Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diploma Thesis Nr. 3428

Development of TOSCA Service Templates for provisioning portable IT Services

Kai Liu

Course of Study:	Computer Science
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dipl. Inf. Tim Waizenegger
Commenced:	December 1, 2012
Completed:	June 1, 2013
CR-Classification:	C.2.5, D.2.13, F.1.2, K.6

Abstract

Provisioning Cloud Computing solutions is a tedious and long process, especially when configuring many components and not only offering the application but also the infrastructure. Today, an administrator has to upload, install and configure all the components of a software solution manually, which not only takes time and is prone to errors but also increases the *onboarding costs* at the cloud provider. Decreasing deployment times by the use of an automated system is favored.

TOSCA provides a specification which allows the deployment and management of cloud services by providing a meta-model. *OpenTOSCA* is a framework called *container*, which can interpret the TOSCA specification and is used in this work to deploy an *Enterprise Content Management* stack on a cloud environment, testing the boundaries of its capabilities. After designing deployment models by the means of a *domain specific modeling* approach, an implementation is realized and compiled into a deployment file. This file is also called a *container file* and is processed by OpenTOSCA to initiate the deployment on the cloud environment, including the necessary middleware.

The goal of this diploma thesis is to develop a TOSCA Service Template, that provides a topology model and automates the deployment of ECM core components. TOSCA Node Types for the middleware and application components have to be defined. To further help modeling the topology, a *domain specific model (DSM)* will be introduced by generically defining all components and their operations. That generic model will be used to realize the actual ECM stack components. The ECM stack is then deployed via OpenTOSCA and the execution is reviewed.

Contents

List of Figures

1 Introduction

Provisioning Cloud Computing solutions is a tedious and long process, especially when configuring many components and not only offering the application but also the infrastructure. Today, an administrator has to upload, install and configure all the components of a software solution manually, which not only takes time and is prone to errors but also increases the *onboarding costs* at the cloud provider. Decreasing deployment times by the use of an automated system is favored.

The electronic processing of documents, the communication via email and many other modern technologies open up new possibilities of generating and analyzing data. This is especially interesting for companies, which want to improve their own efficiency and productivity. Banks, for example, have millions of transactions per day. Keeping track and maintaining a history of all performed actions is not an easy task. Another challenge is data security and privacy, regulated in some countries by law[1]. So called *content management systems (CMS)* or *enterprise content management (ECM) systems* are designed to support businesses not only in processing and storing enterprise data, but also allowing analyzing collected data while staying inside legal regulations.

In a production environment, the amount of data can massively increase in a short time. Millions of emails have to be processed and stored for a long time, depending on the countries regulations[2]. Maintaining a server environment in-house is difficult. Many problems can occur as know-how might be missing, the infrastructure of a building is not suitable and running costs result for power usage and maintenance. This raises the *total cost of ownership (TCO)* and may even result in money sink. Here, the concept of Cloud Computing offers an opportunity, where it is essentially possible to rent software solutions, data storage or computing performance dynamically and on an "on-demand" basis. It is natural, that special software needs to be developed to realize such offerings. Further, ideally the program and hardware resources need to be scalable and elastic, to meet varying demands and dynamic workloads to be as efficiently as possible.

Though specialized to a company's needs, research and development of isolated solutions is very expensive and time consuming. That is why existing (enterprise) software is used and migrated into a network or cloud computing context. Unfortunately, that in itself creates new problems as software is usually not easily ported, especially when online functionality was not considered by the program originally. Often, complex software structures evolve that way, growing historically. This may have multiple reasons like necessity, laziness or quick

[1] http://www.gesetze-im-internet.de/bdsg_1990/ (call date: 2013-05-26)
[2] http://www.recht-im-internet.de/themen/archivierung.htm (call date: 2013-05-26)

fixes when trying to get different software programs to work with each other. Any further alteration to the software architecture or topology is very time consuming, difficult and prone to errors, raising development costs.

The concept of ECM systems works well in a Cloud Computing environment. An ECM is a large application comprised of multiple modules, where the correct configuration of all components is difficult. Using a cloud service offers a reduction in maintenance complexity. Standardized software is offered with various degrees of service quality, which allows choosing the right combination of software service packages for an organization. Ideally, multiple properties like scalability, elasticity, security and user friendly interfaces are provided while the whole system stays manageable.

The deployment of large software distributions, including the setup and configuration of the needed middleware, can be very difficult, easily taking weeks and is very susceptible to human errors. Many configuration steps have to be performed manually while looking for a potential mistake is difficult to do. An automated installation routine is favored to counter the above mentioned risks. The ability to reuse so called *Templates* simplifies repeated deployments and automation. Further, error sources can be eliminated while the management overhead is also decreased.

TOSCA provides a specification which allows the deployment and management of cloud services by providing a meta-model. *OpenTOSCA* is a framework called *container*, which can interpret the TOSCA specification and is used in this work to deploy an *Enterprise Content Management* stack on a cloud environment, testing the boundaries of its capabilities. After designing deployment models by the means of a *Domain Specific Modeling* approach, an implementation is realized and compiled into a deployment file. This file is also called a *container file* and is processed by OpenTOSCA, to initiate the deployment on the cloud environment, including the necessary middleware.

A model driven approach grants advantages, for example flexible abstract high-level view which allows the simple combination of multiple software components, maintaining the re-usability of the resulting topology model and keeping the overview. This can further decrease deployment times of similar demands, while also minimizing error factors.

Scope of work

As a deployment example, the ECM stack of the IBM[3] SmartCloud Content Management (SCCM) enterprise software will be deployed, to show and proof that an automated and unsupervised deployment is possible with the OpenTOSCA environment.

The goal of this diploma thesis is to develop a TOSCA Service Template, that provides a topology model and automates the deployment of ECM core components. TOSCA Node Types for the middleware and application components have to be defined. To further help

[3]Trademarks of IBM in USA and/or other countries

modeling the topology, a domain specific model (DSM) will be introduced by generically defining all components and their operations. That generic model will be used to realize the actual ECM stack components. The ECM stack is then deployed via OpenTOSCA and the execution is reviewed.

Outline

This diploma thesis is structured as followed:

Chapter 1 - Introduction :

The development of a TOSCA Service Template for provisioning portable IT Services is introduced as well as the scope of this thesis.

Chapter 2 - Fundamentals :

Central terms and definitions are given that are important to follow the concepts and technologies mentioned.

Chapter 3 - Industrialization of IT Services and State of the Art :

The automation paradigm is elucidated and current developments in this field will be introduced. The imperative and declarative models as well as examples to their implementations in the automation field are presented.

Chapter 4 - Imperative Service Modeling for Cloud Systems :

The TOSCA standard and the OpenTOSCA implementation are explained.

Chapter 5 - A TOSCA Model for ECM Systems :

The domain specific model and the designed model with its approach are defined. Starting with the concept and illuminating the implementation, the chapter also includes an evaluation of the execution.

Chapter 6 - Summary :

An overview of the results is presented as well as an outlook on possible future work.

2 Fundamentals

This chapter will give some fundamentals for a better understanding of the used technologies. Starting with the Cloud Computing concept and the clarification of the term virtualization, other paradigms like hypervisors or the difference between scalability and elasticity will be explained. A domain specific model will be elucidated and a definition of Enterprise Content Management will be given. Finally, a short roundup of BPEL and BPMN is presented.

Cloud Computing

The idea behind *Cloud Computing* is to provide services transparently. The user does not care how the technology is provided and only cares that he can use it. An analogy can be drawn to our access to clean running water. It is not relevant for us how the water is processed or how it is acquired. Only the result in that it comes out of the tap and is usable by us matters.

This concept is now transferred to the IT environment where resources are made available via simple interfaces so that ideally no administration on the customer side is needed. A simple request at the provider is sufficient. Resources can be anything IT related starting with CPU time or CPU power to online disk space or even complete software services. All requested functions are provided while hiding how they are actually supplied. Cloud Computing is more of an umbrella term to describe technologies that provide solutions so that a user can pay for services based on usage. This concept is also called "pay as you go" whereas a real world implementation would be certain phone bill contracts or bike-rent models. These on-demand computing services are world-wide accessible via a network like the internet that is viewed as a "cloud". The term "cloud" refers to the uncertainty and transparency of where the actual resources are located as well as what happens in it, e.g. how the services are provided. Cloud Computing aims to provide computing, storage and software "as a service".

The "as a service" concept can be roughly divided into three "XaaS" categories, defined by the National Institute of Standards and Technology (NIST) [Nat11]: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. IaaS provides the basic access to hardware where the administrator needs to configure everything on his own to get a running and working cloud service. This includes installing an operating system, configuring and installing required software, taking care of security and access rights as well as generally maintaining the functionality of every component of his service.
PaaS is the next step in this categorization where the PaaS provider already gives the administrator access to a platform offering like PHP-Hosting. In this model the customer does not need to take care of the maintenance of the middleware and can concentrate on deploying

and maintaining his services or software.
Finally the last category is SaaS. Here, the required middleware and software to be offered later is provided. The offerings are maintained by the cloud provider, resulting in updates, upgrades and security details depending of the ordered level of quality. The user transparently has access to the most recent version of his software. This concept can be seen as an online leasing of software services. So-called *Service Level Agreements (SLA)* can be negotiated to define what qualities and services the provider has to provide. For example, maintenance or counter-measures for virus infections and security breaches or backup solutions can be negotiated. These are special contracts between the provider and the customer so the costumer can expect a certain level of quality [Nat12].

Often the terms *private* and *public cloud* are mentioned. These are deployment models where a private cloud is operated for a single organization and can be managed on- or off-premise. A public cloud is usually owned by a cloud service company that offers an infrastructure accessible by the general public or a large industry group. A hybrid cloud then is a combination of a private and public cloud

Virtualization

To be able to provide the on-demand feature demanded by the "as a service" concept and combine the concepts of cloud computing, *virtualization* is used.

There are different levels of virtualization. The idea is to not give direct access to the physical hardware by routing the interaction through a virtual interface which then forwards the requests to the actual physical resources. In doing so it is possible to alter the hardware configuration behind the virtual interface. Now it is possible to upgrade, exchange, remove and add resources behind the interface without having to change the access patterns.

This flexibility can sacrifice performance on a x86 system. There virtualization needs some performance and the resource behind a virtual interface will perform worse compared to the direct access usage because of the occurring overhead. It is important that the virtualization technology minimizes the overhead and thus the performance loss so that the speedup by aggregating resources is optimized. Special hardware like the IBM System z on the other side does not have these limitations.

[Wag11] classifies three kinds of virtualization. The first one is the *infrastructure virtualization*, followed by *database virtualization* and finally *service virtualization*.

Infrastructure virtualization provides virtual resources which are mapped to physical resources via software. As there are already different kinds of hardware resources infrastructure virtualization can further be divided.

The first kind of infrastructure virtualization is *server virtualization*. Traditionally a server has a fixed amount of resources where the resources are calculated so that it can withstand the workload at peak demand. As this peak load is usually only temporary, in the worst case most of the time the server will be nearly idle, wasting all the free resources.

With server virtualization the applications are not directly installed on a physical machine but instead on virtual machines. This allows starting additional machines on the same hardware to use any resources that happens to be free at a time. Ideally a migration at run-time between different physical machines is possible so that no downtime or data loss will occur.

Another kind of infrastructure virtualization is *storage virtualization*. Here, an additional controller provides access to virtual disks for other systems. Like server virtualization the hardware behind the controller can be exchanged and the storage capacities can be dynamically allocated as needed. To do so the controller has a mapping function by providing virtual disks for servers and maps them to actual physical storage space. In doing so location independence is introduced. That means that the server does not know where the data is actually stored but only how to access them via the controller.

It is also possible to virtualize a database. Here it is possible to use the resources of multiple clients to form one big database. That way the database is dynamically scalable by adding or removing additional resources as needed. Again, the distribution of the actual physical resources is transparent. In any case the client sees and uses the database like a traditional database.

As an application may use multiple services an architecture that supports such an use case is called *service-oriented architecture (SOA)* and interconnects services via a middleware [Wag11].

Hypervisor

Hypervisors are software between the physical hardware and any virtual server. It enables creating and running virtual systems and when at least one virtual system is hosted the system on which the hypervisor runs on is called the host machine.

There exist two different kinds of hypervisors, *type 1 native/bare metal* and *type 2 hosted*. The former runs directly on the hardware whereas the latter runs on top of another host operation system that provides I/O support like Linux[1] or Windows[2]. As less overhead exists for type 1 hypervisors, they generally have a better performance than those of type 2. These host systems are transparent and allocate resources on their own. That way the user and the used software does not need to take care of workload balancing and is distributed across the hardware automatically by the hypervisor[3].

[1] Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both

[2] Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both

[3] http://publib.boulder.ibm.com/infocenter/eserver/v1r2/index.jsp?topic=/eicay/eicayvservers.htm (call date: 2013-05-26)

Software-Defined Computing

A new term for virtualization in a greater scale emerged in the last few months: *Software-Defined Computing*. This new expression is actually an umbrella-term for further new concepts like *Software-Defined Network*, *Software-Defined Security or Software-Defined Data Center* [SDE12].

Following the Cloud idea and the incorporated virtualization requirements, the software-defined concept continues the train of thought towards a completely software controlled system. Everything is being virtualized, even the infrastructure like traditional physical wiring.

Software-Defined Network (SDN) is a software layer that substitutes the switches and hubs control in the network configuration. Usually, the data flow and data control is handled by the switches or routers directly which in SDN is replaced by a virtual controller managing all. This provides advantages in flexibility in which hardware is actually used and changes to the logical topology can be implemented with less effort. Especially in a Cloud environment with hundreds of switches the administration is easier. The physical location is not relevant anymore. SDN can not only control the virtual switches but physical switches too, as long as they use the same protocol.

Providing security in a network is essential, which is why the *Software-Defined Security* is closely coupled with Software-Defined Network [SDE12]. Firewalls and load-balancers are another software-layer where all data passes through. Those provide the virtual network with the ability to react to threats and problems easily while also being flexible. For example, when an attack occurs setting up another software firewall or rule can be done swiftly. This adaptive security concept is possible by abstraction, pooling of resources and automation.

Software-Defined Storage gives flexibility in storing huge amounts of data. The resources can be allocated depending on the current demands. Again, resources are pooled to dynamically assign and replicate them further enabling the horizontal scaling option and essentially being a transparently distributed data storage.

With a network, security and storage concept, *Software-Defined Data Centers*[4] are now realizable, combining all advantages while reducing overall costs.

The vision is to move away from potential hardware limitations, the Software-Defined Computing concept allows pooling any hardware resources and also the deployment of new networks, for example, can be done in hours instead of days. The Software-Defined Computing paradigm is still young and the control and management possibilities are even more interesting when being able to perform automatically.

The basic difference between cloud computing and software-defined computing is that an enterprise has more control over the location and data flow as direct control over the infras-

[4] http://www.vmware.com/solutions/datacenter/software-defined-datacenter (call date: 2013-05-26)

tructure is given. Switching to a hybrid cloud is possible too as the control over all resources lies within an organization[5].

Scalability

The dynamic allocation of resources as needed to a certain workload demands *scalability*. A distributed system is called scalable when the addition or removal of users and resources is possible without experiencing a significant or no decrease in performance. According to Neuman [Neu94] not only the resources have to be scalable but also the administrative complexity. A system that can theoretically handle a varying demand in resources is not usable if the administrative overhead increases to an amount where it is not possible to actually use the additional resources.

There are two types of scaling: *vertical scaling* and *horizontal scaling*.

Scaling vertically (or scale-up / scale-down) is adding or removing more powerful hardware components to a current system or even completely migrating to a new and more powerful physical machine. The addition of hardware is limited among other things by the current hardware capabilities. For example, to add more CPUs the mainboard must support and have additional free sockets or when adding RAM there must be enough memory banks and larger modules must be supported. Excess resources can be distributed via virtualization. Scaling vertically is further limited by the software being used. For example, a 32-bit application cannot address more than 4GB of memory or the file system may not be able to support an arbitrary large number of files[6]. For example, assuming an e-mail system saves every email a large enterprise receives in a separate file then a NTFS file system can save 4,294,967,295 ($2^{32} - 1$) e-mails. Further assuming the enterprise receives 5 million e-mails per day the system can only save emails for about 2.4 years. That is not sufficient as in certain countries archiving emails is required for at least six years[7]. In principle, everything is always vertically scalable, limited only in the current technology available or meeting physical boundaries [Wag11].

The other type is called scaling horizontally (also scale-out / scale-in). Here, improving computing performance is done by connecting more machines or nodes to the network and using the network for transferring data between the nodes. This is usually more cost effective than investing in a single more powerful component of an existing machine. So called *off the shelf* computers can be used where a small configuration can then be easily gradually expanded. Though cheaper in hardware, the administrative complexity rises easily with this method. Synchronization issues as well as a potential reorganization after structural changes can be time consuming and the underlying network must be very fast. Here, the trade-off between investment and administrative costs must be carefully considered. [Wag11]. The most important issue with horizontal scaling is that the software must be able to support

[5]http://www.computerweekly.com/feature/Software-defined-datacentres-demystified (call date: 2013-05-26)
[6]http://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx (call date: 2013-05-26)
[7]http://www.recht-im-internet.de/themen/archivierung.htm (call date: 2013-05-26)

it. For example, a database must be consistent and it must be ensured that transactions are performed correctly. Guaranteeing this while using multiple instances is very difficult and has limitations. Large databases usually have a high number of concurrent accesses by multiple clients. Adding more nodes raises various questions like how to partition a very large table, how to synchronize write accesses or whether the network bandwidth is fast enough. That is why data bases are restricted when trying to scale them horizontally. [Cod70]

Figure 2.1 shows a graphical representation of the mentioned scaling concepts where in 2.1(a) the server is replaced by a server rack with more performance and in 2.1(b) two nodes of with the same hardware specifications are added to an existing computing network.

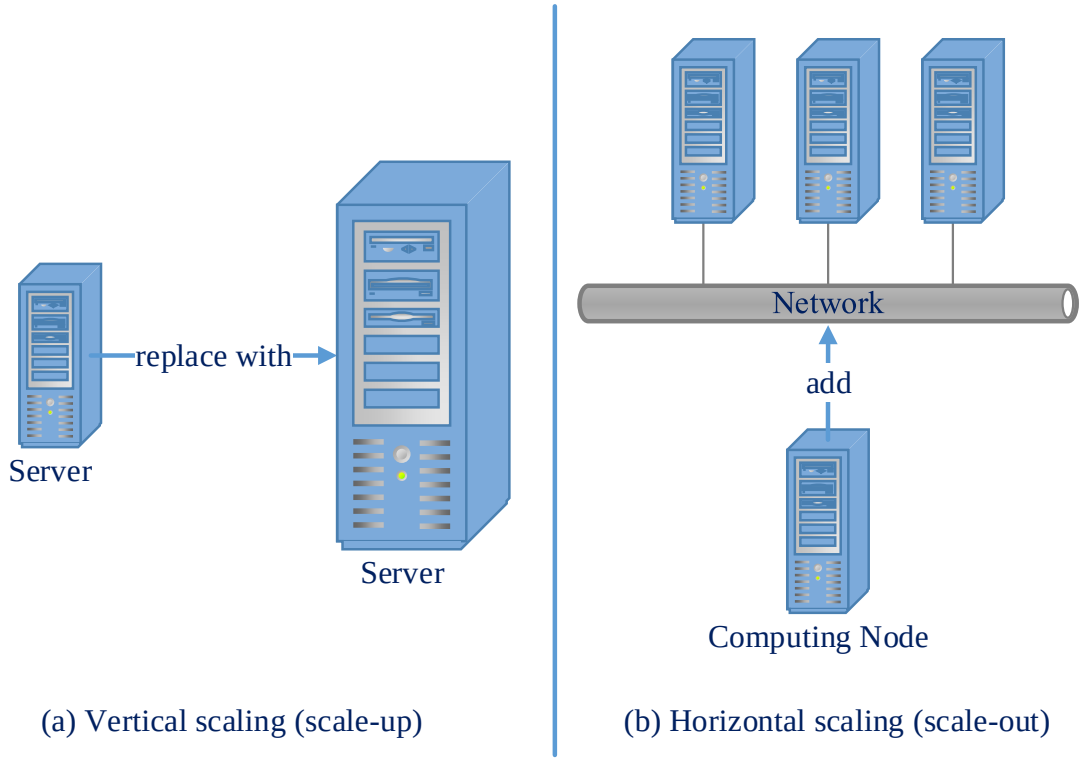


Figure 2.1: Scalability Concept

Elasticity

When considering scalability, *elasticity* is sometimes mentioned at the same time or these terms are used interchanged, indicating that they mean the same. That is actually not true as elasticity describes the ability to react to sudden unexpected workloads. The National Institute of Standards and Technology defines rapid elasticity the ability to quickly and sometimes automatically adapt to demand levels, including the outward and inward scaling [Nat12]. Elasticity expects "unlimited" or at least more than enough resources to be able to meet this sudden rise of workloads.

For example, an online social network needs elasticity as the workload varies heavily, depending on countries, time and other access habits. Here, elasticity is possible as the cloud must be large enough and there must be sufficient different kinds of workloads [Nat12]. If the cloud is too small in regard of the available physical resources, elasticity cannot be met as it would too easily meet the physical boundaries. At a point all resources are spent whereas in a large public cloud sufficient capacities usually exist. That is why scalability is mostly necessary in all cloud sizes and forms (private, hybrid, public), elasticity is not.

Scalability is the provisioning of planned resource capacities that are somewhat expected to be needed over a longer period of time. It follows a more linear workload expectancy than elasticity where elasticity is the dynamic reallocation of already available resources.

For elasticity to work, there is a need for automated management plans. A metric and rule sets (e.g. thresholds) must exist that initiate the adaption to a certain workload.

Enterprise Content Management (ECM)

Enterprise Content Management is not a single closed software solution but typically a combination of multiple components that enables to organize structured and unstructured data and information. This is especially important for large companies that have thousands of employees. Millions of e-mails must not only be delivered and received but also stored. It also must be possible to retrieve specific information like research documents in an acceptable time frame. The organization of large data volumes is vital and comprises all kind of data that are important for an enterprise's success. This includes contracts, research documents, legal documents, costumer profiles et cetera.

The *Association for Information and Image Management (AIIM)* defines ECM as a collection of strategies, methods and tools to capture, manage, store, preserve and deliver data. This definition already shows the five core concepts of an Enterprise Content Management System.

Capture is the acquisition component of data and delivering to the right components inside the ECM system. To achieve that goal the data has not only to be captured but also prepared and processed. Capturing data includes (semi-)automatically digitalizing traditional analogue information like paper documents and including them possibly seamlessly. For this, scan technology is developed like for example Intelligent Character Recognition (ICR), an improvement of Optical Character Recognition (OCR) and Handprint Character Recognition (HCR).

Manage is the component where the captured data is being organized in such a way that it is possible to look for and find a specific information. For that usually databases are used as those already deliver means to store, look up and find data. The administrative side of manage is the access restrictions. Not every employee should be able to access all data an enterprise produces and processes, especially when multiple users collaborate on the same project but have different restrictions.

Store is the part in which the contents of the management component are persistently stored. Here two functional categories are used. The first one is the catalog repository that primarily stores metadata that is used to search and retrieve the content. Content repository on the other hand stores the content in a retrievable or reconstructible way.

Preserve is actually also a storage system but in regard of long-term storage. Especially important is the possibility to migrate the stored information to new media. The past showed that storage technology can get obsolete as new concepts promise safer storage and lower cost whereas is may not be possible to read the old media where the data is stored in the future.

Deliver is the aspect of ECM to make the content accessible to the user by different means. These delivery channels can be electronic or by traditional means like printed paper and postal services. [Kam06]

Domain Specific Language

A *Domain Specific Language* (DSL) is a construct that is often build around an existing programming language, raising the abstraction level. In general, programming languages are multipurpose tools that allow realizing all kinds of projects. In Java[8], for example, it is possible to implement office software but also encryption applications.

As the name suggests a DSL is specifically created to be used in a certain problem domain while it can be textual or graphical. Only in that domain the language is efficient and useable, reducing the gap between semantic and syntax [Spi01]. Thus, a DSL is able to describe what is to be modeled while still being human readable. In fact, domain-specific modeling is modeling the problem rather than the solution for it [Ama04]. Through an automatic code generator and a domain specific library the final code is generated.

Thinking in nouns and verbs and borrowing from natural languages, the concept in how a DSL works will be more apparent [DF08]. Nouns are entities in the problem domain while verbs are the acts that are performed or connections between the nouns. For example, entities could be an "application server" and an "operating system". A verb could be that the application server is hosted on the OS, resulting in a "hosted on" verb. Using those three types the whole problem can be described: an application server hosted on an operating system.

There exist a range of DSL patterns in how a DSL can be used whereas only a few are briefly explained here. *Piggyback* is a DSL that builds upon another existing language when it shares common elements. The generated code is in the human readable language the DSL is based on and not in the machine-readable intermediate language.

A *language extension* pattern is adding new functions to an existing language, extending the syntax and semantic. The difference to the piggyback pattern is the existing language being

[8] Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates

a tool to generate the DSL whereas in extension the original language is already part of the DSL.

The opposite of expansion is also possible, *language specialization*. Here, only parts of an existing language are used, resulting in removing features. This is used when the DSL has a very specialized purpose and can have some wanted side effects like discouraging the usage of certain functions (e.g. unsafe memory allocation).

Domain Specific Modeling

Domain Specific Modeling (DSM) is a further abstraction layer that is above a DSL. It is a model driven approach to describe correlations between entities and their relationships. A working code is created by special tools[9]. Actually three components are needed in such an approach: a domain specific meta-model, a code generator and an optional component library.

The library can contain code parts for the generator either from generic third parties, in-house or self-written code. That is why this construct can help during the code generation process as more functions are available or do not need to be defined again. Especially when being in a later code cycle stage the component library can be handy.

To be comfortably usable, developing a graphical user interface is important. Otherwise the gained advantages by using a DSM can diminish as the overview is lost when designing complex configurations.

The most vital part is the mapping between the model with the DSL behind it. The code generator and the UI must be well adapted and require a lot of attention when building them. A lot of details must be considered and creating a working DSM including the DSL is no simple task.

[9]http://www.dsmforum.org/ (call date: 2013-05-26)

WS-BPEL

WS-BPEL is an acronym for *Web Services Business Process Execution Language*. Currently in version 2.0 its objective is to provide a language that is usable to orchestrate business processes based on web services (WSDL) [BPE07]. Orchestration is the combination of various services to a composition which in turn describes an executable business process.

It is a XML based language that enables modeling relationships between entities, handlers, declarations for process data and activities to be executed. Every service has a certain scope and can only be decided within it. Activities behind a communication partner stay hidden so that the internal behavior is not visible.

WSDL (Web Services Description Language) have ports which act as interfaces to access the functionalities behind them. BPEL actually has no human interaction in mind which is why only web services communicate with each other. To allow human interactions a web service can provide an user-interface though. The communication lines between entities (called partners) are done via partner links whereas partners can have a set of partner links.

BPEL is designed to enable modeling at an abstract high-level. This abstraction allows setting up complex relationships while still keeping the overview. Open source tools like an eclipse plugin to create BPEL plans or an application server with Apache ODE to run them exist.

BPMN and BPEL

As BPEL does not provide a graphical representation, BPMN (Business Process Modeling Notation) is often used to model business processes and then output the BPEL code by using an interpreter or plugin for BPMN. Though it enables the modeling of business processes it is actually mightier than BPEL, meaning in BPMN it is possible to model processes that are not realizable in BPEL[10].

[10]http://www.omg.org/bpmn/Documents/FAQ.htm (call date: 2013-05-26)

3 Industrialization of IT Services and State of the Art

Today computers are a necessity in the business world. They increase proceedings and processing of simple as well as complex processes and thus save time, work labor and money. A requirement is though, that these IT systems must be reliable, secure and fail-proof so that they can be trusted to work as intended. Imagine a bank which would not consistently save the balance of the accounts of their customers or the web portal does not respond reliably. This would breed insecurities and reduce confidence in using the service. The shift from traditional work processes like manual labor over (local) computed aided processing to online computer based assistance (intranet, cloud computing) changed the requirements of the concerned company structures and software used. As such, the systems must be adaptable and manageable.

In the same field of business the requirements usually are the same. For example, in the banking sector that would be reliability and security. As those basic demands usually do not change it is actually not necessary to develop a new security system from scratch for every bank, avoiding expensive isolated solutions. Existing software offerings can ideally be used to save a lot of research and development costs, ultimately reducing the total cost of ownership (TCO). Especially the shift to cloud based IT services should allow a reuse of already existing software systems.

There exists the idea to move towards a so-called *industrialization* of IT services, meaning that the rapid deployment and fast distribution of large programs is made possible. It essentially leads to provide standardized software that can be easily deployed, configured and managed for the customer. Customization, modular buildups and special offerings to satisfy any kind of requirements and use cases are possible. Through standardization of the core components, cooperation between different vendors is possible, increasing the service quality for the customers in the end. If done successfully, parts or the whole offering can be sold at a lower cost to the customers, increasing competitive capabilities while also encouraging developing and using standards. [Clo13]

To exemplify, the idea comes from constructing automobiles, for example, where a car is not a custom machine but comes from a production line where every car of a series is basically the same. Nevertheless, it is possible to have certain adjustment to the customers' preferences and optional extras like leather seats or extra security aspects. That way the car can be manufactured fast and cost-efficient while still providing customization options. Most of the work of constructing the car is automated with little human input, minimizing errors and workplace hazards which again lead to less costs.

For IT services this idea is transferred to automate deployment and management processes as much as possible to take advantage of the before mentioned paradigm. Before TOSCA, there was an industry wide standard missing that providers could follow to be able to cooperate and create compatible software solutions. Hence different approaches were taken to implement automated provisioning of software, using different approaches for their respective implementations.

3.1 Automation and Programming Concepts

Automation is a common concept in the non-IT industry. Originally introduced and still used in manufacturing this paradigm lead to program and use robots to perform labor that was originally done manually. Today though, the idea to perform work as autonomous as possible already extended to other fields like Transportation, Agriculture or Construction. The human interaction shifts from performing the actual work in person to controlling, monitoring and managing it through man-machine interfaces where it is then possible to observe the quality, efficiency, productivity and reliability. [Gol12]

But automation can also take different forms than robots. Using computers, the information transmission speed increases a lot allowing other systems to do automated decisions. These systems for example enable stock brokers to buy and sell their stocks at incredibly speeds after pre-defined trading instructions, called *automated trading* or *algorithmic trading*[1]. But using automated means may have unforeseen consequences as especially in the stock market automated panic sales can influence the economy significantly.

In regards of large software offerings, automation seems to be the logical step towards being able to repeatable deploy an application comfortably.

The automation paradigm presents a logical step when deploying large enterprise software solutions. Here, the complexity can be overwhelming depending on how many and which software modules are used. The manual installation, configuration and necessary management throughout the service's life cycle is prone to errors while taking a lot of time and being tedious. An untrained team might have to spend months in setting up all necessary dependencies as specialists may be too expensive.

Looking at the advantages of automation like reducing error hazards, re-usability, time and potential cost savings the automatic deployment of IT services is desirable. In research and development the willingness to try new things is more apparent than in a production environment where everything must be working correctly and reliably. In such a production environment, a correct automated system can play out its advantages as less mistakes performed means a reduction in costs for maintenance and error correction. [Sri01]

There already exist a few programming paradigms that allow automatic processing in the automation context, roughly categorized into *imperative* and *declarative* approaches.

[1] http://www.economist.com/node/5475381?story_id=E1_VQSVPRT (call date: 2013-05-26)

The Imperative Model

The *imperative programming model* is in essence any programming concept that allows changing a program state by following one or multiple instruction statements by using a sequence of operations. The idea is to view the computer as a classical von-Neumann machine where all actions and operations are explicitly coded in a respective programming language. This results in commanding the system exactly how and in which sequence of commands the computer will change its state. Just like in natural languages the term *imperative* tells someone, in this case a computer, what to do and literally means to command actions that are to be performed. Programming languages that are imperative are for example Assembler, Basic or Pascal. The use of conditional jumps (e.g. *GOTO* statements)[Ste11] is dominant where the program is actually one single algorithm [DH06].

The introduction of functions like loops and if-statements lead to the *structural programming model*. This is what a programmer today usually uses, structuring a program's algorithm into reusable human readable parts. Programming languages which enable such a programming style are the more high-level programming languages we know today like Ada or Fortran [DH06].

A further extension to structural programming is *procedural programming*. Though often used as a synonym, they actually are not the same as procedural programming uses functions or methods in which only inside the function or method a variable is valid and accessible for example. Here, not the whole state is changed but only a part of it. With this concept building separate software libraries that are referenced later is now possible, enabling re-usability.

There exist many more detailed sub-categories under the imperative model which are not discussed here as that is not the scope of this work. Figure 3.1 shows the correlations between the discussed concepts as a Venn diagram. The imperative model is the superset of the structural model and procedural model whereas the procedural model again is a subset of the structural model [Neb12].

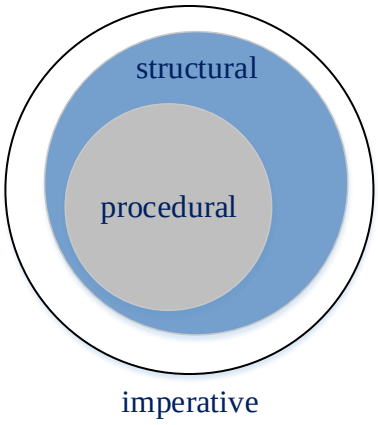


Figure 3.1: Imperative Concept

The advantage of the imperative model is that the concept is widely known. It is relatively easy to follow a program's functionality and execution sequence as everything is explicitly given. The programmer has complete control over every behavioral aspects of his program.

A major disadvantage however is that the overview of a large program can diminish rapidly. The more methods are added and the more functionality is included, the harder it is to maintain the whole code.

In this thesis the expression *imperative* is used as an umbrella term for all previously discussed models in this section where the program follows the exact instructions it is assigned to execute, no matter if the actual algorithm is structured, procedural or straight imperative in the classic sense.

The Declarative Model

The *declarative* concept is regarded as the opposite to the imperative paradigm. Instead of directly instructing the computer *how* to perform a series of actions to get to a goal, it is described *what* the goal is that has to be accomplished. Different ways to specify a goal exist, like defining the goal by properties or stating requirements and capabilities. Therefore, a logical relationship between entities or a model is being required.

The model created has properties assigned to and together with an interpreter or similar construct allow creating an implementation or instance of the model. Naturally, the semantic behind the symbols must be understood by the interpreter.

Analogue to the imperative paradigm, the declarative concept is an umbrella term for further sub-categories like logical programming and functional programming [Llo94].

In *functional programming* the focus is on the application of *mathematical* functions, not functions in the classic imperative sense. The difference is that the result of a function in imperative programming may have side effects, meaning that it will not only return a value but also alter the state of the machine [Hug90]. Mathematical function's outputs on the other hand are purely influenced by the parameters that are inputted [Hud89] and the only output is a value. A value assigned to a variable will never change and therefore a function cannot have side effects. As a result, the order of execution of functions does not matter as values can be evaluated at any time and that no flow of control must be considered.

Based on the theoretical concept of the lambda calculus [Jun04] [Cha97], programming languages like *Scheme* which was renamed to Racket[2] were created. This theoretical construct is a formal system where variable binding and substitution is possible.

Logic programming is another sub-category of the declarative paradigm. Instead of using mathematical formulas, logical constructs are used to define how the machine is supposed to react. A representative for a logic programming language is *Prolog*, among others used for the IBM Watson System to process natural language [AL11]. Prolog is based on the Horn clause,

[2] http://plt-scheme.org/ (call date: 2013-05-26)

essentially comprising the *empty clause* that also is interpreted as a halt statement, the *goal statement* with only negative literals and the *procedure declaration* that has exactly one positive literal and ≥ 0 negative literals [vK76].

Figure 3.2 shows a simple Venn diagram presenting an overview over the declarative, functional and logic concepts.

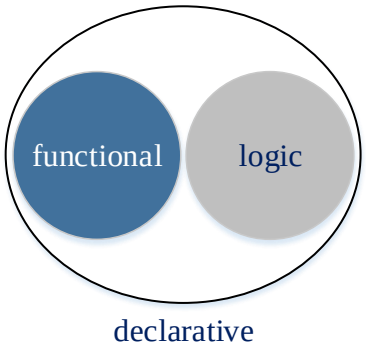


Figure 3.2: Declarative Concept

The advantages in the declarative approach lie in the higher abstraction level when describing what the program is supposed to do and thus it is easier to maintain the overview over large programs. The semantic is simpler to understand because of that abstraction, leaving more room to see the program as a whole. Any changes to the programs behavior are possibly easier as only needs to be described what needs to be done.

The downside though is that the declarative paradigm requires a solid implementation of the underlying framework itself. In order for being able to let the environment execute described features, the underlying algorithms should be able to catch every possible use case, resulting in complicated constructions. Though more effort is needed to create a functional declarative environment the benefits can be huge later on.

3.2 Related Work in Deployment Systems

Automated deployment systems are being developed at multiple institutions and companies. In this section a few examples for imperative and declarative implementations are briefly introduced, followed up by a short discussion about the benefits and drawbacks they provide.

Scripts

A straight forward method do deploy an application is the usage of *scripts*. They are imperative and can be structured or just a series of commands that are being executed. Unix-Shell scripts like in Linux environments are handy and relatively simple to use as they are natively supported by those operating systems.

Puppet Enterprise

Puppet Enterprise is the commercial version of the open source Puppet Project[3], founded in 2005. Written in Ruby[4], it uses a custom declarative language or a Ruby Domain Specific Language to describe the configuration of system services, allowing the definition of reusable modules. Created to manage cloud services, it is also possible to maintain internal company systems. The commercial version contains more functionality like an user interface, support for VMware VMs or managing user accounts. Puppet supports a number of Linux distributions and also recent Windows versions.

It is designed to manage an infrastructure while pre-defined or custom modules are stated via relationships, following a high-level concept [lab11]. A module is a graph of relationships that is reusable. The user interface is graphical, and because of the declarative language the programming effort is minimized. Necessary information is discovered during runtime and compiled into manifests. Manifests are the idempotent declarative Puppet programs, meaning that the reapplication of the same manifest will result in the same state of the system. That is why manifests are not simply called scripts. A system-specific catalog holds the resource and its dependencies and Puppet then applies any required actions to the target system, using the previously compiled manifests.

The system being governed is viewed as an iterative life-cycle with four stages: *Define*, *Simulate*, *Enforce* and *Report*. In the definition stage the relationship graph with modules and their desired state is declared. It is then possible to simulate any changes to the system before committing it, allowing testing and avoiding faulty updates. Enforce is the actual realization of the planned changes, altering the state of the system according to the model provided. Finally, monitoring of all modules is possible by the reporting component. Any intended or

[3]https://puppetlabs.com/ (call date: 2013-05-26)
[4]http://www.rubyist.net/ slagell/ruby/index.html (call date: 2013-05-26)

accidental changes are logged so that appropriate steps can be taken if anything went wrong to restore a correct state of the system. Figure 3.3 is taken after [lab11], showing the stage cycle.

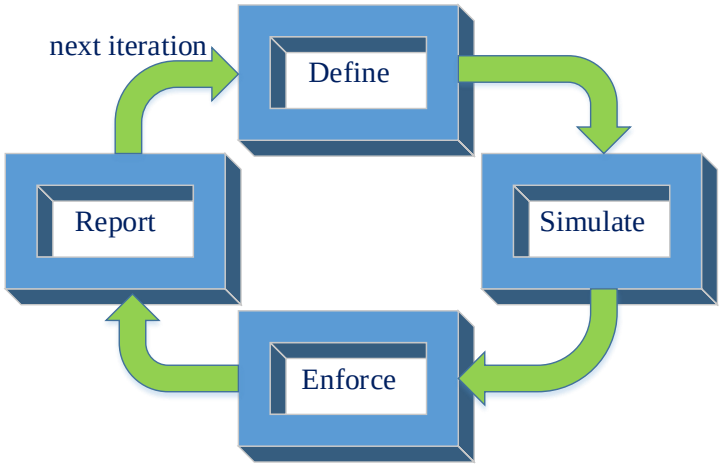


Figure 3.3: Puppet Enterprise Stages (after [lab11])

Chef

Chef[5], like the original Puppet Project, is open source and is capable in managing services. Support for Windows exists, though Chef's roots are in the Linux environment. It uses Ruby with a domain specific language to define the configuration of the system. Chef uses the cooking domain as an analogy for creating and managing a system environment. For example, a command-line tool that enables the management of Chef's main components is called *Knife*, *Recipes* define how infrastructure components are to be deployed and managed and *Cookbooks*[6] are a collection of recipes that define scenarios like setting up a database with all dependencies considered. By abstracting and through that special naming it is emphasized that the defined recipes are supposed to be platform independent and are usable in other projects. There already exists a pre-defined library with many recipes on the official Opscode website[7], allowing trying out different deployments. Of course, it is possible to define custom recipes or combining recipes into another one.

Chef runs as a client/server model where the clients send information to the server which then together with the recipes can react accordingly to configure the clients. In version 11, Chef Server API was rewritten in Erlang, a language designed for concurrency [AVWW96]

[5]http://www.opscode.com/chef/ (call date: 2013-05-26)
[6]http://docs.opscode.com/ (call date: 2013-05-26)
[7]http://community.opscode.com/cookbooks (call date: 2013-05-26)

and other features like being able to change code during runtime. Compatibility for the older Ruby-based versions is maintained.

Figure 3.4 shows an use case example for an utilization of Chef to configure some system components. Here, a user creates a custom database recipe that is added to the cookbook. Previously created custom recipes were stored in the recipe library, together with pre-defined recipes. Further some recipes from the library to describe the intended states of the operating system and a web server are added to the cookbook. Finally, the application of the cookbook results in the configuration of all defined components.

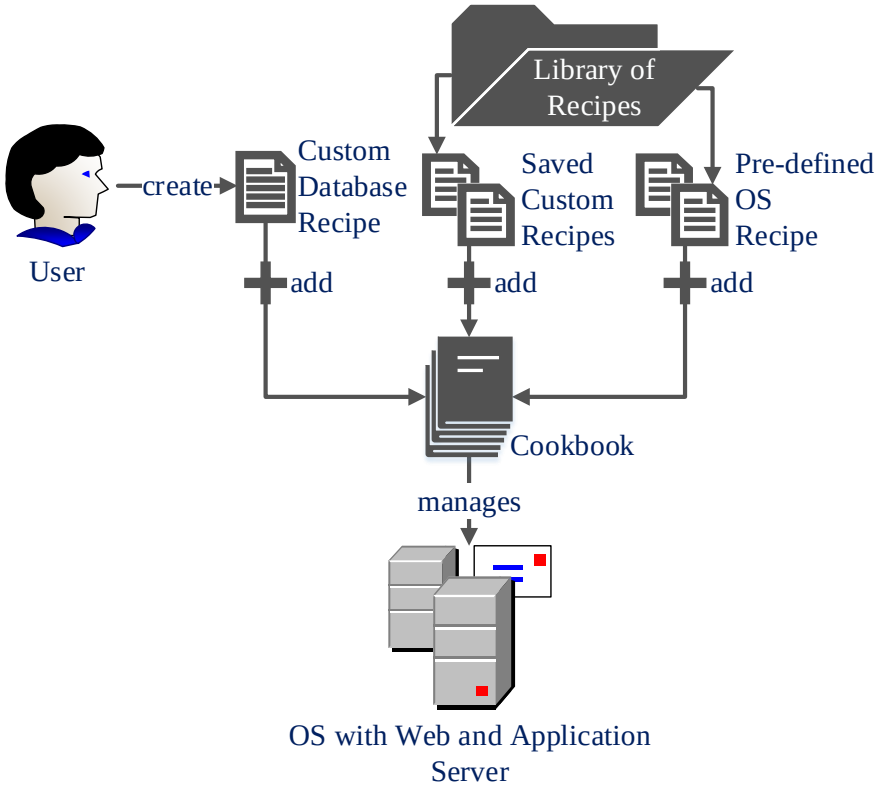


Figure 3.4: Chef Use Case Example

OpenStack

OpenStack is an open-source project that offers an imperative Infrastructure and Platform as a Service and OS deployment and is licensed under the Apache License. A large collaboration of different companies like IBM, HP, Intel, Cisco and VMware among others work together under the management of the OpenStack Foundation. The goal of OpenStack is to comfortably create and publish cloud computing services. The release cycle was changed from a 3 month

period to a 6 month period whereas the current release (2013.1.1) is code named *Grizzly* and was made public on May 9th 2013[8].

OpenStack components

There are a number of different projects running under the OpenStack Project, called components, each having their own code name. Seven of those are the *core components* receiving the current development focus. The general function and their codename in brackets are listed below:

- Compute (*Nova*) is the component that provides virtual servers on demand
- Object Store (*Swift*) allows storing and retrieving files without mounting directories
- Identity (*Keystone*) is the authentication and authorization project
- Dashboard (*Horizon*) provides a web-based user interface
- Block Storage (*Cinder*) enables a persistent block storage for guest VMs
- Network (*Quantum*) manages network connectivity
- Image (*Glance*) is a repository for virtual disk images

All Python components communicate via JSON APIs with each other and when all projects are used, form a powerful distribution system for VMs. Not all are obligatory so a smaller deployment system can be set up. Nova marks the pivotal component as most other either relies on it or communicates with it [Ope13b].

There exist other side projects which enhance the capabilities of the core components, complement them or provide new functions. As that list is quite large, they will not be discussed here.

OpenStack Functionality

To orchestrate a cloud, Nova needs to be able to communicate with the *hypervisor*. The OpenStack Compute component already supports multiple hypervisors like KVM, UML, VMware vSphere 4.1, Xen and others. Nova further allows the management of users or *tenants*, giving them roles with different quotas like the number of volumes or instances they are allowed to create.

Many different ways exist to set up a cloud environment where images and their instances play a central role. *Images* are templates for VMs managed by Glance. An *instance* of an image is a running individual machine on a physical node, where Nova manages the instances. It is possible to run multiple instances from the same base image as images are not subject to

[8]https://wiki.openstack.org/wiki/Releases (call date: 2013-05-26)

changes. The runtime states are saved in other virtual resources specifically created for each instance.

The block storage of data is divided in two different concepts. The first one are the *persistent "volumes"* which are independent from instances. That means that another instance can mount them, if required but it is not possible to mount them concurrently. The *ephemeral storage* is associated with a single unique instance. Rebooting the VM or the host server will not erase it but when the instance is terminated, the data ceases to exist. This storage is presented as a war block, leaving the creation of a partition or file table to the OS of the VM [Ope13b].

Launching an Instance

Figure 3.5 shows how a VM is started as an instance. The *image store* Glance holds a number of predefined images and Cinder contains a set of mountable volumes in the *volume store*. The user chooses a base image that is then copied (1) to the local drive of the computer node with the instance. An ephemeral volume is mounted too that will cease to exist after the instance is terminated. From the *block storage* another persistent volume (blue colored) is loaded from (2), where data changes are saved (3) across independent instances [Ope13b].

OpenStack Heat

Heat is a project started in March 2012 that provides a simplification in orchestrating cloud services. It is based on the Amazon Web Services (AWS) CloudFormation template format and translates any configurations into OpenStack REST calls. Those templates are reusable and all necessary information, configurations and settings are saved into one file. AWS CloudFormation already works with Puppet and Chef to a certain degree and Heat's templates can be integrated in both[9].

[9]https://wiki.openstack.org/wiki/Heat (call date: 2013-05-26)

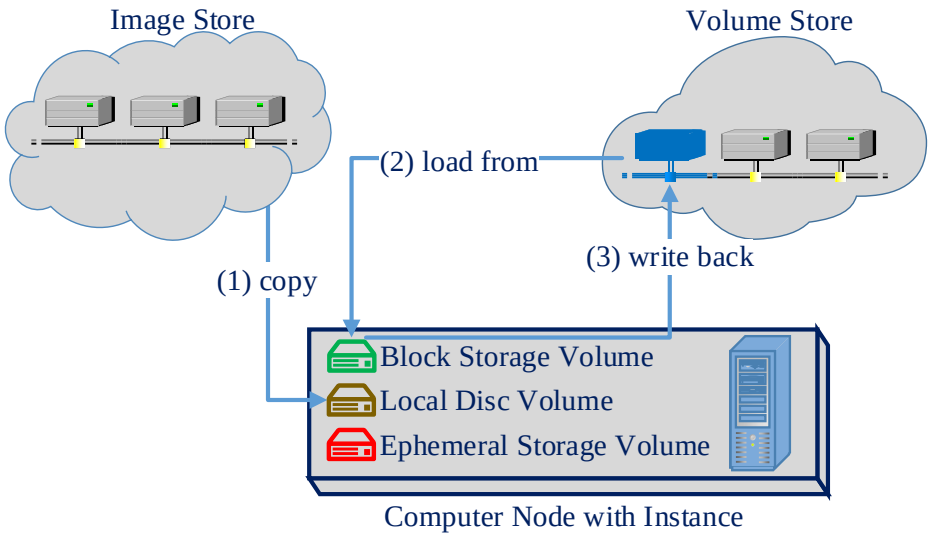


Figure 3.5: OpenStack Instance (after [Ope13a])

IBM Workload Deployer 3.1

The *IBM Workload Deployer 3.1 (IWD)* is a software solution that not only can deploy Infrastructure as a Service and Platform as a Service but also Software as a Service offerings. It is designed to quickly build and install virtual systems over one central software suite. By defining policies and rule sets it can manage and maintain service level agreements on a high-level basis. IWD is part of the PureApplication offering by IBM. IWD supports VMware ESX, IBM PowerVM and IBM z/VM hypervisors [HHM+11].

IWD Deployment Models

To meet the requirements of different services three deployment models types exist: *virtual system patterns* and *virtual application patterns*, complemented by the support for *virtual appliances*. The IWD program is shipped with some basic patterns and templates and it is possible to define and create own patterns.

Virtual system patterns define repeatable topologies so that a middleware system is created in the end. These patterns consist of an operating system with additional software. For example, a pattern can deploy a Linux operating system and then automatically install an application server on it. As the deployed middleware still needs to be configured, it is possible to add scripts that then do all necessary modifications to the installation. The topology is provided according to the requirements of any additional software. Configuring the middleware according to the scripts that are provided by the user is also supported.

Another deployment method is the use of *virtual application patterns*. While virtual system pattern focus on the topology, virtual application patterns emphasizes the application itself. With this approach the description of a software offering shifts from the topology to the application itself. IBM Workload Deployer 3.1 will then choose, deploy and configure the required infrastructure according to the set policies given during the description of the characteristics of the to be deployed application. A special use case of virtual application patterns are *Database Patterns*. As the name already suggests a database is described by selecting the requirements and IWD will create the topology accordingly. Alternatively it is possible to clone an existing database. This allows the IBM Workload Deployer to provide *Database as a Service (DBaaS)*.

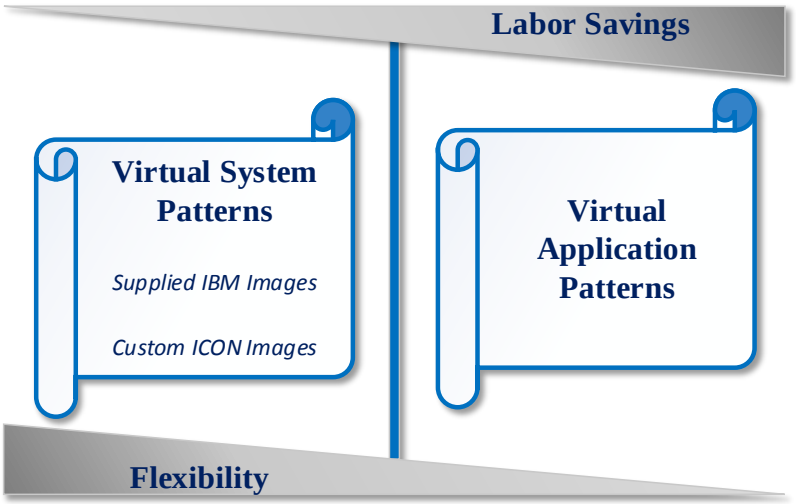


Figure 3.6: IWD Pattern types (after [MFH+12])

Virtual system and virtual application can be combined, allowing planning and deploying complex cloud services. Figure 3.6 shows an overview of the two pattern types. Because Virtual Application Patterns are highly optimized, little to no further customization are intended and they are to be used as they are. This concept allows for a rapid deployment of multiple instances on multiple servers if no variables need to be changed. It does save a lot of time and is convenient but it also means that it is a very rigid construct. Virtual System Patterns on the other hand offer more flexibility additionally to the deployed IaaS offerings further adaptions to the use case can be made. Polarizing, the user must choose between more flexibility offered by the Virtual System Patterns and the labor savings supplied by the Virtual Application Patterns [CIK+12].

This offering is complemented by the support for *virtual appliances*. These are a prepacked software stack that combines the whole infrastructure and software application in one package. The images being used are *hypervisor edition images*, meaning that they follow the *Open Virtualization Format (OVF)*. These images can be created from scratch or existing ones can be customized with the provided *IBM Image Construction and Composition Tool (ICON)*. Virtual appliances can be imported into IWD. The deployment of multiple virtual machines through

one image is not supported. Instead the usage of virtual system patterns is encouraged [MFH+12].

3.3 Comparison and Assessment

Every deployment method has its benefits and downsides. A short assessment of the general imperative and declarative models was already given in their respective sections. Here, the introduced implementations are compared in the following.

Scripts

No special framework is needed to locally or remotely install an application when using scripts. They can take advantage of other automation concepts like Apache Ant to extend their functionality. Testing a command is easy and due to the strict imperative model following the code execution sequence is simple. Because of their simplicity they are usually part of other automation systems. For example, a BPEL workflow can invoke a script that installs an application or a series of scripts are executed while a deployment framework actually only acts as an user interface.

Though straight forward, using them in a cloud environment needs special adaptions like login credentials to work properly and they cannot natively communicate with hypervisors to request an infrastructure. For such a functionality complicated workarounds and additional libraries, software or self-written code is needed. As a result, shell scripts are more suited to be used if an infrastructure already exists and only applications are supposed to be deployed or if smaller steps need to be automated. Maintaining the code is rather difficult in large scripts and thus they provide neither flexibility nor adaptable structures.

Puppet and Chef

Basically Puppet and Chef have the same scope of automating the configuration of cloud services declaratively. Both are free to use in their non-commercial version. Both originated in managing Linux operating systems but have added support for recent windows versions. In continuously providing pre-defined templates, both solutions can reduce the programming efforts of users, allowing a management with minimal programming skills.

The differences are in the details. Puppet does not use Ruby but a custom declarative language, aiming to be simpler and thus easier to get used to. The Ruby syntax of Chef is not model driven like in Puppet and more difficult to learn, resulting in a steeper learning curve. That has a negative effect for training new personnel in a short time and maintaining the configurations.

Chef is procedural and gives better control over the possible performable actions of the automated configuration system as in a model driven approach. Every command can be customized as the code is directly accessible.

The question of which offering to use is not easy to answer and it depends on the preferences of the user. Simplicity versus control must be weighted up as well as the willingness in learning the descriptive languages.

Nevertheless, choosing Puppet or Chef makes little difference when trying to set up a cloud service from scratch. They only provide management support but no automatic deployment and termination of the infrastructure, expecting that a VM is already set up and running. Still, the management aspect is interesting and may be included in other solutions like TOSCA to take care of any management functionalities.

OpenStack

OpenStack only provides IaaS and PaaS support in its core components. That means that servers may be set up but the actual applications that are deployed on the cloud server cannot be easily updated or replaced as this is not the scope of OpenStack. It is possible to completely install and configure all desired applications in one (large) image and then copy that image to all new machines. Unfortunately this approach is not flexible or adaptable. For every update or any small change in the system, the whole image needs to be build scratch and all time consuming settings need to be done again.

This monolithic approach has its advantages in being able to deploy copies of an existing cloud installation onto other nodes rapidly in a fast and simple manner. If other applications next to the base image are to be installed, a script can be delivered with it, initiating any other installations after OpenStack provided the infrastructure.

Figure 3.7 shows the three examples that can increase OpenStack's flexibility concerning the image handling. The first uses a base image and downloads additional software after the deployment via a collection of scripts. The second option is to include some packages in the base image and only the configuration files are pulled from the external source. This can potentially save bandwidth if packages have a huge size. Only getting the customization files allows deploying large images quicker as reading images from the same image is faster. By choosing a selection of configuration files a high degree of customization is still possible, by omitting and ignoring unneeded packages in the image. Finally, the last option is to integrate all packages and configuration files in the image. This marks the usual approach as known before, allowing for very fast deployment but little customization during the process. [10]

OpenStack is not intended to manage software lifecycles running on the VMs it provisioned. It is supposed to only in make the infrastructure available. Other software solutions have to take over to manage the systems, where Puppet and Chef can tie in.

[10] http://www.openstack.org/blog/2011/07/building-and-maintaining-image-templates-for-the-cloud/ (call date: 2013-05-26)

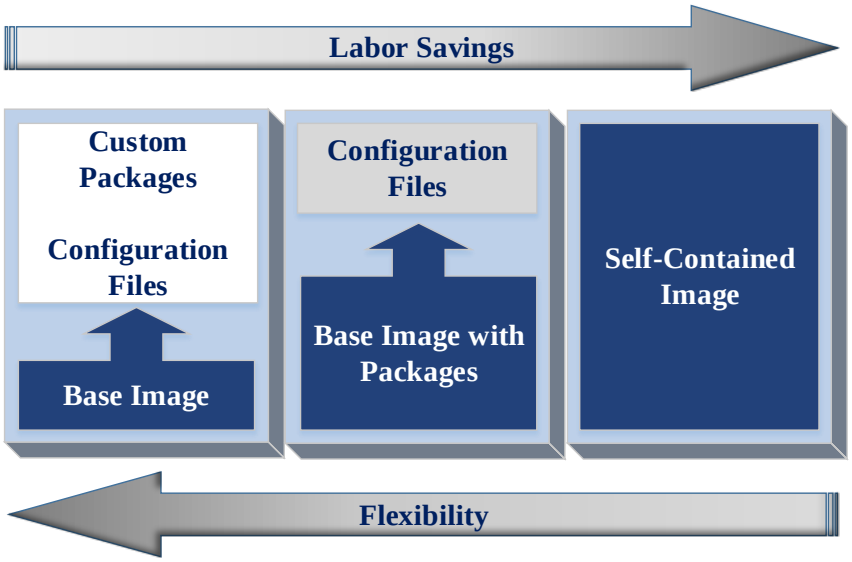


Figure 3.7: OpenStack Image Lifecycle Management

OpenStack Heat is a project trying to complement the core components with a feature to enable the orchestration of cloud applications via a one-file template. Only the AWS CloudFormation is supported which encourages a vendor lock-in. it is a first step to enable OpenStack to model and provide management functionalities beyond the provisioning of the infrastructure.

IBM Workload Deployer 3.1

IBM Workload Deployer 3.1 is a powerful tool to deploy cloud environments. It uses both an imperative and declarative approach which can provide a lot of flexibility and allows high-level view of the topology. The separation in virtual system patterns, virtual application patterns and virtual appliances allows for an informed choice when deciding on a deployment method. Not having to describe every step that is to be performed in detail enables a certain degree of transparency for the user. IWD can deploy all kinds of "as a service" concepts. IWD natively supports AIX which may be specifically needed in some of IBM solutions.

The functional range of IWD is quite large, allowing the provisioning of an infrastructure and also managing the applications that are installed afterwards. It is still possible to use IWD in the same way as OpenStack by preparing images with packages and installation agents or configuration files. The image approach offers a benefit in deployment time if the same image is deployed on multiple clients. An image creation and customization tool is provided in IWD to support this concept.

The usage of different patterns and choices give user control over how a service is deployed. Having the option to save a template and reuse or edit it later on saves a lot of time.

Unfortunately even though IWD is able to perform many tasks, when trying to deploy a new cloud service as cheap as possible the IBM Workload Deployer 3.1 may not be the right choice. OpenStack is an open source project while IBM Workload Deployer is a commercial product, raising the TCO if only the IaaS feature is used. [CIK+12][HHM+11]

In this work the proposed TOSCA approach with its OpenTOSCA implementation tries to combine all the advantages of the pre-mentioned methods while trying to avoid the shortcomings. Many concepts can be reused in a TOSCA context as it aims to provide all XaaS functions like IWD while still being open source.

4 Imperative Service Modeling for Cloud Services

In Chapter 3 some implementations for automatic management were introduced. Most open source solutions concentrate on a narrow aspect of proving a cloud service and supporting its lifecycle. Aside from commercial solutions, deployment systems are being developed that allow providing, managing and terminating cloud services in one application or framework. OASIS with their TOSCA standard is trying to establish such an offering on an open source basis.

4.1 TOSCA

TOSCA is an acronym for *Topology and Orchestration Specification for Cloud Applications*. It is developed and maintained by the OASIS project in which multiple partners from the industry like IBM and SAP collaborate to develop an open standard to provide portability of cloud applications and services.

Portability in the TOSCA sense means that a portable service's definitions and structure can be understood by any other party that supports the TOSCA standard. A so called *Service Template* created in one vendor's environment is interpretable in another vendor's environment. For the portability aspect to work, the functionalities inside a TOSCA container file must be supported by the executing environment.

The idea of TOSCA is to prevent a vendor lock-in for cloud applications. This is done by providing a specification that standardizes the format of cloud applications. A certain Service Template can then be interpreted by any service provider. The structure and behavior of a cloud application can now be construed by another container, even if the application was originally developed for a different enterprise IT department, service developer or cloud provider. This way the interoperability is ensured and it is possible to create more complex services by combining individual components which were originally created by different service vendors [OAS12].

This flexibility allows a customer to choose and combine any offering from a pool of standardized cloud services to his preferences and create a customized solution. The complex and time-consuming creation of a large isolated software suite that has an equivalent functional range is circumvented.

Four general roles are defined to categorize the responsibilities. A *solution developer* creates software for cloud environments. This can be an adaption of existing solutions or initiated by a customer order.
The *cloud provider* offers a cloud platform on which the cloud services are executed. Those services are procured via a market place or self-created.
The *marketplace provider* gives access to a platform where solution developers can present their products and where customers can browse through the offerings. That way the customer can do an informed decision by comparing prices or service coverage [Clo13].

4.1.1 Overview

TOSCA defines a meta-model that allows describing and defining IT services. It not only determines how the topology is build up but also how it is managed later. More precisely: the life-cycle of a cloud application with its creation, management and termination stages can be defined. The structure is defined by a so called *Topology Template* or *Topology Model*. To manage the service throughout its lifespan a set of *plans* is used. Plans that deploy the application are called *Build Plans*, those for management services are *Managements Plans* and for the end of a life span *Termination Plans* are definable.

A management environment is needed to execute the Service Template. Such an environment is called *TOSCA container* or *container* in short.

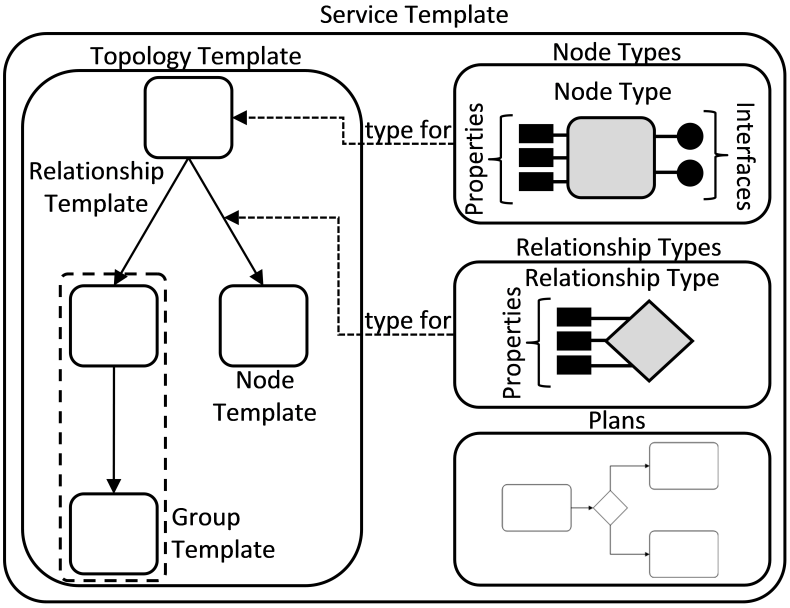


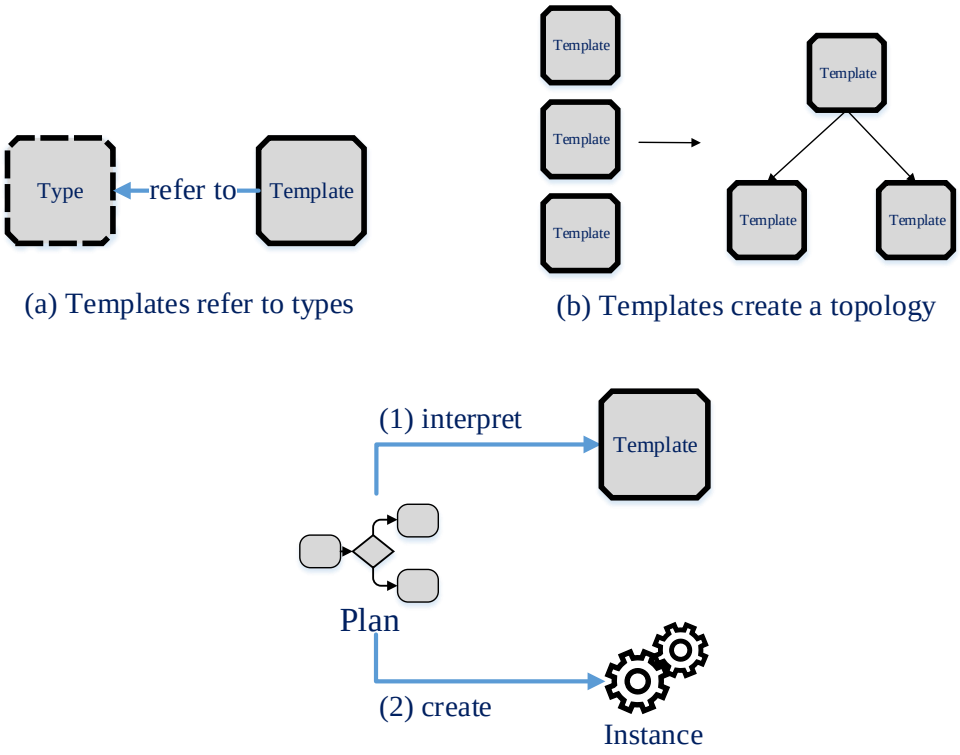
Figure 4.1: TOSCA Service Template (from [OAS12])

4.1.2 TOSCA Meta-Model

With TOSCAs meta-model it is possible to create a model for the topology of a cloud service. It comprises types, templates and plans to provide all necessary modeling components. Figure 4.1 shows the TOSCA meta-model.

The whole process of deploying and managing a service is a directed graph where the nodes are represented by *Node Templates* and the edges are *Relationship Templates* between those nodes. Together they define the *Topology Template* (sometimes referred to as *Topology Model*) of a service which in turn is part of a *Service Template*. The Templates refer to Types and add additional information about a component. This directed graph is the modeling aspect in TOSCA.

Plans are also part of a Service Template and interpret the information and execute the appropriate actions by instantiating the Templates and following the commands described in the plans. The instance represents the running service and falls under the container's responsibility. The extent to how far a container monitors and manages an instance depends on the container's developer.



(a) Templates refer to types

(b) Templates create a topology

(c) Plans interpret the topology and create instances

Figure 4.2: TOSCA Types, Templates and Instances

The relationships between Types, Templates and their Instances are illustrated in figure 4.2 where a generic process sequence is shown. Subfigure 4.2 (a) shows that a Template always refers to its Type. In subfigure 4.2 (b) a set of Templates create the Topology Template. Finally, the Topology Template is (1) interpreted and (2) instantiated by a Plan as shown in subfigure 4.2 (c).

Templates and Types

The *Node Template* determines the occurrence of a *Node Type* where a Node Type can be understood as a blueprint of a component of the service. The properties and operations available to interact with the component via interfaces are specified. The idea of Node Types is to provide an abstract description of a component that is reusable and referable multiple times. A Node Template references the Node Type and adds further needed information to it to operate properly such as how often the component may occur and which parameters are to be used.

A *Relationship Template* refers in the same way to a *Relationship Type* with additional information about the direction of any relationship between nodes. As such, a source and target element is defined as well in what way they interact with each other. Optional constraints can also be described here.

The *Topology Template* contains the Node Templates and Relationship Templates. This all in turn is part of the *Service Template* which includes all the separate parts which are the Topology Template, the Node Types and Relationship Types and finally one or multiple *Plans*. Creating a collection of specific Node and Relationship Templates is allowed and this construct is called a *Group Template*.

In TOSCA it is further possible to define nested structures. A node in the graph can actually be another service template that is deployed by the original service template. That way the complexity of a service can be arbitrary high.

Plans

The Node Types and Templates as well as the Relationship Types and Templates describe the generic structure of the service and contain all necessary information about the described service. A *Plan* interprets this data to create a running instance by executing all appropriate commands.

Any values needed before or during a Plan's execution can be inputted during the deployment by user interaction, may be automatically obtained by automatic operations during runtime or property files. Alternatively, default values can be used or a combination of all those input options is possible. The operations defined in the Node Types of the Node Templates are usually used though it is still possible to define and use other actions in the Plans.

As mentioned above, different plan types exist where each is responsible for the deployment, management or termination of the software. They can cover the whole lifecycle management of any IT service. Plans are process models like workflows that usually contain multiple steps, especially in more complex environments. The TOSCA specification uses existing languages to define the process models like BPEL [OAS12].

4.1.3 Artifacts

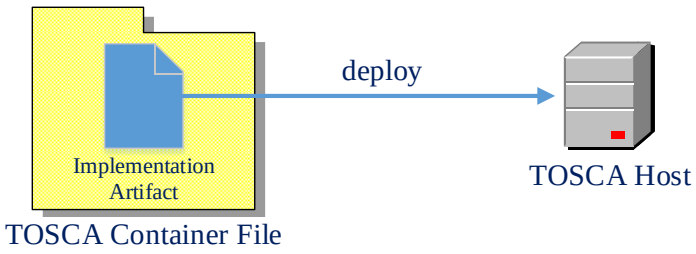
Additional files that are contained in the TOSCA container are called *artifacts*. They provide the needed functions to enable the planed management of the cloud application over its lifecycle. That comprises any further scripts, methods, functions, libraries or even installation binaries and images. To differentiate between binaries and functionalities, two types of artifacts are defined, *deployment artifacts (DA)* and *implementation artifacts (IA)*.

Deployment artifacts are images, binaries or scripts that are deployed to install, manage and configure the cloud application. They are needed to instantiate a Node Type in the target machine.

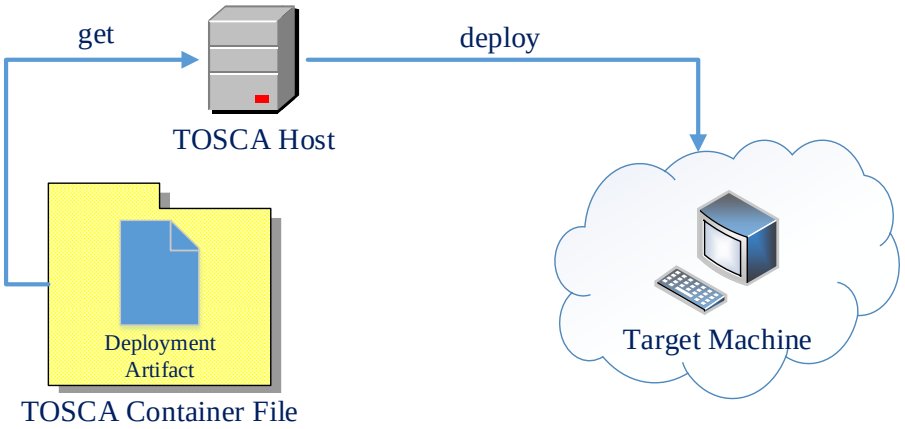
Implementation Artifacts on the other hand provide the functionalities for the Build Plan to process the Deployment Artifacts correctly. This can be a WAR file, for example. They are not used in the target environment but on the host environment that is deploying the cloud application.

The TOSCA container must support any implementation artifacts that are contained and referenced in the Service Template. Otherwise an error will occur during import. This may result in an incompatibility of container files between different container implementations of the TOSCA specification as only a set of artifacts are supported by any container.

Figure 4.3 shows the order in which the artifacts are processed. The implementation artifacts with the necessary functions are deployed on the host (a). The host can then initiate the actual deployment on the target machine by following the Build Plan using the functions now available and referencing/uploading the deployment artifacts on the client in the (private) cloud (b).



(a) deploy functions on host



(b) get artifact and initiate deployment onto target machine

Figure 4.3: TOSCA artifacts

4.2 OpenTOSCA

OpenTOSCA is an implementation of TOSCA developed at the University of Stuttgart. In this thesis the v1 alpha build was used. It follows the recommendations described in the TOSCA Primer document [TOS13] provided by OASIS.

4.2.1 TOSCA Standard Compliance

Though possible in the specification, OpenTOSCA currently only supports the imperative model. The declarative approach is not supported yet and may be included in later iterations.

As the design is supposed to be used in a general way, different Templates and Types are defined after the TOSCA standard's core concept as shown in figure 4.1.

The operational behavior of plans is set by WS-BPEL 2.0 Workflows. In the current version it is only possible to define the Build Plan. The support for management and termination is planned for future releases.

The container file follows a fixed internal layout that has multiple folders and contains all necessary assets whereas there is one folder for each of the deployment artifacts (DEPL-ARTIFACTS), implementation artifacts (IMPL-ARTIFACTS), additional software assets like WARs, images or installers (IMPORTS) and the plan (PLANS). Figure 4.4 shows the internal structure. The Service Template is located in the root folder and is a XML file. The THOR container file itself is a regular zipped file archive.

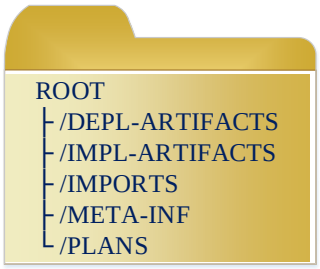


Figure 4.4: THOR Directory Structure

OpenTOSCA is designed to be able to process *THOR* container files. In a renaming procedure the container files were renamed to *Cloud Service Archives (CSAR)* container files which is the official promoted term. In this work however uses the original THOR term as this thesis is based on the alpha build of OpenTOSCA which still describes the files as THOR files.

4.2.2 OpenTOSCA Architecture

The OpenTOSCA container runs in an *OSGi Framework*. OSGi is an acronym for *Open Services Gateway initiative* and is a module system as well as a service platform for Java based software. One of the most prominent examples for an application using the OSGi Framework is the Eclipse IDE.

OSGi follows a component model where a component is also called a *bundle*. The modular approach allows for easier exchange of existing parts as well as making expansions simpler. Also, the bundles can be managed at runtime, meaning it is possible to start, stop or update them during execution. Next to the component model a *service model* is used. Bundles offer their functionalities over services which are saved in a service registry. Other bundles can then bind services and use them.

Figure 4.5 shows the modular architecture of OpenTOSCA as suggested by using the OSGi Framework.

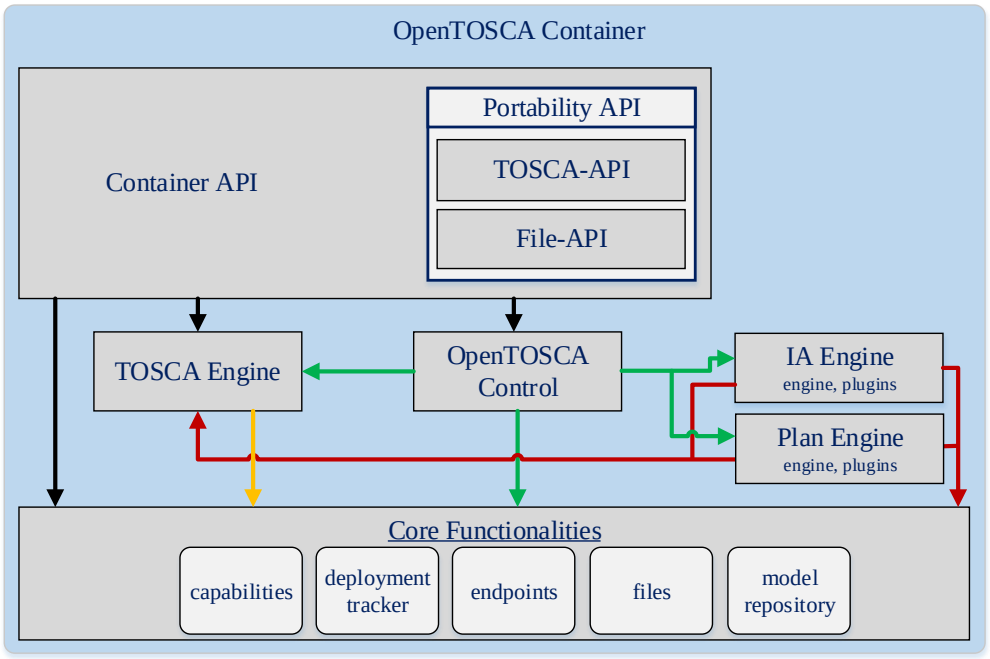


Figure 4.5: OpenTOSCA Architecture

The arrows in figure 4.5 point in the direction of the initiated communication. The called components act according to the requests and return any results.

The *Container API* is the root for all REST-URIs. The *File-API* handles uploads of new THOR files and the extraction of files out of them. Traversing through THOR files in the container is handled by the *TOSCA-API* for traversal and retrieval. These two APIs form the *Portability API*. The Container API also points to running deployment processes and monitors their status. Available methods are mapped and invoked when requested by a deployment process.

The *Core Functionalities* component is the central store of the container. It uses the local DB and file system while the actual implementation is organized in internal components. This provides the advantage that the internal interfaces may change without influencing the published interface. The *Core Endpoint Service* manages WSDL and REST Endpoints and the *Implementation Artifact Engine (IA Engine)* puts the endpoints of deployed implementation artifacts in there. The *Plan Engine* uses those to update the addresses in other plans. The *Core File Service* manages, validates and extracts the THOR files, parses the THOR Manifest and saves THOR meta-data. The Container API invokes the save feature and the other engines may access the necessary files. The *Core Model Repository* manages the Service Templates and the *Core Deployment Tracker* monitors the deployment status of THORs. Finally, the *Core Capability Service* manages the capabilities of IA / Plan Engine plugins and those of the container.

The *OpenTOSCA Control* component checks the availability of needed main components and provides control features while also being responsible for activating engines. The current deployment status of a THOR file is set here, too

The *TOSCA Engine* is responsible for validating a TOSCA-document and resolving the included imports and references. At the same time it also holds the references available for further usage. Additionally, it provides consolidation functionalities on Node Templates.

The *Implementation Artifact Engine (IAEngine)* is called by the Node Template and gets the consolidated IAs via the Core. It analyzes necessary capabilities, delegates them to the right plugin for deployment and returns the relative endpoint which will then be saved. The Plugin then fetches the respective implementation artifact, reads the relative path and deploys the file. Then, the absolute endpoint is returned.

Finally, the *Plan Engine* processes the plan elements directly from the TOSCA-XML data. One plugin per planning language is needed. Currently only the business process language BPEL is supported. Plans invoke the operations provided by Implementation Artifacts. When a plan is deployed, the engine analyzes which operations are needed and establishes the corresponding bindings.

4.2.3 General workflow and process sequence of the OpenTOSCA-Container

Working with the OpenTOSCA container is fairly rigid. Though only the deployment feature is implemented in the current version the general process in creating a THOR file and executing it inside the OpenTOSCA container should be the same for the Management and Termination Plans.

Creating a THOR Container File

Before starting to model all necessary elements the overall design must be defined. After having an idea what is supposed to be deployed, the actual implementation can begin.

The THOR container file needs at least one Service Template (an XML document) defining the topology. Here, the existence of any deployment and implementation artifacts is held as well as all Node and Relationship Templates. All further necessary information like fixed credentials or VM properties are also defined. Following the THOR container file layout shown previously in figure 4.4, all artifacts and other software parts are stored in the respective folders.

Since no other plans besides the Build Plan are implemented yet, only a deployment procedure can be specified. Using BPEL, a workflow is designed using already existing supporting tools. For example, in eclipse the BPEL designer plugin can be used for any modeling attempts. All plans are saved in the respective plan folder of the container file. The folder structure is then zipped into an archive and the file name ending changed to `.thor`.

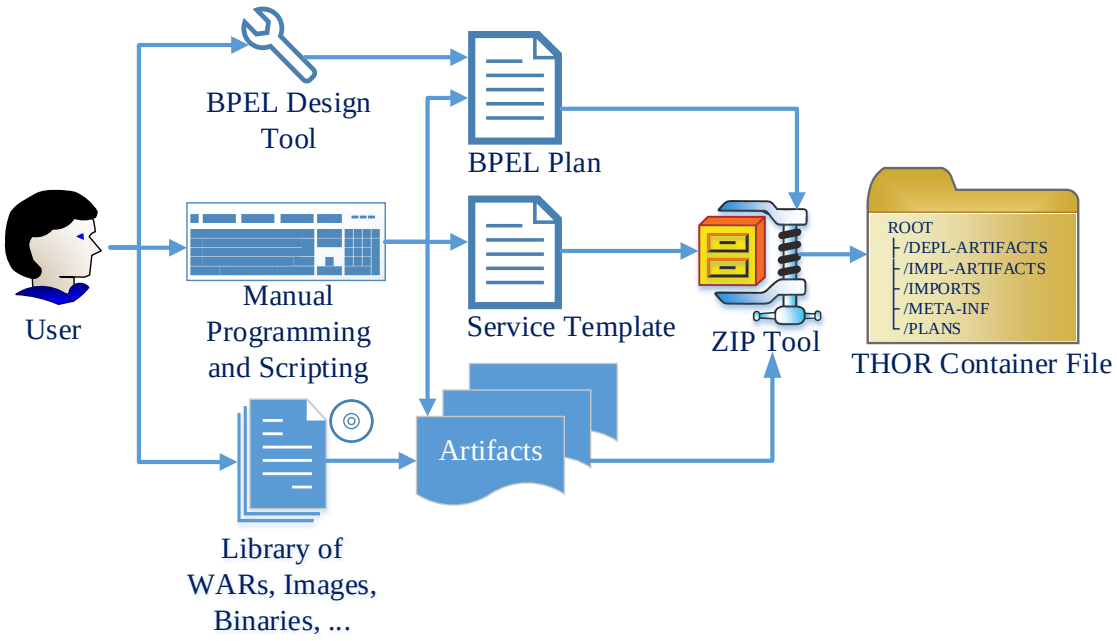


Figure 4.6: Creating a THOR Container File

Figure 4.6 shows a visual scheme for all steps described. The *User* either creates plans by using a BPEL design tool or edits them manually. As there does not exist a completely working graphical tool for OpenTOSCA for defining Service Templates yet, these have to be manually coded in XML. Deployment and implementation artifacts are all directly stored in the container file's respective folders. With a ZIP tool all these parts are then combined in the THOR container file.

Deployment via the OpenTOSCA container

By executing the THOR file in the OpenTOSCA container a running instance of the service is instantiated on the target machine.

Using a web front-end, the THOR file is loaded and extracted to the host's local file system. The contents are then sent to the core which will return the QName. With that QName the OpenTOSCA Control component will resolve the Service Template via the TOSCA Engine. Then the extracted Service Template is returned and used to deploy the assets as the Service Template dictates. For that the IADeployment is invoked which deploys the implementation artifacts.

The previously on a business process server deployed plans are invoked. The actual execution of any plan must be initiated via a SOAP call either through the business process server's web front-end (if supported) or by using a SOAP tool. For that, the plan's endpoints must be known or manually looked up as they are not returned or shown after the deployment steps by the container. The plan creates an instance by using the Implementation Artifacts on the Web Application Server and the following its instructions. Any in the Service Template referenced Deployment Artifacts are deployed on the target machine in the process. After all these steps have been processed, the deployment is done on the OpenTOSCA side.

Figure 4.7 and figure 4.8 are a graphical description of all the steps explained above.

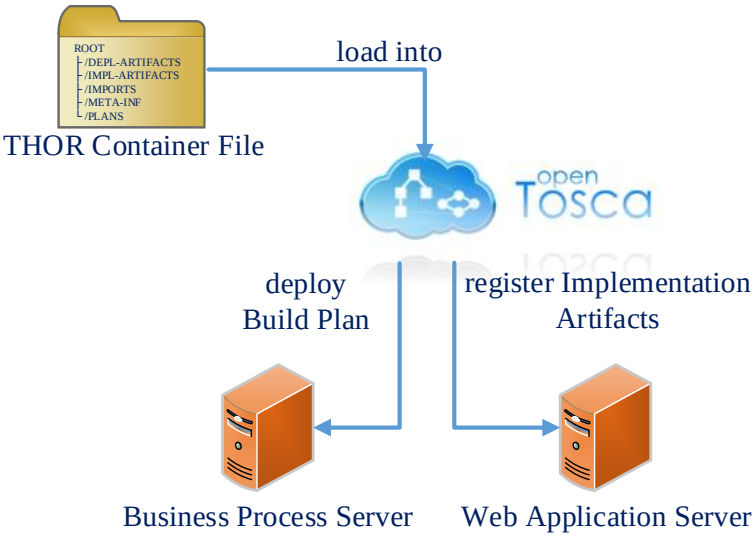


Figure 4.7: Loading a THOR into the OpenTOSCA environment

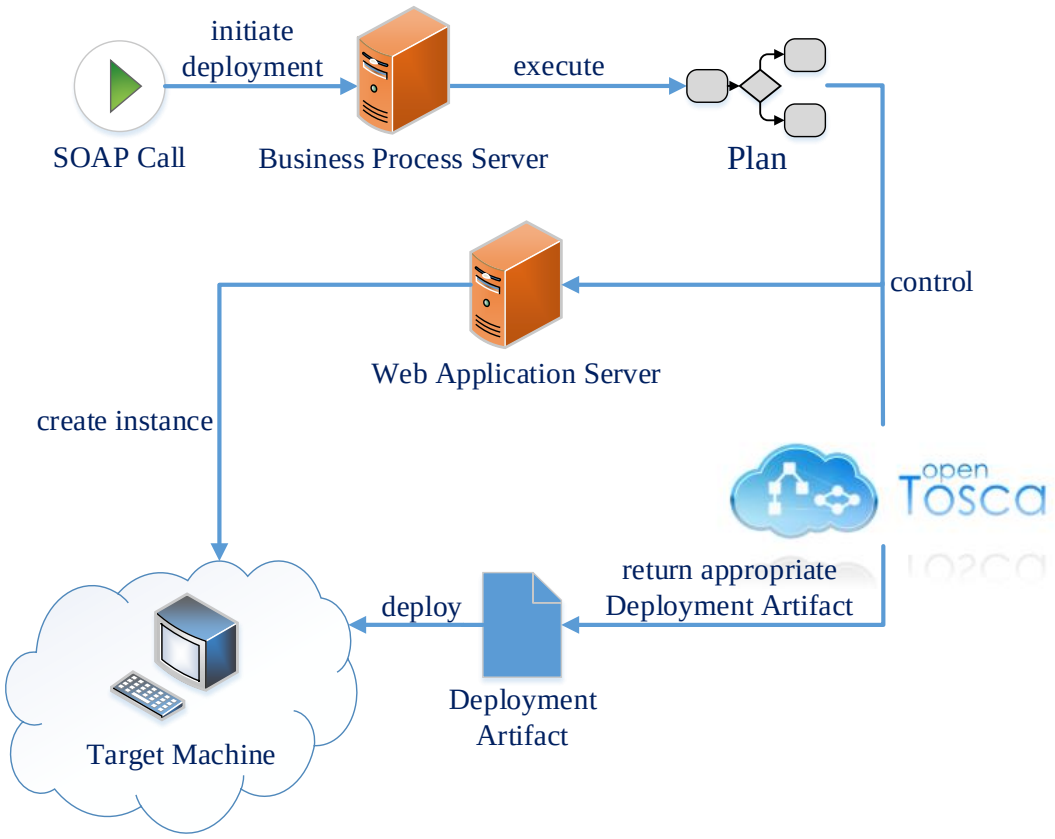


Figure 4.8: Deploying a cloud service

5 A TOSCA Model for ECM Systems

OpenTOSCA was used to deploy an ECM stack and test capabilities and functionality. In this chapter the approach is elucidated starting with the concept and development approach, going on with its implementation and concluding with an analysis of the results.

5.1 Concept

The development approach was an iterative, slowly expanding the scope and evolving the program. A domain specific model was defined to describe the different components needed to deploy the ECM application, providing a unified naming system as well as allowing maintaining a better overview. After that, a corporate ECM software is introduced and the core components of its basic ECM stack are described. Finally, the model for the OpenTOSCA container is drafted that was implemented later.

5.1.1 Domain Specific Modeling in TOSCA for ECM

Creating a domain specific model results from a practical point of view. It enables users to immediately recognize the functions of a component without having to learn much about them. Usually a user does not care what kind of database or LDAP system is used. It is only relevant that an authentication system is present and working and not how it was implemented. The domain specific language helps to insert another abstraction layer that simplifies the design process and thus allows an easier modeling of the topology while passing the information to the operator in a more understandable way. The Service Template in OpenTOSCA itself is already a kind of DSL, based on the XML scheme (an XSD file).

The TOSCA specification provides a meta-model describing any use case that might emerge while not describing concrete Node or Relationship Types. In the ECM context a more specific definition of usable nodes (nouns) and predefined relationships (verbs) is useful. The structure of TOSCA is suitable for an extension-like DSM by adding specific Node and Relationship Types. That would not reduce the functionality of the original syntax and semantics in any way, instead it adds new types. That ensures that even non-IT affiliated users may understand what semantics a node represent. Developing a complete DSM with all functionalities like a code generator is not in the scope of this thesis.

In this work the meta-model part is designed by extending the TOSCA meta-model and therefore a possible naming system and structure of Node and Relationship Types is proposed.

Figure 5.1 shows the designed generic Node Types with example functions and figure 5.2 the respective designed Relationship Types as arrows. The notation follows the Vino4TOSCA proposal of [BBK+12].

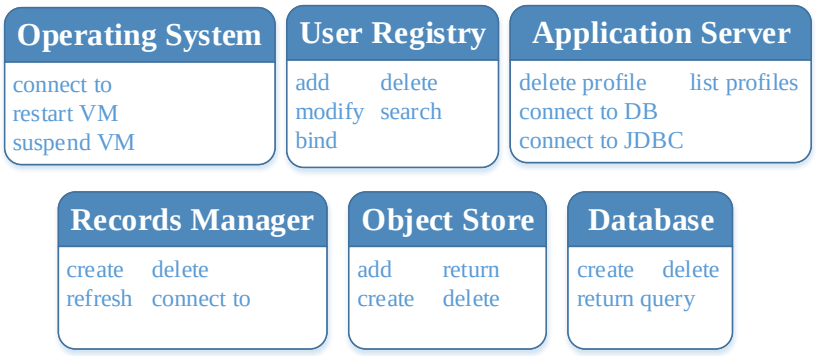


Figure 5.1: Model: Node Types

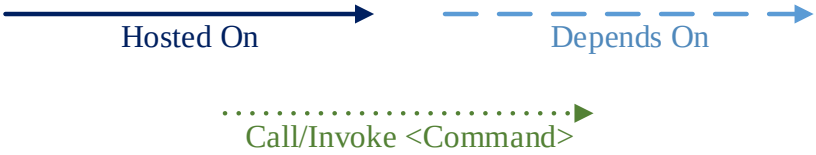


Figure 5.2: Model: Relationship Types

5.1.2 IBM SmartCloud Content Management

SmartCloud Content Management (SCCM) is an IBM service offering based on an ECM system, providing email, security, records management and other support functionalities. Thus the goal is to provide an enterprise platform for managing multiple risk and compliance solutions. This platform aims to reduce the costs while being compliant with corporate governance and industry-specific regulations and simultaneously improving visibility and control of business relevant content.

SCCM Architecture

The SmartCloud Content Management application consists of several components that must be installed and configured correctly.

Figure 5.3 shows a simplified view of the whole system architecture with multiple components that make up the whole service. The actual SCCM package is deployed on a VM which supports an *AIX* operating system as backend. On top of that a *WebSphere Application Server (WAS)* is deployed which offers a Java application server runtime environment with additional

features like enhanced reliability, scalability or network deployment. The data is stored in a *DB2* database which provides a scalable and reliable data center.

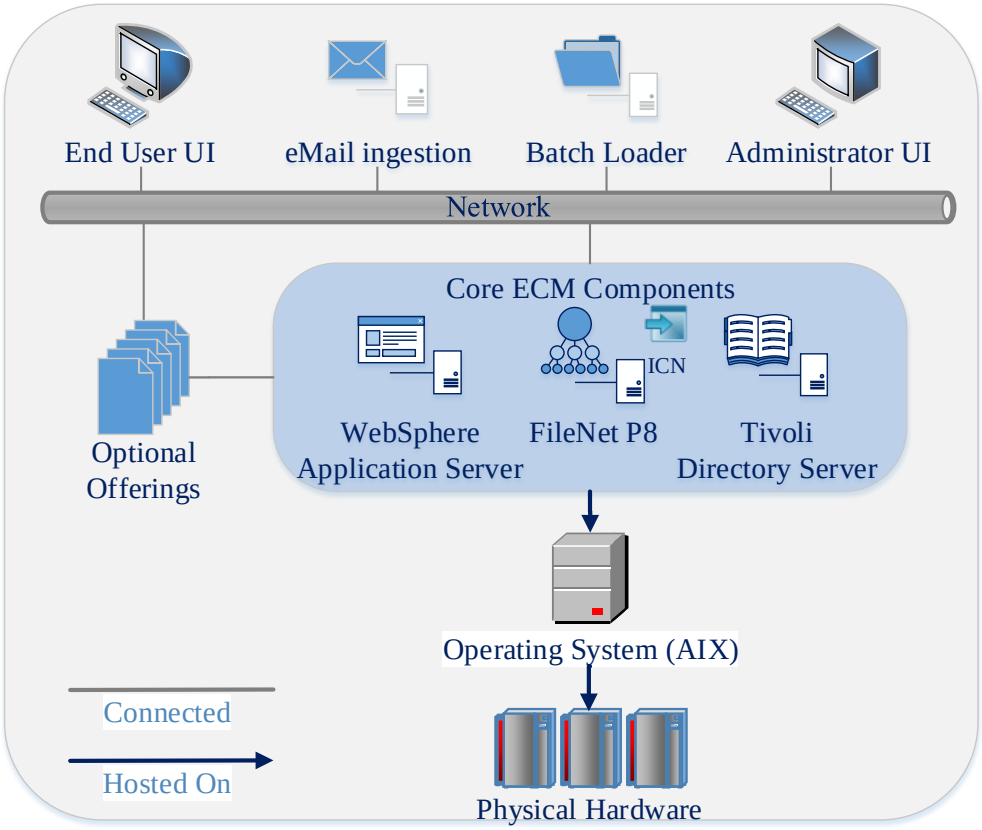


Figure 5.3: SCCM Architecture Overview (after [IBM12a])

The core ECM system is FileNet P8. It already provides core management capabilities for versioning, security and lifecycle management of records, where records can be any kind of document or asset in the system. Furthermore it is able to provide workflow capabilities for a business process management like electronic forms and such.

Multiple optional offerings can interact with the core ECM stack. For the user interaction *Workplace XT* or *IBM Content Navigator (ICN)* can be used. The full text engine is provided by the *Content Search Server (CSS)*. Cognos is able to archive reports created by client administrator directly into FileNet P8 via a *Content Management Interoperability Service (CMIS)* interface. Support for searching and exporting email documents the *eDiscovery Manager (eDM)* interacts with the buildup. The *IBM Content Collector (ICC)* component is ingesting emails and can only be run on a Windows system. Therefore ICC runs on Windows and is connected with AIX.

Several service components can now be connected to the system. The user interaction is done via the e-Discovery UI and End user UI. All emails are transferred through a VPN tunnel first to the ICC component which then forwards these records to the ECM system

for further processing. The *Secure Batch Upload* component enables it to comfortably capture multiple assets automatically. Lastly, the optional LDAP integration provides a user access management tool [IBM12a].

SCCM client view

The client view of SCCM is basically composed of two large services. The *Document Compliance Archiving* service captures and archives all relevant documents except emails according to its retention rules into the system. This can be done manually by using the *Business User Client* management tool or automatically via a batch loading process. The service not only provides archiving mechanics but also retrieving methods [IBM12b].

The other component is the *Email Compliance Archiving* service. Here, by setting and customizing retention rules, emails are sorted and processed accordingly to those settings. A default retention rule is applied if no custom rule exists. Archiving and retrieving methods are provided too by the IBM Content Collector (ICC) [IBM12c]. The retention policies are organized in a *file plan*. Here a structured, functional-based or organizational-based filing schema is defined. It specifies how the records in the system are organized hierarchically [IBM12d].

5.1.3 Specific Use Case Components - the SCCM ECM Stack

For simplicity reasons and time savings, only the basic ECM stack of SCCM was used. A complete deployment on the used VMs would otherwise take too long in the development process when rapidly testing and debugging deployment scripts.

The basic ECM stack comprises the database, user registry, the application server and the ECM software. In this thesis, the *DB2* database, *Tivoli Directory Server* LDAP, *WebSphere application server (WAS)* and *FileNet P8* ECM software with the *IBM Content Navigator (ICN)* were used. ICN is a document management tool that is accessible by a web front-end.

Figure 5.4 shows dependencies between these components being deployed on one target system. All arrows represent a "depends on" relationship while the server icons are only symbolic. There were no further VMs running on the target machine.

The reason for this particular allocation of the components is based on the SCCM components dependencies. TDS needs a local DB2 installation and FileNet P8 with its ICN require an application server they are hosted on.

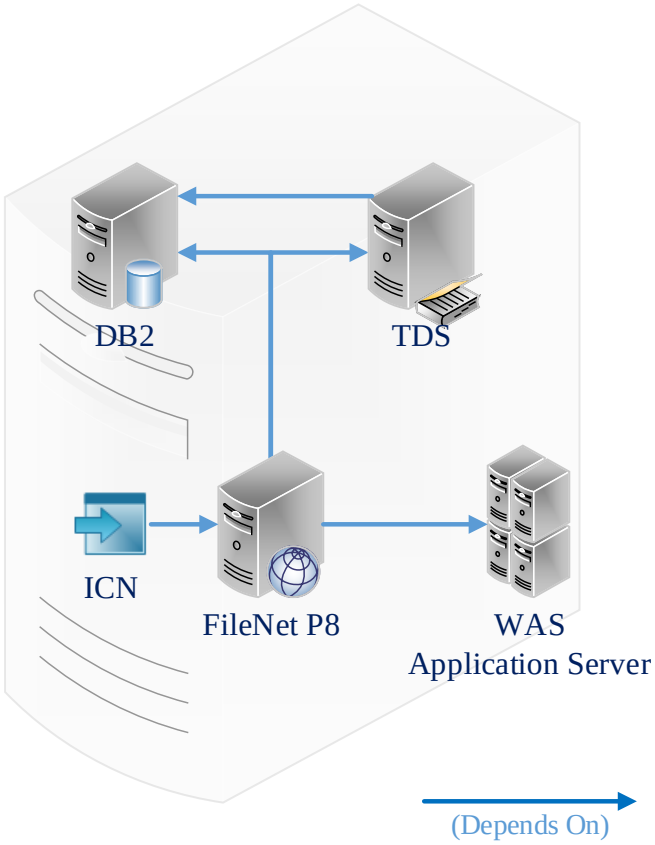


Figure 5.4: SCCM ECM Stack Dependencies on a Target Machine

5.1.4 An OpenTOSCA Model for SCCM ECM

The model follows the concept the same iterative approach that was used when adapting the installation scripts on the VMs.

The first model was a simple *monolithic* deployment where the whole ECM software was logically seen as one Node Type with a *hosted on* relationship between the operating system. Figure 5.5 illustrates that approach. The idea was to extend the model step by step to be able to create a modular Service Template in the end. The visual notations follow the Vino4TOSCA proposal of [BBK+12].

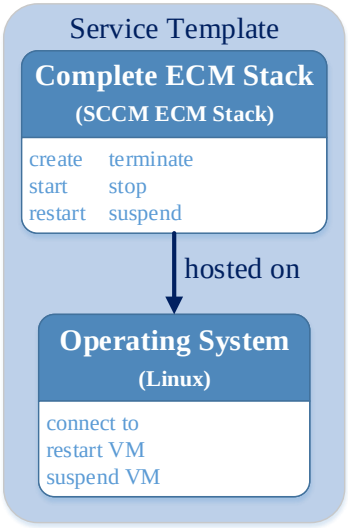


Figure 5.5: Monolithic OpenTOSCA Model

In the next model the components were divided into two nodes. This "packaging" of the actual components allows updating components better later without changing the deployment scripts that much and is closer to an actual working scenario as the Service Template is supposed to be flexible. Separate Node Types were defined with the first one being the operation system. The OS still hosts everything whereas the components are grouped into further Node Types. The Database and User Registry services are consolidated into one Node Type and the Application Server together with the ECM component and ICN form the other. In figure 5.6 this setup is illustrated.

Finally, by taking advantage of the nested Service Templates, the base Service Template consists of the operating system which hosts two Service Templates. The first one contains the ECM system and ICN together with the application server like in the previous grouped model and analogue the database is bundled with the user directory. The difference is that the components inside those Service Templates are defined by their own Node Types, not by an inflexible script that is invoked. Figure 5.7 shows this approach.

The reason for this particular allocation is based on the SCCM equivalent component's dependencies which were shown in figure 5.4.

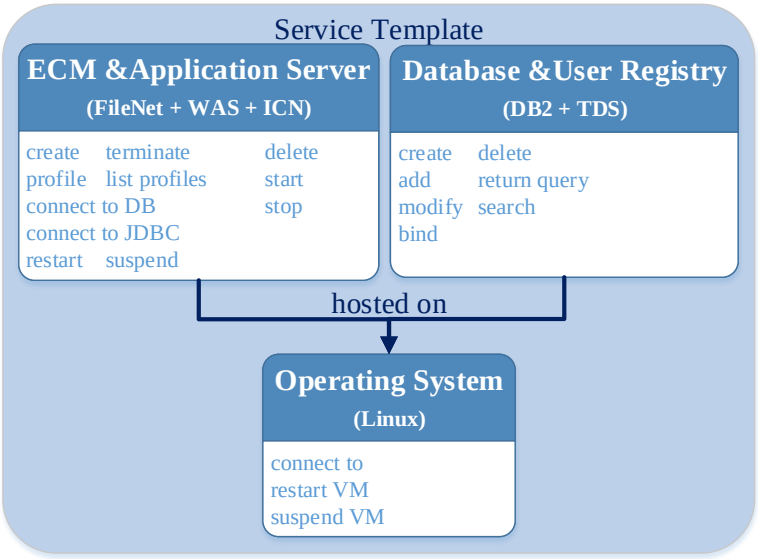


Figure 5.6: OpenTOSCA Model with Consolidated Node Types

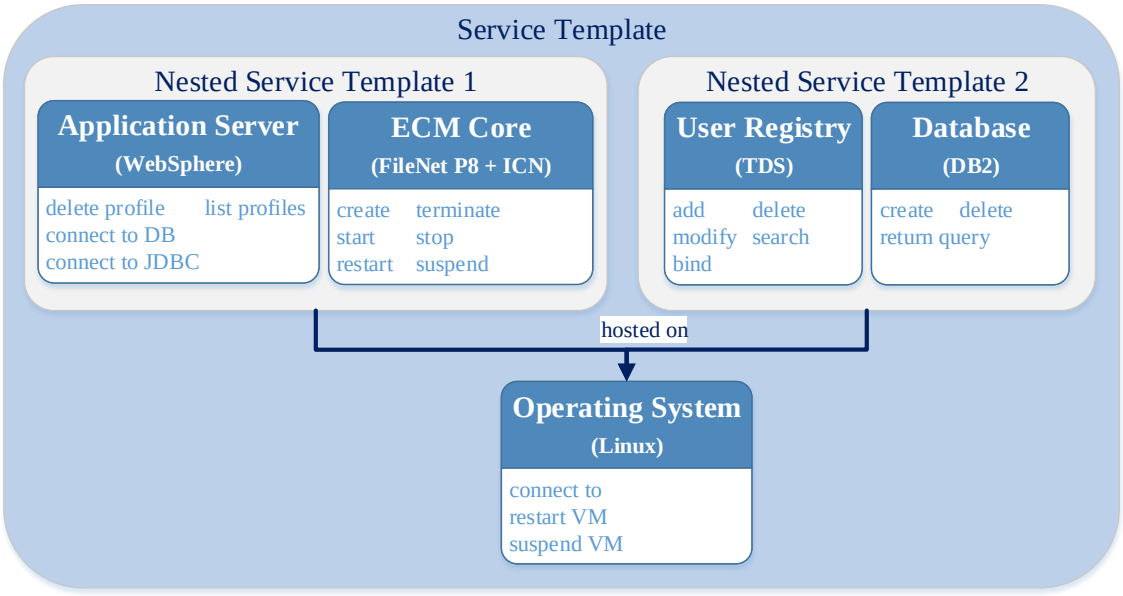


Figure 5.7: OpenTOSCA Model with nested Service Templates

An arbitrary more detailed modeling is possible but the successful implementation of the nested Service Templates model is sufficient to show a proof of concept of the basic TOSCA

standard's modeling concepts. For example, a Service Template could deploy the nodes each on different VMs with their own operating system and connect them via a network.

5.2 Development Approach and Implementation

Like the concept, the development was also done iteratively. After choosing a deployment model the actual modeling was done in eclipse and both the host and target machines were configured for the automatic deployment.

Two virtual machines were provided as development environment for the Service Template. The Red Hat images were pre-installed on the VMs and it is assumed that the deployment and configuration of all IaaS offerings was already performed.

5.2.1 Deployment Model

The monolithic model after figure 5.5 was chosen as it is sufficient to show a proof of concept for OpenTOSCA's functionality in deploying an application. The Service Template thus only contains two Node Templates, one each for the operating system and the complete SCCM ECM core components. This allows developing based on the actual installation scripts for one VM and existing code could be reused.

5.2.2 Operating System

First, a local *virtual machine (VM)* was set up to develop, test and verify the deployment scripts. The virtualization was done on a *Windows 7 Professional* host machine with virtualization software that allows taking snapshots of a VMs current state.

A script pack was provided by IBM, already containing all of the necessary scripts needed to completely deploy a *database*, the *application server* as well as the *ECM* component including an *user interface* on a local machine. Originally the script package was intended to be deployed by an IBM Workload Deployer (IWD) and thus needed to be adapted to the VM deployment by OpenTOSCA.

The script pack mainly consisted of ANT and Python scripts which initiate the deployment on a *Red Hat Enterprise Linux (RHEL)*. The pack was modified by self-written Linux Shell scripts to perform all necessary installation and configuration steps automatically and then re-packed in a new script package.

As RHEL is not freely accessible, an open source Linux was used to develop on in the beginning. Unfortunately, different Linux distributions are not necessarily compatible with each other. Only if a distribution is based on the same kernel version, a possibility exists that packages from different distributions may work with each other.

Fedora, part of the Fedora project owned by Red Hat, was the initial distribution used where the scripts were tested. It is a free community based Linux distribution that provides a modern desktop environment and is based on the Red Hat kernel.

As Fedora encourages a short life cycle and thus tries to keep up with new technologies. It aims to be on the on the forefront when new technology is presented. A short life-cycle allows developers to not concentrate on providing bug fixes and support but instead to create new content and features for a new Fedora version. Because of that, Fedora is not a good platform for product development as changes in the kernel may be profound. Also the short support span of a few months is not realizable in a production environment for any larger company.

It soon showed that in Fedora 18 many lines needed to be changed and altered in the provided scripts to perform an automatic deployment on a local VM. Many features and more recent software versions may not exist on an enterprise Linux distribution. In a company environment, the support needs to be long lasting as any change to the topology can be fatal. That is why only well tested software is used. A better suited Linux distribution for further development was needed and thus the operating system was switched for *CentOS 6.4*.

CentOS is an acronym for *C*ommunity *ENT*erprise *O*perating *S*ystem. As the name suggests, it is community driven and open source. Its goal is to provide a free enterprise level operating system with long lasting support. It is also 100% binary compatible with the *Red Hat Enterprise Linux (RHEL)* distribution. That means that working code running successfully on CentOS will also work in the appropriate RHEL version. As long support spans are necessary and asked for by customers the software used is older and seems to be out of date. UIs may not be have the modern design or are as user friendly as in more recent operating systems. Instead stability, functionality, good documentation and having well proven solutions to problems and questions are prioritized. CentOS' success is reflected in being able to be competitive with Debian which used to be the Linux of choice for a Web Server. CentOS is actually not financially supported by Red Hat and uses the released source code of RHEL to develop its content.

The final target Linux distribution *Red Hat Enterprise Linux* in version 6.3 was pre-installed on VMs in an intranet and the automatic deployment was verified there.

5.2.3 Deployment Development on Local and Intranet VMs

The first step in adapting the scripts was to get them running on a host with a locally installed virtual machine. The host system comprises a *Windows 7 Professional* operating system with virtualization software capable of taking creating, managing and terminating various VMs. By being able to take snapshots of current VM states, rolling back to a working state after entering failure states was simple. Any VM state was saved in a snapshot-tree. This reduces unnecessary redeployment steps that were successfully executed before and allowed focusing on problematic script sections without discouraging any experimental attempts.

Development in a Local Environment

As mentioned, switching to the CentOS 6.4 distribution required fewer adaptions to the original scripts than Fedora 18. Only two files were altered: the `deployTenant.xml` and the `properties file`. The `deployTenant.xml` contains many Ant calls whereas in one line the Java class path needed to be explicitly specified. The properties file contained all information needed to the deployment scripts like administrator names, passwords, repository locations and such. Here, the password and base folder for the ECM application were set. All separate script files were bundled into a `.tar` archive for the planned transmission over the network. As Ant calls were used, an Apache Ant environment was added into the bundle, ensuring that the scripts will be executable on any Linux target environment.

Development for the Intranet Environment

After verifying the automatic deployment on the local VM the next step was to use provided VMs in a secured intranet environment. Two machines were used: one acting as the host machine was set up with a running OpenTOSCA environment and the other acted as the target machine (client). The other machine was the client where only the operating system was installed. For an automatic authentication without entering a password, on the respective machines an asymmetric SSH key pair (RSA keys) was created and registered.

At first, it was assumed that OpenTOSCA was already able to upload the central scripts to the target machine's temporary folder `/tmp`. It needed to be confirmed that the scripts are running flawlessly on the RHEL distribution. The host at this stage acted as an online repository, holding all installers, binaries and images of the ECM stack and the scripts were put into the acting online repository of the host system. Via automatic transfers over SSH by using further scripts, the original script package could be transferred to the target machine. By sending another command over SSH, the deployment process was started by executing the installation scripts remotely on the client and reviewed subsequently.

Figure 5.8 illustrates the approach. Subfigure 5.8 (a) symbolizes running the installation scripts directly on the virtual machine whereas subfigure 5.8 (b) demonstrates that the scripts are first transferred to the target machine and executed there. While the scripts are running another connection is build up to transfer all needed binaries from the host's repository.

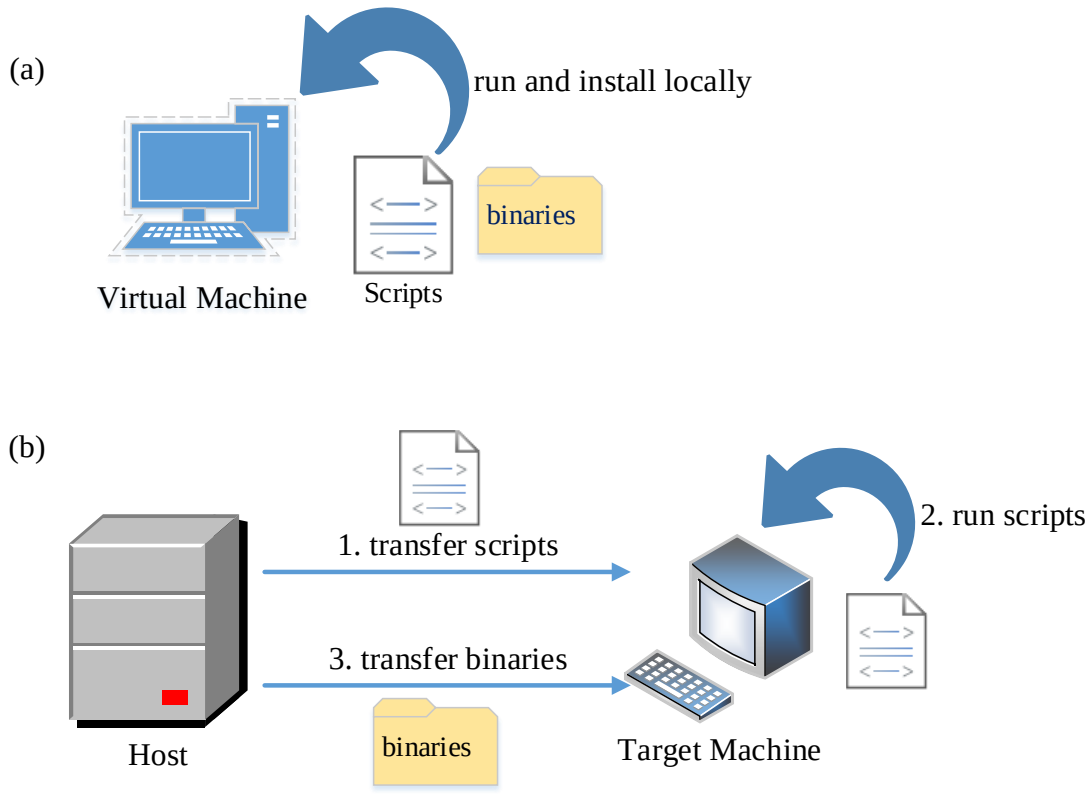


Figure 5.8: Deployment methods on virtual machines

Development for the Container

Finally, a Service Template following the monolithic concept presented in figure 5.5 was created. The Build Plan was done with the *BPEL Designer* plugin for *eclipse* which provided a graphical interface. It is an open source software and though only providing the basic functions allows designing workflow processes.

The created deployment workflow followed a few simple steps. After receiving the credentials, the host logs into the target machine and transferred the small start script package to a target folder on the client. The package is then extracted by sending an extraction command via SSH and finally the start script is called. The start script gets all necessary files from the host's online repository including the main installation script package by an automated login procedure. Eventually, the initial installation script is also automatically invoked. Now the automatic deployment and configuration takes place.

The Build Plan itself is a packed archive inside the THOR file that contains all the necessary BPEL specific files like XSDs, WSDLs and the deployment descriptor. The following figure 5.9 shows the workflow that was created.

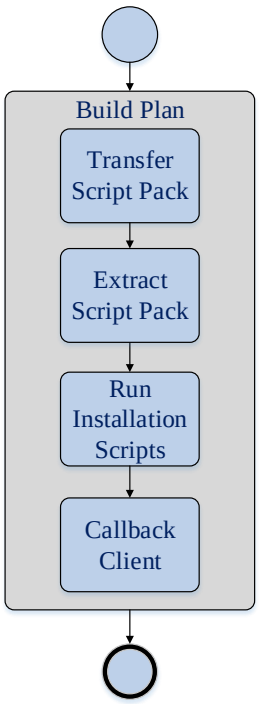


Figure 5.9: The designed BPEL workflow

The start script pack is the only Deployment Artifact and the Implementation Artifact is a Linux service WAR file with all functionalities included to establish the SSH connections and sending commands. After verifying the correctness of the deployment the Build Plan, Service Template, all Artifacts and any other additional documents were packed into a single THOR container file.

This was processed by the OpenTOSCA container and eventually the final deployment was initiated via a SOAP call including the login name and private RSA key for the automatic authentication on the target machine.

A simplified example workflow for a distributed deployment is shown in figure 5.10. Here, the ECM stack is installed and configured on three VMs which are connected via a network. The installation and configuration of the database and LDAP system takes place while the deployment of the application server is performed at the same time. When both are finished the installation of the ECM core begins. Then, all components are configured after receiving the respective endpoint information to allow the correct configuration.

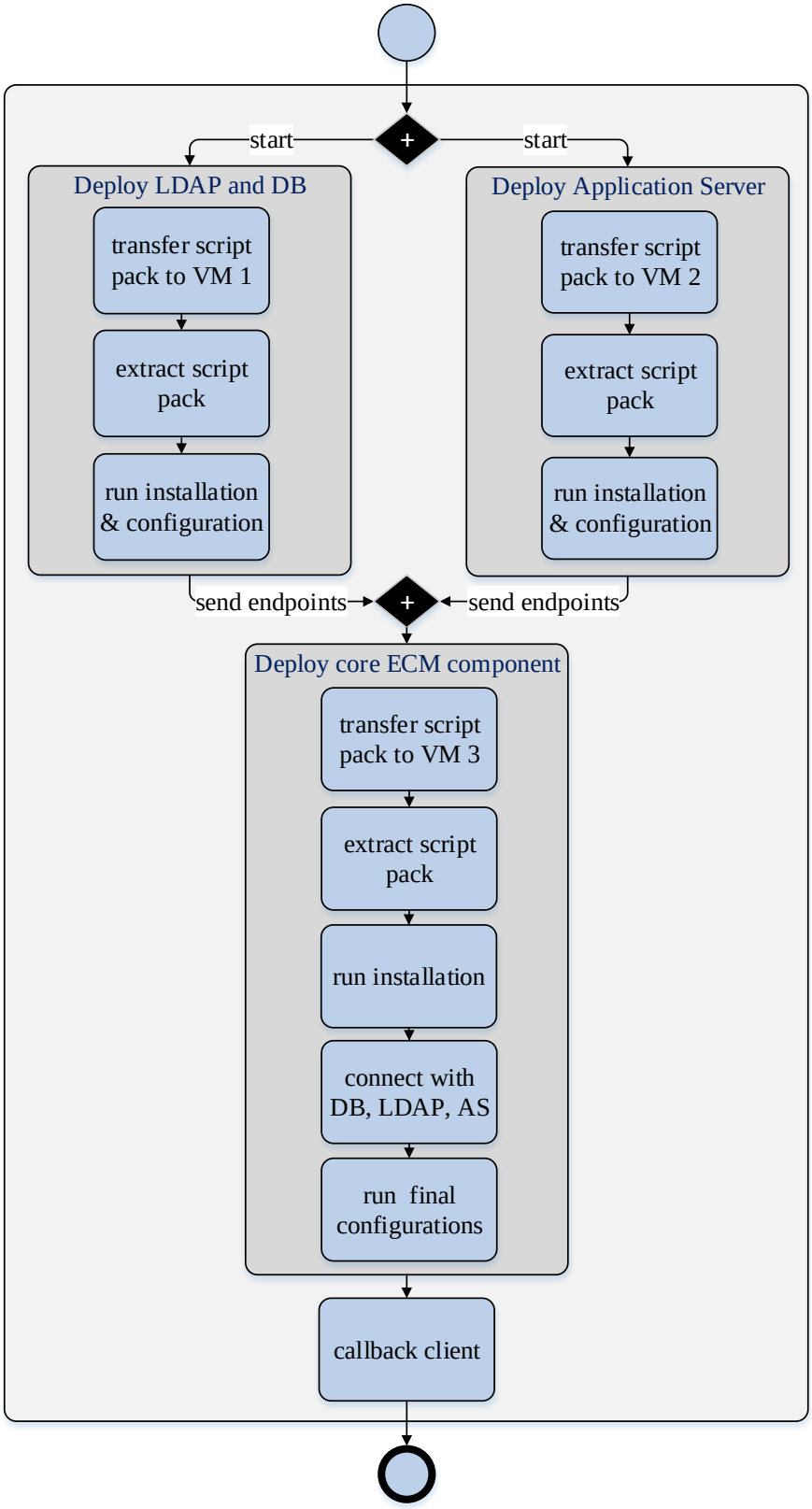


Figure 5.10: A simplified extended workflow for a distributed deployment

5.3 Evaluation

Using an alpha version of a deployment system can naturally lead to problems. First though the concept itself is evaluated and subsequently the implementation, especially OpenTOSCA's capabilities. Any problems are enlightened together with possible solutions to them.

5.3.1 Concept Analysis

By only deploying the basic components, one deployment test could be reduced to about 1.5 hours. Further, by using the monolithic approach the build plan could be kept as simple as possible and is also the simplest deployment method while still being able to verify that OpenTOSCA follows the TOSCA specification.

As the main deployment only relies on imperative scripts not a lot of logic is needed in the container except for dropping the commands and managing the artifacts. These functionalities are provided by WAR files that are invoked through the Build Plan.

The TOSCA concept works but the specification leaves a lot of freedom in any actual implementation. It is not ensured that every container can work with another specific container file as the container itself must be able to support the container files, especially implementation artifacts. This limitation is mentioned in the TOSCA document itself [OAS12].

Implementing the other proposed models shown in figure 5.6 and figure 5.7 can be realized in future iterations. Encapsuling the respective installation scripts into separate packages (e.g. database and user registry in one) is possible. A concurrent installation of the database with the user registry, the application server and ECM application was tested and confirmed as functional during the development in this thesis. The configuration of those components is not as easily performed, as the core ECM application relies on a working state of all other components. Only after the configuration of the database, web application server and user registry are finished, the final configuration steps for the ECM core application can be initiated.

Creating new Node Types and Templates is simple as they are described in XML files. The structure is always the same, allowing for copy-paste code with subsequent adaptions. This makes it relatively efficient to extend the Service Template even without a graphical modeling tool. Unfortunately this leads to a lot of manual programming effort, resulting in having to write code directly into the XML file. The tool support is under development though.

5.3.2 Review of the Implementation

Usability Assessment of OpenTOSCA

The execution approach inside the container file is imperative and the whole logic in how the artifacts are to be deployed is integrated in the sequential Build Plan. As most of the logic is there, the container itself is left with just invoking certain commands.

This focus on the Build Plan and rigid imperative structure creates problems when trying to iterate over it. Basically, it behaves like a script that deploys everything step by step. As the TOSCA standard itself is not complete yet, adaptations to newer versions have to be made in the program to stay conform to the TOSCA standard. There is no support for a declarative approach to describe the topology in OpenTOSCA yet.

As there are no predefined Templates for cloud applications yet, they have to be manually configured. This may take a lot of time but the re-usability of the container is maintained. A working graphical tool is certainly needed.

Currently, besides the OpenTOSCA development there already exist further projects in the *CloudCycle*[1] project which OpenTOSCA is part of. *Vino4TOSCA*[2] was previously mentioned which provides a visual notation for the components. *Valesca4TOSCA* is a graphical tool to design the Nodes Types, Relationship Types, Artifacts Types and Service Templates. It is still under development just like the container but a demo version is available. A working build would speed up the creation of a Service Template with all necessary elements a lot and would also reduce mistakes. It is hard to keep the overview of the whole Service Template, especially if multiple Node and Relationship Types and Templates are defined. Even though the monolithic model was chosen, going through its Service Template is tedious.

A tool or offering that can organize already deployed THOR files would be handy. At the moment the user himself has to manage all deployed artifacts, temporary files and any other residual data. Keeping track what implementation artifact is for which application can get messy. The OpenTOSCA container always deploys an implementation artifact, allowing multiple instances of the same which is just registered on the business process server under a slightly altered name by using a naming scheme.

When deploying the THOR file in the container, the steps for copying the contents, registering the Implementation Artifacts on the web application server and deploying the Build Plan on the business process engine all need to be initiated by separate clicks. There is no other user interaction possible so those steps could have been consolidated to one deployment command. The decision to have multiple separate steps may lie in the fact the web UI was used in a presentation where each step was explained in detail.

There is currently no support for multiple hypervisors. At the moment, it is only possible to communicate with and request a VM from Amazon EC2. Further support for hypervisors can

[1] http://www.cloudcycle.org/ (call date: 2013-05-26)
[2] http://www.vino4tosca.org/ (call date: 2013-05-26)

be added by implementing the respective Implementation Artifacts. This is why this is no limitation in the OpenTOSCA container itself but by missing appropriate IAs.

Performance Assessment of OpenTOSCA

Processing the monolithic THOR container file was fast. Extraction and deployment of the artifacts on the host were quick as only the Service Template was read, the implementation artifacts were registered on the web application server and the Build Plan was deployed on the host's business process server. All data was transferred locally where the execution and processing performance solely depended on the physical or virtual hardware resources available. The execution of the build plan was also fast as it only contained simple commands like the upload of a script to the target machine and the initiation of the automatic deployment a via SOAP call.

On the target machine a more powerful system was needed. The overall performance depended heavily on the application being deployed and the processing power of the target machine. Not only the RAM and CPU power but also the HDD speed was a factor. Even though only the basic components were deployed, one test run took about 1.5 hours, severely slowing down test deployments. The VMs had a 2.5 GHz CPU with 16 GB of RAM assigned to and were located on two IBM System x3755 M2 racks. OpenTOSCA only sends commands and the container only waits until the scripts are finished to proceed. This was a problem as a timeout on the business process server occurred when a script ran too long. A false failure was then detected as the server assumed that the running service crashed.

OpenTOSCA has a good performance for small container files. Tests with larger THORs have to be done to more conclusively be able to give a more accurate overall assessment.

OpenTOSCA Limitations

The alpha v1 build of OpenTOSCA was finished in November 2012 and is based on the working draft 5 (WD5) of the TOSCA specification. The container was developed by a group of students at the University of Stuttgart in a student project. This thesis uses that version while no modifications to its source code were necessary.

The original container requires an internet connection. The XML serializer threw an exception when trying to deploy a THOR file while the VM did not have a working internet connection. To understand that behavior the traffic during the startup of OpenTOSCA was analyzed and debugged. The result was that during the loading process of OpenTOSCA a XSD file was downloaded from an external internet source and copied into the working folder as a configuration file. Without that file, OpenTOSCA cannot function properly when trying to deploy the Implementation Artifacts as that XSD file specifies some XML definitions that are used for the Topology Template processing. The provided virtual machines had only access to the intranet. As that download behavior is hardcoded into the container a local HTTP

server was modified to simulate a working connection to the requested file. Rerouting the DNS lookup to the local HTTP server served as a workaround.

The used business process engine was *WSO2 Business Process Server* in version 2.1.2. In the standard configuration it expected a working external IP address whereas the assigned intranet address was sufficient. For those reasons the network adapter must be active, otherwise a connection to the wso2 server could not be established.

Another limitation of the alpha v1 OpenTOSCA build is that it can only perform three steps: extracting the THOR file, deploying the Implementation Artifacts and deploying the Build Plan on the business process server. Executing the build plan still had to be done manually.

Testing the created Build Plans can be done by using the *SOAP UI* tool or any other software that can send calls to the previously deployed web service's endpoint. The free version of SOAP UI was enough to do all needed tests which comprised of assigning values to the request message and analyzing the results.

Another method could have been the usage of the web UI of the previously deployed business process engine directly. There, a request call to the service's endpoint in question could be sent in the same way as it was done via SOAP UI in theory. Unfortunately, the business process server's web UI interprets line breaks differently as in SOAP UI. During tests, the call over the progress engine's web UI did not work whereas it was consistently successfully executed when using SOAP UI. An analysis of the HTTP server's log files showed that the private RSA key which was sent in plain text and embedded in the SOAP call message was altered by the business process server's web UI. The RSA private key has a fixed format that is not allowed to be changed in any way. As the web UI automatically rearranges the lines the key was always rejected as false and the SOAP call was never successful.

As no working graphical design tool exists for OpenTOSCA yet, XML files and BPEL processes had to be created manually. This resulted in a high amount of effort in compiling a working THOR file and relying and iterating on previously created container files could provide at least some time savings.

Several other restrictions remain as the first version of OpenTOSCA is more of a proof of concept itself. Topology setups are being ignored and all Node Templates in a container are just deployed, regardless of what was actually selected. This was intended by the development team as that functionality was planned to be implemented in a later project.

It is important to specify the exact location within the container if elements are imported. If not, the Core might deliver the wrong file as the internal list is unsorted. It is further not possible to get other assets via download (e.g. HTTP) natively, all needed components have to be included in the respective container folders or downloaded by other means (e.g. scripts). These and other constraints prevent OpenTOSCA to freely combine different Service Templates. It is not clear in some cases which namespaces will be used or which parameters. The same components that are unique in their respective Service Templates may be wrongly deployed multiple times as an OpenTOSCA does not check if the component is already available.

For the Implementation Artifact Engine a running local Tomcat server with an installed Apache Axis2 Web Application was needed. Web services and .war files inside the same THOR must have unique names. Otherwise an already deployed service or .war will be overwritten by the new one. Service names must be the same as Axis2 will otherwise not be able to deploy them, regardless of different filenames.

The Plan Engine can only sequentially process multiple plans while little status information is prompted. Also it is not possible to undeploy already deployed offerings via the OpenTOSCA front-end. Only deployment functionalities are implemented and usable, missing the management and termination part which leads to OpenTOSCA not providing a complete application life cycle support. To undeploy anything the web front-ends of the web application server (Implementation Artifacts) and business process server (Build Plan) had to be used. Removing the extracted temporary THOR file contents also had to be done via the host's local operating system.

The management and termination functionalities need to be added to completely provide all functions that TOSCA suggests. At the moment the OpenTOSCA container is ported to working draft 14 (WD14) which amongst others defines security aspects to keep up with the specification.

5.3.3 Evaluation of the Results

The monolithic approach was successfully modeled, implemented and executed not only by using tools but also by manual programming. The THOR container file was compiled and via OpenTOSCA deployed and the Build Plan contained most of the logic. The container itself only acted as a kind of command deployment system and the file container as a holder of the start script pack. The deployment is not completely automatic as the deployment of the implementation artifacts and the Build Plan must be initiated manually via the web front-end.

Starting the Build Plan via a SOAP call is intentional as the idea is to deploy the THOR container files on the OpenTOSCA host first and the customer can then choose which cloud services he wants to have deployed on the target environment from a larger selection.

Performance-wise the container is quite fast. That may result from the created test THOR file as it had a simple Service Template structure with only two nodes. The deployment artifacts only comprised of one start script that then downloads all other necessary packages from the virtual repository onto the target machine. Essentially, the Build Plan only contained the file transfer of the start script archive and the extraction with the subsequent execution on the target VM was initiated. The wait times during the deployment heavily depend on the software being deployed. Naturally, a complex enterprise offering takes more time to be installed and configured. A thorough performance test has to be performed with multiple use cases and complexities to get a more diverse assessment.

It is not very comfortable to use OpenTOSCA as the usability is not that good. Too much has to be done manually. Not only the creation part but also the execution in the container needs

a lot of non-automated steps. It is not user friendly as the user has to know and send his private RSA key inside an unencrypted SOAP call via another tool while also having to find out the respective endpoint to call. The target machine must have the host's public key stored so that the authentication can even work. Further the deployment steps in the OpenTOSCA container have to be initiated manually.

This gives a very disjunct feeling when using the container. Under the usability aspect, OpenTOSCA needs improvement, both in the UI as well as in the automation aspect. In this version it is only possible to describe the container's functionality as semi-automatic at best. Nevertheless the OpenTOSCA container definitely is a proof of concept, granting a promising outlook for future versions as it provides support for multiple Implementation Artifacts and can be extended in its functionalities.

6 Summary and Future Work

The TOSCA standard is very generic and only describes the general way in how Node Types and Relationship Types are supposed to work with each other. With no concrete directives, different approaches are possible to meet the requirements.

OpenTOSCA is an open source implementation of the TOSCA working draft 5 (WD5) specification. The alpha version serves as a proof of concept and was developed by a group of students at the University of Stuttgart in November 2012. Management and termination functionalities for cloud services are not supported yet, only the deployment capability is implemented. The development of the container is still going on, currently being updated to comply with the working draft 14 (WD14) of TOSCA.

An Enterprise Content Management (ECM) system was to be modeled and then deployed via a Service Template and its necessary assets. These were compiled into a THOR container file which was processed in OpenTOSCA.

After designing different deployment possibilities the monolithic approach was chosen which contained the operating system and the whole ECM stack in a single Node Type. This was sufficient to present a proof of concept that the implementation for TOSCA works and the deployment of a larger enterprise software offering is possible. Only the core ECM components of SCCM were used as these already took 1.5 hours to install and configure. Deploying a more complex architecture or even all SCCM components would have unnecessarily prolonged any test deployments without giving a more insightful result.

As no graphical tool exists yet for creating a Service Template, the XML had to be manually created. This limits the overview and it is hard to keep track of all the components. Imperative scripts were used to automatically install all software components of the ECM stack on the target machine and the Build Plan only transferred, extracted and started those scripts remotely.

The Build Plan itself is a workflow, described in BPEL and deployed on a business process server. Starting the Build Plan had to be done via another tool as the web UI of the business process server did not pass the embedded private RSA key of the host correctly. The private key has a specific format that is not allowed be changed. Otherwise it will be rejected.

Using OpenTOSCA can be tedious as even the execution of the THOR file's contents on the host had to manually be initiated step by step. The overview when having deployed multiple container files can diminish quickly, especially when Implementation Artifacts have the same original name. The container does not check if a particular Implementation Artifact was already deployed before. Instead, OpenTOSCA deploys it again under another deployment name without taking advantage of the already existing component. This is possible as the

deployment name is changed every time by the container before the Artifact is uploaded to the web application server.

No management or termination capabilities are implemented yet. The termination has to be done manually by either using the respective web UIs of the web application and business process server or by deleting the files by hand.

A graphical modeling tool with the name of Valesca4TOSCA is currently under development that allows the definition of Node, Relationship and Artifact Types as well as being able to create a Service Template based on them. A visual notation (Vino4TOSCA) is already proposed providing a unified way of modeling any topology. A customer front-end that shows services the OpenTOSCA container has to offer is being developed. This allows customers to choose and combine cloud offerings to their preference.

Future Work

The CloudCylcle Project

The OpenTOSCA project is part of the CloudCycle Project, together with Vino4TOSCA and Valesca4TOSCA that aims to allow cloud providers to offer their cloud applications easily and cost-efficiently while maintaining service level agreements, guaranteeing security and scalability. There is also an extension to BPMN proposed to support TOSCA-specific elements [KBBL12].

Different parties are involved in the project like IBM, the Fraunhofer Institute or the University of Stuttgart and are supported by the Federal Ministry of Economics and Technology of Germany. It is an open source TOSCA runtime environment (container) that uses a web user interface.

Integration with other Offerings

OpenTOSCA is a work in progress and goes into the right direction. When the complete automatic deployment support is finished, only the management and termination functionalities are missing. The termination can theoretically already be modeled via BPEL to specifically undeploy all installed features. The *Termination Plans* have to be distinguished between ending the instances and disconnecting them from other components, and the undeploy process from the OpenTOSCA environment.

Undeploying would require the Plan to identify and uninstall any Implementation Artifacts and Plans as well as deleting the extracted THOR parts from the host's system. Commands sent over SSH could handle the THOR termination by deleting or editing files directly on the target machine. Interaction possibilities with the APIs of the application and business process servers have to be looked into.

For the management aspect other software can be used. The integration of Chef, Puppet or any other management solution could be investigated. A successful combination would rapidly expand OpenTOSCA's portfolio of offerings as those applications are already under development for quite some time and have a larger user base. An API could be written for each third party software to leverage their functionality.

To remedy the limitation concerning the hypervisor support, OpenStack could fill that gap. Requesting a virtual machine by OpenTOSCA is currently only possible in the Amazon EC2 environment, though directly adding support for hypervisors is possible by implementing the respective Implementation Artifacts. If OpenStack could be integrated too, vast possibilities would be enabled as OpenStack is specialized in providing an infrastructure.

Figure 6.1 shows a draft that uses OpenTOSCA as a framework with an OpenStack core that provides the deployment function. OpenStack could provide and manage the infrastructure while management software like Puppet takes care of the cloud service's maintenance. OpenTOSCA controls all components via an API while in the background other offerings take care of the actual executions. OpenStack interacts with the hypervisor which is located of the container environment.

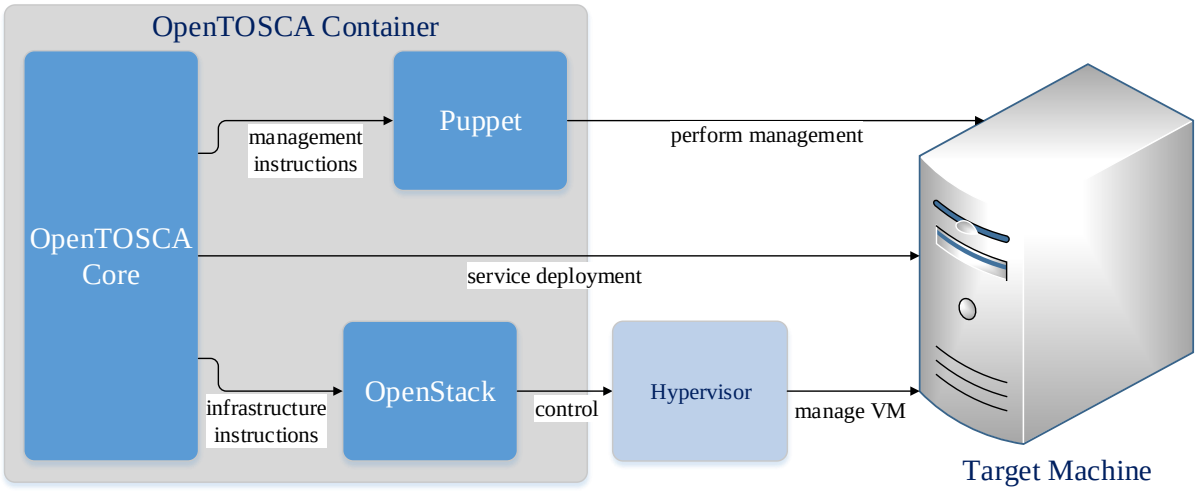


Figure 6.1: Draft of OpenTOSCA integrating third party offerings

Possibilities for Future Improvements of the Model

The topology model can be designed in arbitrary detail. The introduced models in this work are one possible example in how a Service Template can be build. Depending on the deployment framework and the specific ECM software, these designs can vary in various degrees. Figure 5.10 showed a more complex Build Plan that installs the components on three different VMs. It is possible to model every component in its own Node Type with the respective Relationship Types.

This grants the highest degree of flexibility as the choice of which database or which core ECM component is to be deployed is up to the customer. Bundling components into their own Node Types grants standardized software packages. Alternatively, Service Templates can be nested into other Service Templates. The choice between specialization and standardization is then truly depending on the user's preference.

Management Plan functionalities can contain horizontally or vertically scaling aspects to dynamically adjust to a workload change. Maintenance tasks are also definable, which can comprise updating a certain server, suspending or even taking them offline for a while. Having those functions is important in a production environment but are not trivial to implement. Many issues have to be dealt with concerning the overall state and stability of the system or how the traffic is handled during updates or upgrades.

Final Verdict

TOSCA provides many possibilities and OpenTOSCA provides a suitable framework. Many existing concepts could be combined in one application or the functionalities could be developed from scratch. A completely working container still has to be developed and there are other projects running which implement a TOSCA container.

Other projects like OpenStack Heat work on an alternative which will provide a similar management package for cloud services. It has to be seen which particular offering will dominate or if multiple solutions will co-exist like in the case of Puppet and Chef.

TOSCA has the potential to have an important impact on the industry in how cloud offerings are provided. A lot of research still has to be performed and the industry might accept TOSCA if a working implementation is finally developed and released.

Bibliography

[AL11] P. F. Adam Lally. Natural Language Processing With Prolog in the IBM Watson System. *ALP ISSUE*, 2011. URL http://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/.

[Ama04] V. M. M. d. Amaral. *Increasing productivity in high energy physics data mining with a domain specific visual query language*. Ph.D. thesis, Universität Mannheim, 2004. URL http://www.bsz-bw.de/cgi-bin/xvms.cgi?SWB11675449.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996. URL http://ftp.sunet.se/pub/lang/erlang/download/erlang-book-part1.pdf.

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA. In *OTM 2012, Part I*, volume 7565 of *Lecture Notes in Computer Science (LNCS)*, pp. 416–424. Springer-Verlag, 2012.

[BPE07] Web Services Business Process Execution Language Version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/Primer/wsbpel-v2.0-Primer.pdf.

[Cha97] M. Chakravarty. *On the Massively Parallel Execution of Declaraive Programs*. Ph.D. thesis, Technische Universität Berlin, 1997. URL http://www.cse.unsw.edu.au/~chak/papers/diss.ps.gz.

[CIK+12] Z. X. Chen, S. Imazeki, M. Kelm, S. Kofkin-Hansen, Z. Q. Kou, B. McChesney, C. Sadtler. *IBM Workload Deployer: Pattern-based Application and Middleware Deployments in a Private Cloud*. IBM, 2012. URL http://www.redbooks.ibm.com/abstracts/sg248011.html.

[Clo13] CloudCycle. CloudCycle - Innovatives, sicheres und rechtskonformes Cloud Computing. 2013.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970. URL http://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.

[DF08] M. Dalgarno, M. Fowler. UML versus Domain-Specific Languages. In *Methods & Tools - Summer 2008*. Martinig & Associates, 2008. URL http://www.methodsandtools.com/PDF/mt200802.pdf.

[DH06] Y. F. David Harel. *Algorithmik - Die Kunst des Rechnens*. Springer, 2006. URL http://link.springer.com/book/10.1007/3-540-37437-X/page/1.

[Gol12] K. Goldberg. What is Automation. *IEEE Transactionson Automation Science and Engineering*, 9(1), 2012. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6104197.

[HHM+11] D. L. Hakim, A. Hay, M. Mantegazza, P. Piechaczek, S. Mohith, C. Sadtler. *Virtualization with IBM Workload Deployer: Designing and Deploying Virtual Systems*. IBM, 2011. URL http://www.redbooks.ibm.com/abstracts/sg247967.html.

[Hud89] P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 14(3), 1989. URL http://www.dbnet.ece.ntua.gr/~adamo/languages/books/p359-hudak.pdf.

[Hug90] J. Hughes. Why Functional Programming Matters. *Research Topics in Functional Programming*, pp. 17–42, 1990. URL http://www.cs.utexas.edu/~shmat/courses/cs345/whyfp.pdf.

[IBM12a] IBM. *High Level Design Specification - SmartSmart Content Management Release 2.0*, 2012.

[IBM12b] IBM. *IBM SmartSmart Content Management - Administrator Initialization*, 2012.

[IBM12c] IBM. *IBM SmartSmart Content Management - Administrator User's Guide*, 2012.

[IBM12d] IBM. *IBM SmartSmart Content Management - Planning*, 2012.

[Jun04] A. Jung. A short introduction to the Lambda Calculus. 2004. URL http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf.

[Kam06] U. Kampffmeyer. ECM - Enterprise Content Management, 2006. URL http://www.project-consult.net/Files/ECM_White?20Paper_kff_2006.pdf.

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications. In J. Mendling, M. Weidlich, editors, *Business Process Model and Notation*, volume 125 of *Lecture Notes in Business Information Processing*, pp. 38–52. Springer Berlin Heidelberg, 2012. doi:10.1007/978-3-642-33155-8_4.

[lab11] puppet labs. Puppet Enterprise - IT Automation Software for System Administrators to Discover, Configure, and Manage Infrastructure, 2011. URL https://puppetlabs.com/wp-content/uploads/2011/06/puppet-enterprise-product-brief.pdf.

[Llo94] J. W. Lloyd. Practical Advantages of Declarative Programming. 1994. URL http://www.cs.chalmers.se/pub/users/oloft/Papers/wm96/wm96.html.

[MFH+12] J. Miszczyk, R. Ferdous, G. Hurlebaus, J. Jacobson, L.-F. Lee, D. Peraza, K. Staples. *Creating Smart Virtual Appliances with the IBM Image Construction and Composition Tool*. IBM, 2012. URL http://www.redbooks.ibm.com/redbooks/pdfs/sg248042.pdf.

[Nat11] National Institute of Standards and Technology. *The NIST Definition of Cloud Computing*, 2011. URL http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[Nat12] National Institute of Standards and Technology. *Cloud Computing Synopsis and Recommendations*, 2012. URL http://csrc.nist.gov/publications/nistpubs/800-146/sp800-146.pdf.

[Neb12] M. Nebel. *Formale Grundlagen der Programmierung*. Springer, 2012. URL http://link.springer.com/book/10.1007/978-3-8348-2296-3/page/1.

[Neu94] B. C. Neuman. Scale in Distributed Systems. In *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994. URL https://cs.uwaterloo.ca/~brecht/courses/854-Scalable-Systems-Software/Possible-Readings/general/scale-dist-sys-neuman-readings-dcs-1994.pdf.

[OAS12] OASIS TOSCA TC. Topology and Orchestration Specification for Cloud Applications - working draft 05, 2012. URL https://www.oasis-open.org/committees/document.php?document_id=45638&wg_abbrev=tosca.

[Ope13a] OpenStack Foundation. OpenStack Basic Installation Guide, 2013. URL http://docs.openstack.org/folsom/basic-install/content/basic-install-folsom.pdf.

[Ope13b] OpenStack Foundation. *OpenStack Compute Administration Guide*, 2013. URL http://docs.openstack.org/trunk/openstack-compute/admin/bk-compute-adminguide-trunk.pdf.

[SDE12] VMware vCloud Networking and Security Overview. Technical report, VMware, 2012.

[Spi01] D. Spinellis. Notable Design Patterns for Domain Specific Languages. *Journal of Systems and Software*, 56(1):91–99, 2001. doi:10.1016/S0164-1212(00)00089-3. URL http://www.spinellis.gr/pubs/jrnl/2000-JSS-DSLPatterns/html/dslpat.html.

[Sri01] K. Srinivasan. The Evolving Role of Automation in Intel Microprocessor Development and Manufacturing. *Intel Technology Journal Q3*, 2001. URL http://130.203.133.150/viewdoc/download;jsessionid=2CFF8288AED69ED9E3FD3C2A7C152EB5?doi=10.1.1.217.546&rep=rep1&type=pdf.

[Ste11] M. Stevens. Programming paradigms and an overview of C - COMP3610 - Principles of Programming Languages, 2011. URL http://cs.anu.edu.au/student/comp3610/lectures/Paradigms.pdf.

[TOS13] Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0, 2013. URL http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf.

[vK76] M. van Emden, R. Kowalski. The semantics of predicate logic as a programming language. *ACM*, vol. 23:pp 733–742, 1976. URL `http://www.doc.ic.ac.uk/~rak/papers/kowalski-van_emden.pdf`.

[Wag11] F. O. Wagner. *A Virtualization Approach to Scalable Enterprise Content Management*. Ph.D. thesis, University of Stuttgart, 2011.

All links were last followed on May 26, 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Kai Liu)