these libraries are placed in a directory in the CSAR which is also called `libraries/`. The previously mentioned helper methods are used again to get the correct paths and copy the directory.

Copy Resource Agent's Files

If this is the first time the resource agent processed a template in that CSAR, its files must be copied. To check whether a resource agent was used for the first time it is checked if its `artifact-templates.xml` has already been copied into the CSAR. This file is prefixed by the provider agent name and the resource agent name. It will always be present for resource agents because a resource agent without artifacts can not be used to provision infrastructure resources. What is described in the next paragraph only needs to be done if such a file does not exist in the CSAR.

The `properties.xsd` file can simply be copied. The `artifact-templates.xml` file, which contains all the ArtifactTemplate elements, must be modified. The path to the primary script of our ScriptArtifact and the ArtifactReferences must be adjusted such that the paths are valid in the CSAR. This path is different for each resource agent so no artifacts from other agents conflict with it. After the paths are adjusted in the ArtifactTemplate, the actual artifact files must be copied to that path.

Create Policy Files

This part only needs to be done if the resource agent has not yet processed any template.

At first, policy files were meant to be statically assigned to artifacts within the provider agent. Each artifact would have its own exception policy and domain policy. Similar artifacts could share common parts of their policy, preventing duplicate policy files. This would allow one to easily see what each artifact is allowed to do, independent of any specific configuration.

Unfortunately, this is not possible. Artifacts typically need to connect to some remote service to fulfill their task. With artifacts for services like AWS the address of the remote service is know. But with artifacts for services like OpenStack, the address is not known to the artifact developer. It is provided by the user as input when selecting the artifact.

To maintain strict confinement by only allowing network access to where it is necessary, the policy file must be modified based on user input. The policy is only a template from which a template engine generates the actual policy. This generated policy is then stored in the CSAR. Although finding the best template engine for the job is not the main focus of this thesis, some care had to be taken when one was selected. There is quite a range of template engines for Ruby to choose from[4]. The most important aspects were that no additional gems are required and that the template language is flexible enough to allow unforeseen data transformation and logic. This is necessary when looking at our issue to add an IP. The input provided by the user for OpenStack is a URL. The policy file must contain the IP address, not the DNS name, of the host given in the URL. This requires some processing and lookup. Other artifacts might require data processing and transformation which cannot be foreseen now.

The well known template language ERuby, implemented by ERB, which is part of a default Ruby installation, allows for this flexibility. Ruby code can be embedded into the template allowing to, for example, parse URLs.

For RelationshipTemplates, the policy template has access to the source and target as well. Depending on them, the policy might need to differ. The authentication information (and thus API endpoint) are typically only stored there and not, for a third time, with the relationship.

The policy files for each artifact are read from the TomoyoSecurityProperties element of the ArtifactTemplate. Each policy file is processed by the template engine. The result is saved in a new file for each artifact. The path in the original ArtifactTemplate is then adjusted.

[4] `https://www.ruby-toolbox.com/categories/template_engines`

Modify Template

This step has to be done for every NodeTemplate and RelationshipTemplate.

The currently processed NodeTemplate or RelationshipTemplate needs to be given a new type, as discussed in Section 5.3.3. The name for this type is derived from the current type and the new type introduced by the resource agent. This new name must be set for the template and must be used when adding the type to the CSAR, which is done later. When changing the type of the current template to something else, it must be done with care, because these names are in namespaces. This means that the namespace, if not yet present, must be added to the template's XML document. And if so, a free namespace prefix must be used.

Copy New Type

If this NodeType or RelationshipType is being used for the first time (i.e. this resource agent has run for the first time) its XML file must be copied into the CSAR. This is the file called `type.xml` in the resource agent's subdirectory. It contains the type and its implementation element. The type element must be adjusted. Its name has been determined in the previous step. In addition, the DerivedFrom element of the type must be set to the original type of the NodeTemplate or RelationshipTemplate. The `type.xml` file imports the `properties.xsd` and the `artifact-templates.xml` files. These imports must be adjusted, because the files were renamed. Then, the NodeTypeImplementation (or RelationshipTypeImplementation) must be given a name and the correct type name must be set. Finally, this new file containing the type and its implementation element must also be added to the metadata.

5.4 OpenStack Plugin

The first provider agent was implemented for OpenStack. With DevStack[5] a free and easy to deploy OpenStack environment was available. This allowed cost-free and rapid development of the agent. The agent is not only the first implementation of the above described architecture but should also be a guide to develop provider agents for other providers. This has been done in the evaluation phase, see Chapter 6.

An important implementation detail must be mentioned here. By default, OpenStack instances join the default security group. This was not considered when the provider agent was created and only later found out. Due to the relative complexity of leaving the default security group the behavior was left as it is. For details, see Chapter 8.

[5] `http://devstack.org/`

5.5 Executor Tool

The second tool and final piece to get a CSAR which contains infrastructure resources deployed is the executor. Its purpose is to run artifacts associated with resources from the CSAR. There are tools that offer this and with OpenTOSCA[6] there is even one available in source code. This is a requirement since the confinement of artifacts must be implemented, too. But OpenTOSCA is fairly complex and would have require much more time to become acquainted with than to just write our own tool. In essence, the executor is a TOSCA container with very limited capabilities but it does exactly what we need for evaluation purposes of this thesis.

The parsing of a CSAR was already available in the TOSCA module, so the only thing missing was actually running the script with some input, handling the output, and ensuring confinement. This specifically excludes connecting to any machine, support build plans or be very robust.

The input is a CSAR, which is given by the `-c` command-line option. Using the `-s` the confinement controller can be enabled to securely execute artifacts. This means Tomoyo is utilized to confine the executed artifacts to what is allowed by their policy. When starting the executor script, it will check for Tomoyo and exit if it is unable to access Tomoyo. This is mostly to help debug issues with setting up Tomoyo and configuring the permissions correctly. Using `--artifact-dev`, a mode is enabled which allows to preview the policies before loading them and gives the possibility to do further modifications right before running the script. The `-u` option will cause the property changes of NodeTemplates and RelationshipTemplates to be written back to the CSAR. Full-featured TOSCA containers use a dedicated store for keeping properties set during deployment phase. But the executor is just a prototype tool to deploy a CSAR, which is why this way of operating was deemed sufficient.

The executor was implemented in Ruby as well. It works somewhat similar to the enricher. Using the `iterate_topology` method, the executor iterates over all NodeTemplates and RelationshipTemplates. What is done for each of them is described in chronological order in the next subsections.

5.5.1 Prepare Script Execution

First, the input data for the script is prepared. This is a simple XML document which is later passed to the script according to the binding for the script language.

After that, the implementation of the NodeType or RelationshipType is fetched. The ArtifactTemplate for the operation to execute is looked up from this implementation. If it is a ScriptArtifact, an instance of the `ScriptArtifact` class is created. In any other case or if any of these steps fail, a warning giving the exact reason is shown. The NodeTemplate or RelationshipTemplate is then ignored and the next one is processed.

[6] `http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php`

The `ScriptArtifact` class represents an ArtifactTemplate whose type is "ScriptArtifact". Instances of this class offer methods to access the properties stored in the ArtifactTemplate. They also offer a method to extract the artifact to a temporary directory and remove that directory again. The complete artifact must be extracted, before the script in the artifact can be run.

The actual script execution is implemented in classes derived from the `ScriptExecution` class called `BashScriptExecution` and `RubyScriptExecution`. They implement the bindings described in Section 5.1. The `RubyScriptExecution` class also sets up the RUBYLIB environment variable, so extracted libraries can be loaded easily.

5.5.2 Set up Confinement

Before the execution classes can get to work, the confinement must be set up if the `-s` option has been passed. The `ConfinementController` class is responsible for parsing all domain policy and exception policy files from the properties of the artifact and load them. But the policy files cannot be loaded directly, absolute paths must first be injected into the policy. This has been explained in Section 4.5.1. The path to the directory containing the script is not enough, a script also needs access to libraries. These are stored in a different directory. In addition to $SCRIPTDIR the placeholder $LIBRARY is introduced, which will be replaced by the path to the library directory.

The domain policies developed as part of this thesis make use of the `acl_group` feature. This allows to extract common parts of domain policies into a group which is then included for the domain to which they should apply. There is one issue regarding this in the current implementation. The executor does nothing to check if it's own policy conflicts with an existing one. As an example, there might already be an `acl_group 1` loaded. The executor will blindly add additional rules to this group. This is not an issue if Tomoyo is solely used by the executor and only one instance of the executor is run at a time. But for other cases modifications are necessary.

5.5.3 Execute the Artifacts

After all the preparations, a `ScriptExecution` instance is now present on which a simple call to `run` with the input XML and the base directory of the extracted artifact finally runs our script. No matter if it is a shell script or written in Ruby, the result will again be XML. The properties of the returned XML are copied into the original NodeTemplates thereby overwriting the previous properties.

Some cleanup is necessary after the execution. The policy is unloaded and the extracted directory is removed. There is only one way to persist information across script runs. The information must be written into properties of a NodeTemplate or RelationshipTemplate. Through that way, the identifiers of created resources are passed to the artifacts of a RelationshipTemplate, which then uses these identifiers to address the created resources.

6 Evaluation

Generic properties to describe infrastructure resources independent of a provider have been found. Artifacts to provision infrastructure resources at one provider have been developed. The enricher tool was developed to create a CSAR that contains additional properties and these artifacts. And finally, the executor tool was developed to execute these artifacts while also making sure they cannot harm the system. After this toolchain was complete, it was time for an evaluation of how appropriate and usable the toolchain is to provision infrastructure resources.

The evaluation consists of several aspects, which are described in the first section. Each aspect was examined in detail. The results are presented in the following sections.

6.1 Aspects

Just one provider agent with its artifacts was implemented before the evaluation. It was part of this evaluation to implement more. This shows that the general architecture of the enricher and executor fits to multiple providers and is easy to utilize. But not only that, it also shows how to implement artifacts for different providers. For the other artifacts and agents, the fog library is used, but not for the Foreman. Its artifacts are implemented as shell scripts to demonstrate the independence of artifacts from the executor. The agent itself uses Ruby's native HTTP library to connect to the Foreman.

Another aspect is the appropriateness of generic properties determined in Section 3.3. A separate section is dedicated to this aspect, to have a complete overview and not be specific to individual agents.

Quite some effort has been put into securing the execution of artifacts. It was shown for a simple script that Tomoyo is capable of preventing it from doing something it is not supposed to do. Evaluating whether the executor is correctly confining artifacts is another aspect. Some practical attempts and theoretical issues to circumvent the security measure are discussed.

As a final aspect, the whole process of taking a CSAR, giving some information about providers and then get the infrastructure resources set up accordingly is tested on a vanilla system. All necessary steps are documented to show the ease of use and be a guide for others.

6.2 Agent Implementations

Four providers to implement artifacts and agents for were determined in Section 3.5. As part of the integration work only an agent and artifacts for OpenStack were developed. The development of the remaining agents and artifacts for AWS, BareMetalCloud and the Foreman are described in the following subsections.

6.2.1 AWS

The documentation of AWS was of great help to quickly see the possible values for creating instances, volumes, and security groups. Using fog, the interaction itself was also simple. The already existing OpenStack provider agent was a helpful template. Going through all files and changing OpenStack to AWS gave a good picture of what else had to be adjusted.

The provider agent and resource agents had to be extended to support regions and availability zones. This simply meant to query some more information from the user and add AWS-specific properties to the NodeTypes.

There were also a few issues while developing the artifacts for AWS. It was difficult to figure out which DNS names the script used to connect to Amazon. Only the IP addresses were known from Tomoyo's learning mode. They are not ideal to be hard coded into the policy template. It is better to resolve the DNS names when the actual policy is created. Unfortunately, the domain names used by fog could not be determined. Therefore, the complete range of IPs assigned to Amazon was added to the policy.

While the artifact was running and waiting for a resource to be deployed, quite some time can go by. No output is generated, which can give the impression execution is locked up and will never continue. If some indication of progress or activity were implemented the subjective reliability of the executor would improve.

Validation could also be extended to ensure input given by the user or values already present in properties fit to what the provider allows. For example, Amazon's security groups may not have spaces in their name but the NodeTemplate might. This would require a change to the agent interface though, because the complete NodeTemplate must be passed, not just the properties.

While developing the artifacts some errors were made. This caused some resources to be present and some to still be absent. In that case, the resources had to be deleted manually, which costs time and is somewhat unnerving if it must be done repeatedly.

Finally, there is one major issue with the architecture of the executor. It is not possible to have instances in any other security group than the default one. Usually both NodeTemplates are instantiated first, before the artifact of the RelationshipTemplate is executed. This means that the instance and the security group are both present, before the instance is associated with the security group. Amazon only allows changing security group when using VPC. Otherwise the security group to join must be given when the instance is created. This means that using the current architecture, it is not possible to have instances to be part of some other than the

default security group. It would be necessary to somehow detect and handle the connection of an instance to a security group when the instance is created, and the security group must already exist before the instance is created.

6.2.2 BareMetalCloud

Fog did not have good support for BareMetalCloud, which is why the library had to be improved first. This was already mentioned in Section 3.5. Between this first prototype and the actual agent implementation, the API of BareMetalCloud had changed. The documentation on their website did not get updated so the provider had to be contacted. They provided a rough description of the new API methods, which were then implemented in fog. These changes have only been made to the library version of the BareMetalCloud agent, which led to another issue.

The library now differed from the other fog libraries, which caused a problem. Only one version of the same library can be loaded at the same time, because all agents share the same Ruby instance. Attempts were made to unload the library after use, but they were hacks and did not fully work after all. As a quick solution, the modified fog library was copied to all agents thus solving the problem. But the issue with different versions of the same library present in agents still remains.

The `PropertiesHelper` class allows to easily add optional properties, if they are required for the provider. When the user is queried for missing properties any input from the user is accepted. BareMetalCloud has two different locations where resources can be made available. Instead of using the helper class, the provider agent implements the logic himself to be able to validate the input.

BareMetalCloud offers different configurations of servers. Each configuration is described as simple text. A parser for this text had to be written to be able to find configurations that fit to what is specified by the generic size, disk and CPU properties. For the CPU, it was even more difficult because not the number of cores but only the name is given in the configuration text. This name had to be mapped to the number of cores. Such a mapping was generated for all CPU names in use when this thesis was created.

For validation of data, something different was done for the BareMetalCloud agent and artifact. BareMetalCloud only allows a limited number of sizes for storage to be created. A check was not integrated into the agent, but into the ImplementationArtifact. This shows the possibility to do so and gives a reliable way to make an artifact fail when creating a resource. Not immediately helpful but when the behavior in such cases is analyzed it might be of use.

6.2.3 The Foreman

Although the Foreman is capable of provisioning servers at Cloud providers as well, we only integrate it for bare metal provisioning. If Cloud providers are to be used, a provider agent and artifacts can be written directly for them.

The Foreman agent is an exception to all other agents in that the artifacts were not written in Ruby but as shell scripts. This shows that the artifacts can really be of any language. Although shell scripts are not ideal to interact with REST APIs it is still simple enough. The tool `jsawk`[1] was of great help to parse returned data.

Although the artifacts themselves are shell scripts the provider agent and the resource agent had to be written in Ruby. The Foreman provider agent performs the most validation of all. It checks for validity of an entered IP address and MAC address using regular expressions.

The Foreman provider agent has only one resource agent because only servers can be provisioned with the Foreman. There is no support for storage devices or security groups. While the resource agent was developed, it became apparent that the `QueryHelper` is also of use to resource agents. With the use of the `QueryHelper` class, this agent is the most interactive agent of all. This is due to the fact that many values must be selected which cannot be derived from generic properties, like architecture, domain, and environment.

Before using the Foreman agent, an architecture, an operating system, an environment, a domain, and a partition table must be present in Foreman to choose from. Otherwise a deployment is not possible.

6.3 Generic Properties

The four implemented agents use generic and their provider-specific properties. They were good candidates to have a look at the generic properties and if they can be used in a truly generic way.

The first and simplest topic was authentication. As discussed before this is completely provider specific although generic pieces of information like endpoint, username, and password can be derived. All agents use their own properties but use a common name for common properties like username and password, at least where appropriate. For Amazon different names (AccessKeyID and SecretAccessKey) are used. With the Foreman and OpenStack an additional URL property is used that gives the address of the endpoint.

It can be seen that authentication is very similar across providers, but there are individual characteristics. Values for the properties are obviously different across all providers. The decision to have authentication fully provider specific was right.

Looking at security groups next, it must be noted that there are only two providers supporting them. For both, the generic properties work with the issue of very limited expression of security group rules. But this has already been discussed in Section 3.3.6.

For storage, the main generic property is the size with its attributes desired, min, and max. Using a range with a desired value works very well with all providers. The desired value gives the best value to use while min and max allow the agent to choose another value if the desired

[1] `https://github.com/micha/jsawk`

Listing 6.1 Ruby Code to Steal OpenStack Authentication Information

```
Net::HTTP.get('evilserver.com', "c?u=#{io.username}&p=#{io.password}&r=#{io.url}&t=#{io.tenant}")
```

value is not possible. Additional pieces of information like description or snapshot can be used where appropriate. If they are unsupported the agent removes them from the enriched CSAR and their removal is reported to the user.

Virtual machines have the most properties. As with storage, the ranges with a desired value for memory, disk size, and number of CPU cores work very well. Depending on these values a flavor (or configuration) can be determined. If it is possible the desired value can be used directly. As with storage, there are some more optional properties like description and hostname. If supported they are used, otherwise they are deleted. This concept works well and results in an enriched CSAR with properties that fit to the infrastructure resource.

Even though there are generic properties, all provider agents add their additional properties. If for nothing else, then at least for authentication. Many agents use the generic properties to determine actual values required to create the resource at their provider. For example, based on generic properties the correct flavor is determined and its ID is stored as a provider-specific property. TOSCA supports this well and most of the time the user is not involved.

6.4 Artifact Confinement

The actual security measures are provided by Tomoyo. It ensures that only what has been allowed by a policy can be performed. A test of the reliability of Tomoyo is not part of this section. Only a quick test to see what happens when a policy is violated has been performed. The remaining part of the section focuses on issues closer to the executor and enricher, where things can go wrong and prevent confinement as expected by the user. By no means is this be a full security audit, only a few apparent issues are discussed.

The OpenStack script to create an instance was slightly modified to also upload the account information to some other server. This identity theft is very dangerous and would, without a code review of the script, not be found. Only a single line sends the information to an evil server. This line is shown in Listing 6.1.

The executor was launched with the `-s` option and as soon as the script for creating an instance was run it failed with an error. The error output of the executor is very comprehensive to help in debugging issues. The important part of the output is shown in Listing 6.2, unimportant parts are replaced by "...".

If `tomoyo-auditd` was running, the request that was rejected has been logged. The output for our example is shown in Listing 6.3, again shortened to the relevant parts. It can be seen clearly that four attempts to connect to IPs of evilserver.com have been made. All of them were rejected.

Listing 6.2 Output of Executor when Confinement Intervened

```
EXECUTING /tmp/.../scripts/OpenStack/vm/createAtOpenStack.rb
ERROR: Exit status was 1
env:
...
stderr:
/usr/lib64/ruby/2.0.0/net/http.rb:878:in ‘initialize’: Operation not permitted - connect(2)
    (Errno:EPERM)
...
```

Listing 6.3 Tomoyo Logfile for Rejected Operations

```
#2013/11/06 22:43:12# profile=3 mode=enforcing granted=no ...
<kernel> /tmp/.../scripts/OpenStack/vm/createAtOpenStack.rb /usr/bin/ruby
network inet stream connect 94.100.180.199 80

#2013/11/06 22:43:12# profile=3 mode=enforcing granted=no ...
<kernel> /tmp/.../scripts/OpenStack/vm/createAtOpenStack.rb /usr/bin/ruby
network inet stream connect 94.100.180.201 80

#2013/11/06 22:43:12# profile=3 mode=enforcing granted=no ...
<kernel> /tmp/.../scripts/OpenStack/vm/createAtOpenStack.rb /usr/bin/ruby
network inet stream connect 217.69.139.199 80

#2013/11/06 22:43:12# profile=3 mode=enforcing granted=no ...
 <kernel> /tmp/.../scripts/OpenStack/vm/createAtOpenStack.rb /usr/bin/ruby
 network inet stream connect 217.69.139.201 80
```

Now on to some more theoretical discussions about security issues. First of all, the policy must of course be closely reviewed. Either the template or the actual generated policy. Otherwise the policy of the artifact simply allows evil behavior, or it will declare domains which are not used by the artifact, leaving the artifact unconfined. The executor could also be extended to check for such conditions or introduce its own policy to restrict applications run by itself.

The advantage of a policy is that it is much easier to read and recognize allowed operations, than reading source code. There are a few very dangerous statements like `reset_domain`, which moves executed applications to other domains, possibly unconfined domains. If the policy allows write access to `/sys/kernel/security/tomoyo`, the script can even obtain complete control over Tomoyo and modify the policy as it desires. While it is the duty of the user to check the policy, the enricher might be extended to warn about dangerous statements like these.

The confined execution should also not give a false sense of security. It can only protect a limited environment. An artifact could still modify deployed infrastructure resources, adding security group rules, or other things and thus doing harm. But the scope is much more limited than without a confinement at all.

6.5 Complete Pass

The enricher and executor are no big pieces of software. Still, they have their dependencies to libraries and operating system configuration. These dependencies are usually not easy to determine from the development environment due to many wrong paths taken and unnecessarily installed libraries or modifications. This section documents all dependencies of the tools which were found when they were run on a new system. This also shows how easy it is to run these tools.

The system was a default installation of OpenSUSE 12.3. 256 MB of RAM was allocated for the machine but the actual scripts use far less.

At first, all necessary dependencies were installed. The scripts require Ruby 2.0, which is not the default on OpenSUSE 12.3. Therefore, an additional software repository[2] was added. Only after that, it was possible to install Ruby in the latest version. To use the test script to run the enricher and executor, the tool `expect` was installed, too. To run the Foreman artifacts, the js package was installed, required by the `jsawk` tool.

Next, the requirements of the tools themselves were installed. That was the zip gem and nokogiri. When installing nokogiri native libraries are being compiled, which is why the necessary development tools had to be installed as well. These are ruby-devel, gcc, and make.

Finally, all dependencies for fog were installed. These can be read from the `fog.gemspec`[3] file. At the time this thesis was created it listed the gems excon, formatador, multi_json, mime-types, builder, net-scp, net-ssh, and ruby-hmac.

It was now already possible to run the enricher and run the executor without confined execution of artifacts. A simple call to the test script was successful. It fed all necessary information to the enricher. The produced CSAR was then given to the executor and all infrastructure resources described within were deployed accordingly, all completely automatically. If the interaction is done by a human the necessary information must be entered when the enricher asks for it. The executor must be run afterwards, which runs without interaction. A very simple and seamless process.

For the confinement to work, Tomoyo had to be set up. At first, the tomoyo-tools package was installed. After that, the kernel boot parameter `security=tomoyo` was added using YaST. Executing the `init_policy` tool set up default policies, profiles etc. thus completing the configuration of Tomoyo. It was then time to reboot the machine. Afterwards, Tomoyo was available.

By default, only root is allowed to change Tomoyo policies. But running the executor as root is not a good precondition for safe execution. Thus, Tomoyo was instructed to allow management by other users than root. Adding “manage_by_non_root” to the Tomoyo

[2] `http://download.opensuse.org/repositories/devel:/languages:/ruby/openSUSE_12.3/`
[3] `https://github.com/fog/fog/blob/master/fog.gemspec`

Listing 6.4 Script to Prepare System for the Tools

```
#!/bin/bash
zypper ar http://download.opensuse.org/repositories/devel:/languages:/ruby/openSUSE_12.3/ Ruby
zypper in expect ruby ruby-devel gcc make tomoyo-tools js
gem install zip nokogiri excon formatador multi_json mime-types builder net-scp net-ssh ruby-hmac
echo "Add 'security=tomoyo' as a boot parameter in grub using YaST."
read
/usr/lib64/tomoyo/init_policy
reboot
```

Listing 6.5 Script to Make Tomoyo Accessible

```
#!/bin/bash
echo "manage_by_non_root" | tomoyo-loadpolicy -m
chown -R user /sys/kernel/security/tomoyo
```

manager did the trick. It was then necessary to change the permissions of the interface `/sys/kernel/security/tomoyo` to allow access by the user. Simply changing the ownership to that user is the safest, as no other users will gain access to the interface that way. Now the executor did run with confined artifacts, using the -s option.

A script to prepare the machine as described above is shown in Listing 6.4. It must be run as root to be able to install packages and gems globally. The individual steps have been reordered and combined slightly for a small number of calls to zypper and gem. Please note that the script is for 64-bit systems and a reboot will be performed at the end. The user must also perform the bootloader modification. This script only needs to be run once. The script to make Tomoyo available for another user is shown in Listing 6.5. It must be run after every reboot, if the changes are not persisted in some way.

7 Conclusions

This thesis consists of three major parts. A generic interface for Cloud providers, secure execution of artifacts and the final integration with TOSCA. From each of these parts a number of conclusions can be drawn. These are presented in this chapter.

When exploring the APIs of many Cloud providers, a huge difference in quality and brevity was found. Both can be achieved independent of each other but short and to-the-point explanations greatly help when trying to understand how the provider's API works.

No really generic interface for many Cloud providers was found, nor could one be developed. Either the interface deliberately limits what can be done and thus finds a common base for providers, like Deltacloud. Or the interface embraces diversity, which requires to have provider-specific code again to use all features of a provider, such as fog.

It is, however, possible to generalize a few aspects of resources which are managed using the API of Cloud providers. The idea to use a range for numeric values taken from another publication [FRC+13] has been extended by a desired value. These three values (minimum, maximum, and desired) proved very helpful to describe numeric properties of a deployed infrastructure resource.

Some pieces of information, like authentication or the region, are similar across many providers. But the actual values are different across providers, which is why they need to be kept specific for individual providers.

In a nutshell, this thesis gives strong indications that a truly generic interface is unfeasible but on top of diverse APIs some selected generalizations can be applied.

Three security modules for Linux were compared to confine execution of artifacts. Their usability and feature richness differed noticeably. SELinux is complex and works best if the whole system is to be secured by use of labels for all objects. AppArmor and Tomoyo strive to offer a "secure enough" solution which is better than having nothing at all by confining only individual applications based on paths.

It is possible to have fine-grained control over what applications are allowed to do with Tomoyo's simple but still feature rich policy language and domain concept. The policy language is also easy to generate from templates.

Two tools, the enricher and the executor, were developed to process CSAR files. Although these tools are far from supporting everything a CSAR can contain, they show that basic features can be supported with a manageable number of lines of Ruby code. But the real important conclusion to draw is another.

The enricher shows that combining generic properties with prompting for provider-specific information is a straight-forward process. The result of the process are properties and artifacts capable of provisioning infrastructure resources like any other resource. The executor shows that a CSAR with these artifacts can result in a completely provisioned set of infrastructure resources. Even further, the execution of these artifacts can be done on the management system because confinement of the execution prevents interference.

Other TOSCA containers could be extended to provision any infrastructure resources by simply running artifacts assigned for these elements. A confined environment can be used to limit damage done by harmful artifacts. Either the modeling tool or some additional tool can put all necessary provider-specific data into a CSAR, essentially doing what the enricher does. The result is that an artifact-driven approach is applicable to all types of resources described in a CSAR.

8 Future Work

The prototype developed during this thesis is a foundation for artifact-driven provisioning of infrastructure resources. But there are several areas where improvements can be implemented. These areas are described in this chapter.

Windows Confinement Controller

The current confinement controller utilizes Tomoyo and thus only works on Linux. For windows, no solution with similar capabilities like Tomoyo was be found. The main focus in this thesis was to show the possibility in confining executed scripts. This has successfully been shown with Tomoyo on Linux. Still, it might be deemed worthy to research possibilities in confining applications under windows as well.

Integrate Learning Mode

Tomoyo offers a learning mode. It can watch an application being run and derive a policy from how the application behaves. This could be integrated into the executor to derive policy files for all artifacts part of a CSAR. Such a feature would allow rapid bootstrapping of policies for artifacts.

DNS based policies

It is not possible to detect the DNS name used to connect to a server. A TCP connection is always established to an IP address. Therefore, the policy can only contain IP addresses when limiting network traffic. But when developing policies this is an issue since domain names are usually used to refer to servers. The IP might even change between creation of the policy and its utilization, especially with load balancing measures.

Therefore, a noticeable improvement would be to somehow allow specifying domain names in the policy, which are then, on the fly, converted to the IP address the application is using.

Artifact Dependencies

Artifacts are regarded as self-contained executable scripts to perform certain actions. While this sounds nice in theory, it leads to massive resource consumption, especially archive size, because every artifact needs to contain all of its required libraries and other files. This thesis already broke with this concept by introducing a `libraries/` directory, which contains the rather large fog library. It is used by many artifacts and significantly reduces CSAR size and extraction time if it is only included once in the CSAR.

Therefore, it is suggested to model dependencies between artifacts, libraries, system packages, etc. For example, an ArtifactTemplate can list the ruby interpreter and the fog gem as a requirement. The TOSCA container then ensures these are installed, maybe even installs them on the fly, before running the artifact. This reduces the CSAR size and even allows to utilize tools which cannot be distributed as part of an archive.

Leave OpenStack Default Security Group

Instances in OpenStack are always in the default security group and join others if the topology model says so. They do not leave the default security groups. This means all OpenStack instances will always be in the default security groups, although this is not modeled. Assuming they should only be in those security groups described in the model, a complex logic as to be applied. If an instance is not associated with a security group, it should stay in the default one. If an instance is associated with at least one security group, it should leave the default one. But if it is associated with the default security group as well, it should not leave it. This requires a global view on all connected security groups and a place to check this in the code. Or, alternatively, some good logic how to handle this.

Plausibility Checks

The current enricher implementation is just a prototype. It still requires a user to be involved in the matter. For example, it is possible to deploy a virtual machine on Amazon and specify for the linked security group to be deployed at Rackspace. These plausibility violations should be detected and prevented.

A Appendix

A.1 APIs and Frameworks for Cloud Providers

Table A.1 shows which provider can be accessed through which API. The "individual" column indicates if the provider uses a self-developed API instead of a standardized one like OpenStack. Ninefold uses Citrix CloudPlatform, which is based on the open source Apache CloudStack code base.[1] Since no other provider is based on CloudStack, the checkmark is in the column for an individual API.

Table A.2 show which provider can be accessed through which library.

Both tables are based on the provider or framework documentation. No attempts have been made to validate their statements of support. No attempts have been made to compare APIs if they are common although not stated explicitly. If no API documentation was found for a provider it is not included here.

[1] https://help.ninefold.com/entries/21478609-Using-the-Ninefold-API-v2-

Provider	type	individual	EC2	OCCI	OpenStack	vCloud
Abiquo	software	✗				
Aruba Cloud	provider	✗				
Amazon	provider		✗			
BareMetalCloud	provider	✗				
Bluelock	provider					✗
Bluebox	provider	✗			✗	
BrightBox	provider	✗				
CloudFrames	software	✗				
CloudSigma	provider	✗				
CloudStack	software	✗	✗			
DigitalOcean	provider	✗				
Dreamhost	provider				✗	
Enomaly	provider	✗				
ElasticHosts	provider	✗				
Eucalyptus	software		✗			
Gandi.net	provider	✗				
GleSYS	provider	✗				
GoGrid	provider	✗				
Google	provider	✗				
Green House Data	provider					✗
HP Cloud	provider				✗	
Host Virtual	provider	✗				
IBM Cloud	provider	✗				
Joyent	provider	✗				
libvirt	software	✗				
Linode	provider	✗				
NephoScale	provider	✗				
Nimbus	provider	✗			✗	
OpenHosting	provider	✗				
OpenNebula	software	✗	✗	✗		
OpenVZ	software	✗				
OpSource	provider	✗				
ovirt	software	✗				
ProfitBricks	provider	✗				
Rackspace	provider				✗	
RHEV-M	provider	✗				
RimuHosting	provider	✗				
Serverlove	provider	✗				
Skalicloud	provider	✗				
SoftLayer	provider	✗				
StormOnDemand	provider	✗				
Terremark Enterprise Cloud	provider	✗				
VCL (Apache project)	software	✗				
Voxel (now Internap)	provider	✗			✗	
VPS.net	provider	✗				
vSphere	software	✗				
XenServer	software	✗				
Zerigo	provider	✗				

Table A.1: APIs to Access Cloud Providers

Provider	fog	Deltacloud	jclouds	libcloud
Abiquo				✗
Aruba Cloud		✗		
Amazon	✗	✗	✗	✗
BareMetalCloud	✗			
Bluelock			✗	
Bluebox	✗			✗
BrightBox	✗			✗
CloudFrames				✗
CloudSigma	✗		✗	✗
CloudStack	✗			✗
DigitalOcean		✗		✗
Dreamhost	✗			✗
Enomaly				✗
ElasticHosts			✗	✗
Eucalyptus		✗		✗
Gandi.net				✗
GleSYS	✗			
GoGrid	✗	✗	✗	✗
Google	✗			✗
Green House Data			✗	
HP Cloud	✗		✗	
Host Virtual			✗	
IBM Cloud	✗	✗		✗
Joyent	✗			✗
libvirt	✗			✗
Linode	✗			✗
NephoScale				✗
Nimbus				✗
OpenHosting			✗	
OpenNebula		✗		✗
OpenVZ	✗			
OpSource				✗
ovirt	✗			
ProfitBricks		✗		
Rackspace	✗	✗	✗	✗
RHEV-M		✗		
RimuHosting		✗		✗
Serverlove	✗		✗	✗
Skalicloud			✗	✗
SoftLayer			✗	✗
StormOnDemand	✗			
Terremark Enterprise Cloud	✗	✗	✗	✗
VCL (Apache project)				✗
Voxel (now Internap)	✗			✗
VPS.net				✗
vSphere	✗	✗		
XenServer	✗			
Zerigo	✗			

Table A.2: Libraries to Access Cloud Providers

A.2 Generic Property Names

This part of the appendix contains the complete description of the mapping from vendor specific names to generic names. Vendor specific names are the pieces of information that need to be given when interfacing with the vendor. The subsections correspond to subsections in Section 3.3.

The first subsection displays the mapping in one table. This is possible, because there are only three generic properties and no additional properties of providers not mapped to generic properties. For all other subsections, this is different, which is why the representation is different. If there is more than one provider, a table gives an overview which provider supports which generic property. For each generic property the number of matches is also given. This number gives an idea how generally applicable the property is. If there are more providers than can be named in the table heading, numbers instead of names are used. These numbers are also used in the following detailed part. This part shows the provider's properties and to what generic name each property maps. They are arranged in three columns to make better use of space. Optional properties are italic. A smaller font size was chosen to save space.

A.2.1 Authentication

Provider	**API Endpoint**	**Username**	**Password**
AWS [AWS13]	region	access key id	secret access key
BareMetalCloud [BMC]	-	username	password
BrightBox [Bri13]	region	username	password
CloudStack [Clo13]	-	API key	secret key
GleSYS [Gle13]	-	username	API key
GoGrid [GoG13]	datacenter	API key	shared secret
Joyent [Joy13]	datacenter	username	SSH Key Name
Linode [Lin13]	-	username	password
Linode [Lin13]	-	-	API key
Serverlove [Ser]	availability zone	User UUID	secret API key
Storm on Demand [Sto]	-	username	password
Voxel [Vox]	-	hAPI key	hAPI secret
XenServer API [Xen]	hostname (somehow) and client APi version	username	password
Zerigo [Zer]	-	username	API key
OpenStack [Ope]	URL	username	password
OpenStack [Ope]	URL		token

Table A.3: Generic Names to Authenticate

A.2.2 Virtual Machine Setup

XenServer can only create virtual machines from templates or as a clone of other virtual machines. Therefore it does not appear here where properties to create new virtual machine are being looked at.

Generic Name	1)	2)	3)	4)	5)	6)	7)	8)	9)	10)	11)	12)	13)	14)	count
Flavor	✗	✗	✗	✗	✗		✗	✗		✗	✗		✗	✗	11
Location	✗	✗	✗	✗	✗			✗		✗	✗		✗		9
Image		✗	✗				✗			✗	✗		✗	✗	7
DisplayName		✗	✗	✗			✗		✗					✗	6
Hostname				✗	✗					✗	✗		✗		5
RootPassword					✗					✗	✗	✗			4
UserData	✗		✗	✗					✗						4
PrivateIP	✗					✗					✗	✗			4
Tags							✗		✗				✗		3
Description					✗	✗							✗		3
Group	✗		✗	✗											3
SSHKeypair	✗			✗									✗		3
IPv4				✗	✗										2
DiskSize				✗	✗										2
MemorySize					✗				✗						2
CPUCores					✗				✗						2
IPv6				✗	✗										2
Hypervisor				✗	✗										2
Metadata							✗							✗	2
ShutdownBehavior	✗								✗						2
UserName											✗	✗			2
UserPassword											✗	✗			2
IP											✗	✗			2

Table A.4: Generic Names to Create Virtual Machines

1) AWS

KeyName: SSHKeypair
*SecurityGroupId.**:
*SecurityGroup.**:
UserData: UserData
InstanceType: Flavor
Placement.AvailabilityZone: Location
Placement.GroupName: Group
*Placement.**:
KernelId:
RamdiskId:
*BlockDeviceMapping.**:
*Monitoring.**:
SubnetId:
DisableApiTermination:
InstanceInitiatedShutdownBehavior: ShutdownBehavior
PrivateIpAddress: PrivateIP
ClientToken:
*NetworkInterface.**:
*IamInstanceProfile.**:

2) BareMetalCloud

config:
os:
imageName: Image
planId: Flavor
name: DisplayName
location: Location

3) BrightBox

image: Image
name: DisplayName
server_type: Flavor
zone: Location
user_data: UserData
server_groups: Group

4) CloudStack

serviceofferingid: Flavor
templateid:
zoneid: Location
account:
diskofferingid:
displayname: DisplayName
domainid:
group: Group
hostid:
hypervisor: Hypervisor
ip6address: IPv6
ipaddress: IPv4
iptonetworklist:
keyboard:
keypair: SSHKeypair
name: Hostname
networkids:
projectid:
securitygroupids:
securitygroupnames:
size: DiskSize
startvm:
userdata: UserData

5) GleSYS

datacenter: Location
platform: Hypervisor
hostname: Hostname
templatename: Flavor
disksize: DiskSize
memorysize: MemorySize
cpucores: CPUCores
rootpassword: RootPassword
transfer:
description: Description
ip: IPv4
ipv6: IPv6

6) GoGrid
privateip: PrivateIP
description: Description

7) Joyent
name: DisplayName
package: Flavor
image: Image
networks:
metadata.$name: Metadata
tag.$name: Tags

8) Linode
DatacenterID: Location
PlanID: Flavor
PaymentTerm:

9) Serverlove
name: DisplayName
cpu:
smp: CPUCores
mem: MemorySize
persistent: ShutdownBehavior
ide:*:
scsi:*:
block:*:
boot:
nic:*:
vnc:*:
tags: Tags
user:*: UserData

10) Storm on Demand
domain: Hostname
features:
password: RootPassword
type: Flavor
backup_id:
image_id: Image
public_ssh_key:
zone: Location

11) Voxservers
hostname: Hostname
configuration_id: Flavor
facility: Location
image_id: Image
postinstall_script:
swap_space:
admin_password: RootPassword
console_password:
ssh_username: UserName
ssh_password: UserPassword
voxel_access:
backend_ip: PrivateIP
frontend_ip: IP
chef_client:
chef_run_list:
chef_node:
chef_server:
chef_env:

12) Voxcloud
hostname:
disk_size:
facility:
image_id:
processing_cores:
postinstall_script:
swap_space:
admin_password: RootPassword
console_password:
ssh_username: UserName
ssh_password: UserPassword
voxel_access:
backend_ip: PrivateIP
frontend_ip: IP
chef_client:
chef_run_list:
chef_node:
chef_server:
chef_env:

13) Zerigo
cluster-name: Location
image-uuid: Image
name: Hostname
plan-code: Flavor
description: Description
firewall-uuid:
image-options:
ssh-key-uuid: SSHKeypair
tag-list: Tags

14) OpenStack
imageRef: Image
flavorRef: Flavor
name: DisplayName
metadata: Metadata
personality:
networks:

A.2.3 Block Device Setup

1) AWS
Size: Size
SnapshotId: Snapshot
AvailabilityZone: Location
VolumeType:
Iops:

2) BareMetalCloud
size: Size
location: Location

Generic Name	1)	2)	3)	4)	5)	6)	7)	count
Size	✗	✗	✗	✗	✗	✗	✗	7
Location	✗	✗	✗	✗		✗		5
Snapshot	✗		✗	✗				3
Name			✗		✗			2
Offering			✗					1

Table A.5: Generic Names to Create Block Storage Devices

3) CloudStack
name: Name
account:
diskofferingid: Offering
domainid:
projectid:
size: Size
snapshotid: Snapshot
zoneid: Location

4) Deltacloud
capacity: Size
realm_id: Location
snapshot_id: Snapshot

5) Serverlove
name: Name
size: Size
claim:*:
readers:
tags:
user:*:
avoid:
encryption:*:

6) Storm on Demand
attach:
cross_attach:
domain:
size: Size
zone: Location

7) XEN API
sr-uuid:
name-label:
type:
virtual-size: Size
sm-config:

A.2.4 Attach Block Device

Generic Name	1)	2)	3)	4)	5)	6)	count
Volume	✗	✗	✗	✗	✗	✗	6
Instance	✗	✗	✗	✗	✗	✗	6
Device	✗		✗	✗		✗	4

Table A.6: Generic Names to Attach Block Storage Devices

1) AWS
VolumeId: Volume
InstanceID: Instance
Device: Device

2) BareMetalCloud
targetId: Volume
ipAddress: Instance

3) CloudStack
id: Volume
virtualmachineid: Instance
deviceid: Device

4) Deltacloud
volume_id: Volume
instance_id: Instance
device: Device

5) Storm on Demand
to: Instance
uniq_id: Volume

6) XEN API
vm-uuid: Instance
device: Device
vdi-uuid: Volume
bootable:
type:
mode:

A.2.5 Security Group Setup

Generic Name	1)	2)	3)	4)	5)	6)	7)	8)	count
Name	✗	✗	✗	✗	✗	✗	✗	✗	8
Description	✗	✗	✗	✗	✗		✗	✗	7

Table A.7: Generic Names to Create Security Groups

1) AWS
GroupName: Name
GroupDescription: Description
VpcId:

2) Brightbox
default:
name: Name
description: Description

3) CloudStack
name: Name
account:
description: Description
domainid:
projectid:

4) Deltacloud
name: Name
description: Description

5) GoGrid
idinfo: Name
name:
description: Description
state:
policies:

6) Storm on Demand
name: Name

7) Zerigo
name: Name
description: Description

8) OpenStack
name: Name
description: Description
tenant_id:

A.2.6 Security Group Rule Setup

Unidirectional Security Rules Mappings

Generic Name	AWS	CloudStack	Deltacloud	GoGrid	OpenStack	count
Direction	✗	✗	✗	✗	✗	5
Protocol	✗	✗	✗	✗	✗	5
Source	✗	✗	✗	✗	✗	5
FirstPort	✗	✗	✗		✗	4
LastPort	✗	✗	✗		✗	4
Group	✗	✗	✗			3
Description				✗		1
Action			✗			1
FirstPort, LastPort				✗		1
ICMPType				✗		1
Security Group					✗	1

Table A.8: Generic Names to Create Unidirectional Security Group Rules

AWS
ingress / egress: Direction
GroupId: Group
GroupName:
IpPermissions.n.IpProtocol: Protocol
IpPermissions.n.FromPort: FirstPort
IpPermissions.n.ToPort: LastPort
IpPermissions.n.Groups.m.UserId: Source
IpPermissions.n.Groups.m.GroupName:
Source
IpPermissions.n.Groups.m.GroupId: Source
IpPermissions.n.IpRanges.m.CidrIp: Source

CloudStack
direction: Direction
account:
cidrlist: Source
domainid:
endport: LastPort
icmpcode:
icmptype:
projectid:
protocol: Protocol
securitygroupid: Group
securitygroupname:
startport: FirstPort
usersecuritygrouplist:

Deltacloud
firewall: Group
allow_protocol: Protocol
port_from: FirstPort
port_to: LastPort
sources: Source
direction: Direction
rule_action: Action
log_rule:

GoGrid
idinfo:
name:
description: Description
portrange: FirstPort, LastPort
icmptype: ICMPType
protocol: Protocol
direction: Direction
address: Source

OpenStack
security_group_id: Security Group
direction: Direction
protocol: Protocol
port_range_min: FirstPort
port_range_max: LastPort
ethertype:
remote_group_id / remote_ip_prefix: Source
tenant_id:

Bidirectional Security Rules Mapping

Zerigo
enabled: Enabled
local-port-from: LocalFirstPort
local-port-to: LocalLastPort
notes:
policy: Action
protocol: Protocol
remote-ip: Remote
remote-port-from: RemoteFirstPort
remote-port-to: RemoteLastPort

Source-Target Security Rules Mapping

Generic Name	BrightBox	Joyent	count
Source	✗	✗	2
Protocol	✗	✗	2
Destination	✗	✗	2
ICMPType	✗		1
SourcePort	✗		1
DestinationPort	✗		1
Description	✗		1
Enabled		✗	1
Action		✗	1
FirstPort, LastPort		✗	1

Table A.9: Generic Names to Create Source-Target Security Group Rules

BrightBox

source: Source
source_port: SourcePort
destination: Destination
destination_port: DestinationPort
protocol: Protocol
icmp_type_name: ICMPType
description: Description

Joyent

enabled: Enabled
targetA: Source
targetB: Destination
action: Action
protocol: Protocol
port: FirstPort, LastPort

A.3 Complete SELinux Module

In Section 4.3.2 a module for SELinux to confine a script was generated. The complete content of the module is shown in Listing A.1.

Listing A.1: Complete SELinux Module

```
policy_module(aws_deploy,1.0.1)

########################################
#
# Declarations
#

type deploy_script_t;
type deploy_proc_t;

########################################
#
# Myapp local policy
#

require {
        type kernel_t;
        type devpts_t;
        type kernel_t;
        type default_t;
        type unreserved_port_t;
        type lib_t;
        type urandom_device_t;
        type file_t;
        type etc_t;
        type deploy_script_t;
        type nscd_var_run_t;
        type proc_t;
        type deploy_proc_t;

        role system_r;

        class fifo_file { write read getattr };
        class process { fork siginh transition sigchld noatsecure rlimitinh getsched };
        class unix_stream_socket { connectto write create read connect };
        class chr_file { write getattr read open ioctl };
        class tcp_socket { write name_connect connect read create getattr };
        class fd use;
        class file { execute read ioctl execute_no_trans getattr entrypoint open };
        class sock_file write;
        class netlink_route_socket { write getattr read bind create nlmsg_read };
        class lnk_file read;
        class dir { read getattr open search };
}
#============= transitions ==============
type_transition kernel_t deploy_script_t : process deploy_proc_t;
```

```
#============= ROLES ==============
role system_r types deploy_proc_t;

#============= deploy_proc_t ==============
allow deploy_proc_t default_t:dir getattr;
allow deploy_proc_t deploy_script_t:file { read entrypoint getattr open ioctl };
allow deploy_proc_t devpts_t:chr_file { read write getattr ioctl };
allow deploy_proc_t etc_t:dir search;
allow deploy_proc_t etc_t:file { read getattr open };
allow deploy_proc_t file_t:dir { read getattr open };
allow deploy_proc_t file_t:file { execute getattr read open ioctl execute_no_trans };
allow deploy_proc_t file_t:lnk_file read;
allow deploy_proc_t kernel_t:fd use;
allow deploy_proc_t kernel_t:process sigchld;
allow deploy_proc_t kernel_t:unix_stream_socket connectto;
allow deploy_proc_t lib_t:file { read getattr open execute };
allow deploy_proc_t lib_t:lnk_file read;
allow deploy_proc_t nscd_var_run_t:dir search;
allow deploy_proc_t nscd_var_run_t:file read;
allow deploy_proc_t nscd_var_run_t:sock_file write;
allow deploy_proc_t proc_t:lnk_file read;
allow deploy_proc_t self:dir search;
allow deploy_proc_t self:fifo_file { write read getattr };
allow deploy_proc_t self:file { read getattr open };
allow deploy_proc_t self:netlink_route_socket { write getattr read bind create nlmsg_read };
allow deploy_proc_t self:process { fork getsched };
allow deploy_proc_t self:tcp_socket { write read create getattr connect };
allow deploy_proc_t self:unix_stream_socket { write read create connect };
allow deploy_proc_t unreserved_port_t:tcp_socket name_connect;
allow deploy_proc_t urandom_device_t:chr_file { read getattr open };

#============= kernel_t ==============
allow kernel_t deploy_proc_t:process { siginh rlimitinh transition noatsecure };
allow kernel_t deploy_script_t:file { read execute open };
```

A.4 TOSCA Extensions

The XML schema defined by TOSCA was extended by NodeTypes and XML types. These are shown in this section.

Listing A.2 NodeType for Block Storage Devices

```
<NodeType name="BlockStorage">
  <documentation>Block Storage Device</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  <PropertiesDefinition element="tns:BlockStorageProperties"/>
</NodeType>
```

Listing A.3 XML Schema for the BlockStorageProperties Type

```
<xs:complexType name="tBlockStorageProperties">
  <xs:sequence>

    <xs:element name="Size" type="xs:int">
      <xs:complexType>
        <xs:attribute name="min" type="xs:positiveInteger"/>
        <xs:attribute name="max" type="xs:positiveInteger"/>
        <xs:attribute name="desired" type="xs:positiveInteger" use="required"/>
      </xs:complexType>
    </xs:element>

    <xs:element name="Snapshot" type="xs:string" minOccurs="0"/>
    <xs:element name="Description" type="xs:string" minOccurs="0"/>

  </xs:sequence>
</xs:complexType>
```

Listing A.4 NodeType for Security Groups

```
<NodeType name="SecurityGroup">
  <documentation>Security Group</documentation>
  <DerivedFrom typeRef="tns:RootNodeType"/>
  <PropertiesDefinition element="tns:SecurityGroupProperties"/>
  <CapabilityDefinitions>
    <CapabilityDefinition
      capabilityType="tns:SecurityGroupCapability"
      lowerBound="0" name="security_group" upperBound="1"/>
  </CapabilityDefinitions>
</NodeType>
```

Listing A.5 XML Schema for the SecurityGroupProperties Type

```
<xs:complexType name="tSecurityGroupProperties">
  <xs:sequence>

    <xs:element name="Description" type="xs:string" minOccurs="0"/>
    <xs:element name="Rules" minOccurs="1">
      <xs:sequence>
        <xs:element name="Rule" type="tSecurityGroupRule"/>
      </xs:sequence>
    </xs:element>

  </xs:sequence>
</xs:complexType>
```

Listing A.6 XML Schema for the SecurityGroupRule Type

```
<xs:complexType name="tSecurityGroupRule">
  <xs:sequence>

    <xs:element name="Protocol">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="TCP"/>
          <xs:enumeration value="UDP"/>
          <xs:enumeration value="ICMP"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>

    <xs:element name="Ports" type="xs:positiveInteger">
      <xs:attribute name="from" type="xs:positiveInteger" use="required"/>
      <xs:attribute name="to" type="xs:positiveInteger"/>
    </xs:element>

  </xs:sequence>
</xs:complexType>
```

Bibliography

[Appa] AppArmor Wiki. URL http://wiki.apparmor.net/index.php/Main_Page. (Cited on page 31)

[Appb] *Novell AppArmor*. URL https://www.suse.com/documentation/apparmor/. (Cited on page 32)

[AWS13] *Amazon EC2 API Reference*, 2013. URL http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html. (Cited on page 82)

[BMC] *baremetalcloud documentation*. URL http://documentation.baremetalcloud.com/display/bmc/baremetalcloud+documentation. (Cited on page 82)

[Bri13] *Brightbox API Documetnation 1.0.0beta*, 2013. URL https://api.gb1.brightbox.com/1.0/. (Cited on page 82)

[Bur10] I. T. S. Bureau. Activities in Cloud Computing Standardization. *Repository, May*, 2010. (Cited on pages 13 and 17)

[Clo13] *CloudStack Developer's Guide*, 2013. URL http://cloudstack.apache.org/docs/en-US/Apache_CloudStack/4.1.0/html/Developers_Guide/index.html. (Cited on page 82)

[FGJ+09] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica. Above the clouds: A Berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28, 2009. (Cited on page 13)

[FRC+13] N. Ferry, A. Rossini, F. Chauvel, B. Morin, A. Solberg. Comparison of Multiple Cloud Frameworks. In *Cloud Computing (CLOUD), 2013 IEEE 6th International Conference on*, pp. 887–894. 2013. (Cited on pages 9, 15, 23 and 75)

[Gle13] *GleSYS API*, 2013. URL https://github.com/GleSYS/API/. (Cited on page 82)

[GoG13] *GoGrid Wiki*, 2013. URL https://wiki.gogrid.com/wiki/index.php/Main_Page. (Cited on page 82)

[Joy13] *Joyent CloudAPI Documentation*, 2013. URL https://us-west-1.api.joyentcloud.com/docs/public/index.html#introduction-to-cloudapi. (Cited on page 82)

[Lei06] A. Leitner. Novell and Red Hat Security Experts Face Off on AppArmor and SELinux. *Linux Magazine,(69)*, pp. 40–42, 2006. (Cited on page 38)

[Lin13] *Linode API Documentation*, 2013. URL https://www.linode.com/api/. (Cited on page 82)

[LKBT11] N. Loutas, E. Kamateri, F. Bosi, K. Tarabanis. Cloud Computing Interoperability: The State of Play. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 752–757. 2011. doi:10.1109/CloudCom.2011.116. (Cited on pages 19 and 27)

[LS] F. Leymann, T. Spatzier. TOSCA Implementer's Recommendation. Script Invocation Conventions. (Cited on pages 43 and 46)

[Ope] *OpenStack API Reference*. URL http://api.openstack.org/. (Cited on page 82)

[Rub] Really simple and naïve Ruby plugin framework. URL http://thomasjo.com/2010/12/15/really-simple-and-naive-ruby-plugin-framework/. (Cited on page 53)

[Run04] C. Runge. SELinux: A new approach to secure systems. *computing*, 2004. (Cited on page 33)

[Ser] *Serverlove Using the API & CLI*. URL http://www.serverlove.com/support/api-questions/. (Cited on page 82)

[SMP13] Z. C. Schreuders, T. McGill, C. Payne. The state of the art of application restrictions and sandboxes: A survey of application-oriented access controls and their shortfalls. *Computers & Security*, 32(0):219 – 241, 2013. doi:10.1016/j.cose.2012.09.007. URL http://www.sciencedirect.com/science/article/pii/S0167404812001435. (Cited on page 30)

[Sto] *Storm on Demand Developer's Zone*. URL http://www.stormondemand.com/api/. (Cited on page 82)

[Toma] Mandriva hat Mandriva Linux 2010.0, Codename Adelie, veröffentlicht. URL http://www.pro-linux.de/news/1/14907/mandriva-linux-20100.html. (Cited on page 35)

[Tomb] *Tomoyo Linux Documentation*. URL http://tomoyo.sourceforge.jp/documentation.html.en. (Cited on page 35)

[Tomc] Tomoyo Linux Home Page. URL http://tomoyo.sourceforge.jp/index.html.en. (Cited on page 35)

[TOSa] Topology and Orchestration Specification for Cloud Applications Version 1.0. URL http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html. (Cited on page 15)

[TOSb] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0. (Cited on page 43)

[Vox] *Voxel's Hosting API*. URL http://api.voxel.net/docs/version/1.0. (Cited on page 82)

[WIC] *Windows Vista Integrity Mechanism Technical Reference*. URL http://msdn.microsoft.com/en-us/library/bb625964.aspx. (Cited on page 30)

[Xen] *XEN API*. URL http://docs.vmd.citrix.com/XenServer/6.1.0/1.0/en_gb/. (Cited on page 82)

[Zer] *Zerigo REST API v1.0*. URL http://www.zerigo.com/docs/apis/servers/1.0. (Cited on page 82)

All links were last followed on November 25, 2013.

Decleration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature