Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Constraint-Based Automated Updating of Application Deployment Models

Felix Burk

Course of Study:	Software Engineering
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Lukas Harzenetter, M.Sc.
Commenced:	May 6, 2019
Completed:	November 6, 2019

Abstract

Cloud applications in research and in the industry can be modelled with application deployment models to increase maintainability and to deploy them automatically. The modelling process requires domain specific knowledge about each component used inside the application, such as a Java application or an operating system, for example, Ubuntu. Furthermore, frequent update schedules of these components force developers to regularly update their deployment models to satisfy compliance rules. The process is error prone, as updating a single component in a cloud application deployment model can break the whole model and its deployment process because updated components may not be able to work properly with other components already in use. This thesis presents an approach that enables developers to automatically update deployment models by formulating constraints that ensure those models are deployable and define which component versions can work together.

The approach described in the thesis uses artificial intelligence planning to automatically detect such problems and update components as needed. The updating process is realized by transforming deployment models into a Planning Domain Definition Language problem which is solved by a planner. The resulting plan is generated and describes how the model can be updated in multiple ways. The plan is then applied to the model and the updated model is presented to the cloud application developer. Constraints are defined which assert that the model is deployable and that each desired update of a component takes place. A prototype is presented which implements the approach shown and can be included in the modelling workflow of cloud application developers.

Contents

List of Figures

List of Listings

Acronyms

CSP Constraint Satisfaction Problem. 15, 17, 18, 19, 20, 23, 31, 32, 45, 46

DBMS Database Management System. 16, 17, 43

EDMM Essential Deployment Meta Model. 15, 16, 25, 29, 30, 31, 32, 41, 42, 43, 45

FOL First Order Logic. 19, 20, 21, 25, 26, 35, 43

HTN Hierarchical Task Network. 27

IPC International Planning Competition. 42

PDDL Planning Domain Definition Language. 3, 20, 21, 29, 31, 32, 33, 35, 39, 43

TOSCA Topology Orchestration Specification for Cloud Applications. 13, 14, 15, 16, 25, 26, 29, 31, 35, 41, 42, 45, 46

1 Introduction

As technology advances further every year, so do applications and their complexity. Increasing demands from users and companies require more sophisticated platforms that need to be built and maintained. Researchers develop a wide variety of prototypes based on different approaches to fulfill some of these demands or to show how such systems can be built. They need to use prototypes even years after they have been built, which means that prototypes need to be accessible for long periods of time. But as technology evolves, prototypes age and complications arise, such as outdated components inside the application itself. Frequent update schedules of components inside the prototypes force researchers to regularly change prototypes to keep them in a functional and secure state [SSS+12]. This occurs frequently in the digital humanities where scientific systems need to function for years to present digital results. The SustainLife [NHS+18] project examines solutions to ease maintenance, increase sustainability and stability for such applications. The project uses deployment models which enables researchers to automatically deploy applications in the cloud.

Applications are expressed in a deployment model, these models reflect versions of application components such as Ubuntu version 12.04 [Har18]. They ease the maintenance, the deployment process and reduce the necessary knowledge of specific domains that would otherwise be needed by maintainers or researchers [SBKL19]. This includes how to install certain parts of an application like a Java virtual machine and how to maintain such a system as a whole. In recent years, several standards emerged that provide these features, such as the Topology Orchestration Specification for Cloud Applications standard by OASIS [TOSCA-v1.0], which is used by the SustainLife project. These standards ease the development of such complex systems. Some of them include a graph to model components in applications. The connections of the graph express dependencies between components, e.g., a Java runtime depends on an operating system such as Ubuntu to operate on. By modelling these topology graphs, simple and complex applications can be constructed. Several works have been published which ease the modelling process, such as validation of TOSCA deployment models by Brogi et al. [BTS18]. Furthermore, efforts by Saatkamp et. al [SBKL19] include detecting problems by using architecture and design patterns.

However, these systems need to be adapted constantly to satisfy security assessments and to provide newly developed features to their users. If a component inside a topology graph is changed to a newer version, its dependencies change, e.g., updating a Java application can lead to the problem that the currently used Java runtime does not support features used in the new application. Subsequently, the Java runtime has to be updated as well. Such cascading updates happen frequently in complex deployment models, as many components depend on each other. If a single one is changed, several others have to be changed as well. Changing the graph manually requires expertise in all components and their corresponding versions involved in the topology graph. Furthermore, this process is error prone, as explained by Saatkamp et al. [SBKL19]. Automatic updates are a common problem in many domains such as package managers in operating systems or programming languages. But there is no automatic way to update cloud deployment models as of now. This thesis aims to solve the problem by explaining an approach which lets application developers automatically update topologies during the modelling phase. The approach is based on planning which is already used in the deployment process itself, as described by Breitenbücher et. al [Bre16]. By instructing a planner to update topology graphs under certain constraints, automatic updates can be achieved. Furthermore, a prototype is presented which can be used by developers and researchers to update cloud deployment models encoded with commonly used standards, such as TOSCA, Chef, or Puppet.

This thesis begins by introducing necessary background information, including deployment models and planning in Chapter 2. Afterwards, related works are presented in Chapter 3. The approach which enables automatic updating of deployment models is presented in Chapter 4. The prototype and how it can be used is explained in Chapter 5. Lastly, the summary is presented and future work is discussed.

2 Background and Fundamentals

This chapter introduces key technologies involved in solving the problem of automatically updating deployment models. It begins by explaining deployment models and the Essential Deployment Meta Model which enables the usage of commonly used deployment model standards. As update problems can be solved by formulating them as a Constraint Satisfaction Problem, the concept is explained in Section 2.2. Planning is illustrated in Section 2.3, it includes a description of a commonly used planning language which will be used in the approach in later chapters. Lastly, the overall problem of dependency resolution is shown in Section 2.4.

2.1 Deployment Models

Technologies that automatically deploy applications require an initial modelling phase which is handled by the user. After the modelling phase the deployment itself takes place. There are two approaches on how to model such applications. Declarative deployment models only describe the state in which all components of an application are after deployment. This includes connections between components inside the application. A deployment engine then enforces the state [EBF+17]. Imperative models do describe a sequence of actions, they deploy the whole application and each component inside it, until the desired state is reached. They explicitly state how to reach the desired state of the application [EBF+17]. Wurster et al. [WBF+19] analyzed different declarative deployment models and defined the Essential Deployment Meta Model. It encompasses essential parts of popular declarative deployment models in a single model. Subsequently, models as defined by technologies such as Puppet, Chef or TOSCA can be mapped to EDMM. Furthermore, a direct comparison between models is possible. The approach presented in this thesis uses EDMM to support a variety of deployment models.

Components in declarative deployment models can be expressed in graph, where connections signify dependencies between them. An *application topology* is a directed, typed graph. It represents the structure of an application by expressing application components as nodes and dependencies as connections. These graphs help modelling existing and future applications, their technical components, and the dependencies between them. Figure 2.1 shows such an application topology. EDMM defines *Model Entities* which include *Components* and *Relations*. They are both instances created from a type, either from a *Component Type* or from a *Relation Type*. They define the semantics of each Component and Relation, e.g., Figure 2.1 shows a Component, an instance of type Ubuntu. The connection between the leftmost Ubuntu Component and Java 6 is of type *depdendsOn*. Figure 4.2 shows terms used in EDMM and how they relate to another. Further Model Entities include *Properties* and *Operations*, they are not part of the application topology. Component Types have Operations which enable algorithms to deploy and maintain them. Artifacts implement Operations or Components. But properties declare a state of a Model Entity. A Property of Java 6 may be the amount of RAM it has at its disposal and an Operation could be an install script executed on the Ubuntu Component that installs the Java 6 Component. Component Types can be used to express versions, as shown by Harzenetter [Har18]. However, they are part of a type hierarchy, for example, the Component Type Ubuntu 12.04 has a parent type which defines the semantics of all Ubuntu operating systems, called Ubuntu.

However, to model dependencies accurately for the planner, additional information has to be provided which is not included in the Essential Deployment Meta Model. *Requirements* and *Capabilities* are used in the Topology Orchestration Specification for Cloud Applications standard by OASIS [TOSCA-v1.0]. These definitions will be used in the approach presented in Chapter 4 to encode the versions fitting to another. They are shown next to the Components in Figure 2.1. A Requirement signifies the need of another Component to function properly. Requirements describe a dependency, as defined in [SCEE17]. For example, Java 6 is dependent on Ubuntu 12.04, subsequently Java 6 has a Requirement that is resolved by the Capability of Ubuntu. If a Requirement and Capability does match, a valid connection is reached. If all Requirements of a Component are resolved, it can be deployed. Therefore all Requirements have to be resolved after updating deployment models. The approach explained in this thesis uses the Requirement and Capability concepts of TOSCA to enhance the existing Essential Deployment Meta Model. The TOSCA standard refers to Components in a deployment model as *Node Templates*, their respective types are called *Node Types*.

In principle, topologies can represent the structure of any application on different abstraction levels [CDE+13]. The graphs can model cloud applications or even dependency graphs between libraries used in a programming language. In this thesis all examples will be limited to cloud applications. The granularity level will be the same as Component Types shown in the running example Figure 2.1. A motivating scenario of an outdated deployment model is given in the next section, it will be used as a running example throughout this thesis.

2.1.1 Motivating Scenario

An example of an outdated deployment model is given in Figure 2.1. It will be used to illustrate the problems faced when updating a deployment model. Concepts and the approach in Chapter 4 will be explained by the means of this particular example. Figure 2.1 shows an application topology, that is part of the deployment model. This example models a cloud application, as its abstraction level shows software which has to be deployed to gain a functioning environment operating in the cloud. The figure contains several nodes that act as Components of the deployment model. Directed edges visualize dependencies between them. They are illustrated with *Requirements* and *Capabilities*. A source node depends directly on all Components, connected to its Requirements.

Each of the nodes has a name according to its Component Type. This means the Component Ubuntu 12.04 represents a regular Ubuntu[1] operating system. The Components Tomcat 7[2] and Java 6[3] depend on the left Ubuntu Component. Furthermore, the Java Webshop 1.0.0 has a dependency on the Components Tomcat 7 and MySQL-DB 5.1[4]. On the right side the dependency between MySQL DB and a Database Management System is visible. The last connection is between the

[1] https://ubuntu.com/
[2] https://tomcat.apache.org/
[3] https://www.java.com/en/
[4] https://www.mysql.com/

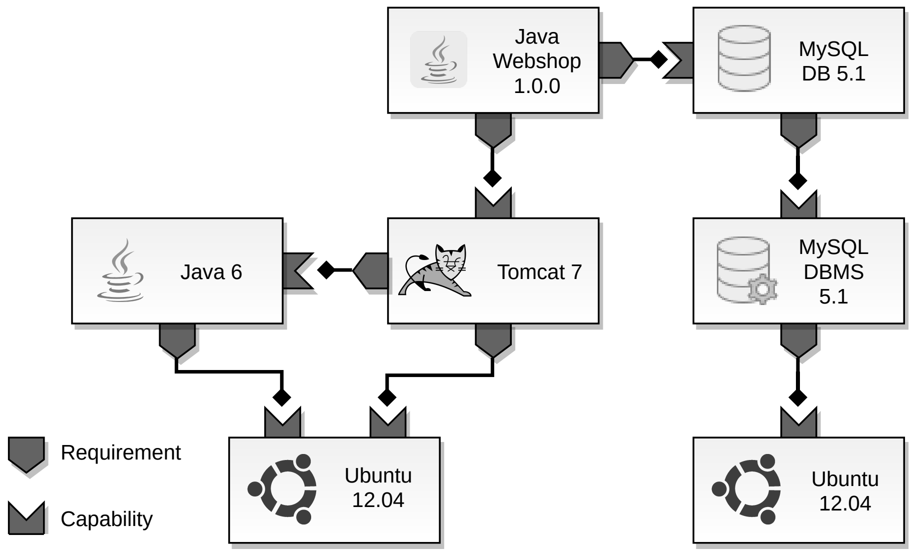


Figure 2.1: The figure shows the topology of an outdated application deployment model. It is used to deploy this particular cloud application.

DBMS Component and another Ubuntu operating system of version 12.04. Additionally, nodes contain their respective version number, as the Component Type describes a particular version. Every connection consists of a Requirement and a Capability. The source Component has the Requirement which is fulfilled by the Capability of the sink.

This deployment model is severely outdated. Version 12.04 reached its end of life in April 2017 [Cana], the general extended support of Java 6 ended in December 2018 [Ora]. A newer version of the Ubuntu operating system is version 18.04 [5]. But this version does not contain the necessary Capabilities for the Java 6 Component. Subsequently, Java has to be updated as well, Java version 8 is compatible with Ubuntu version 18.04 [Canc]. After replacing those three Components, all other Requirements and Capabilities have to be validated. The updated Ubuntu version 18.04 does not support MySQL DBMS version 5.1, another update has to be made. MySQL DBMS is updated to version 5.7 and the MySQL DB Component to the same version as well. The new DBMS version is supported by Ubuntu version 18.04 [Canb].

After updating the two Ubuntu Components, two connections broke. The connection with the Java 6 Component and with the MySQL DBMS. After they were updated, another connection broke, between the MySQL DBMS Component and the MySQL database another update had to be made. Five Components had to be updated in total to reach a deployable model with a newer version of Ubuntu. This example illustrates the cascading updated problem. After updating one or multiple Components, several connected Components had to be updated as well. The updated model can now be deployed, newer versions ensure security Requirements. To reach valid connections after the updating process, the topology, its Components, and connections can be modelled as a Constraint Satisfaction Problem, which will be explained in the next section.

[5]http://releases.ubuntu.com/18.04/

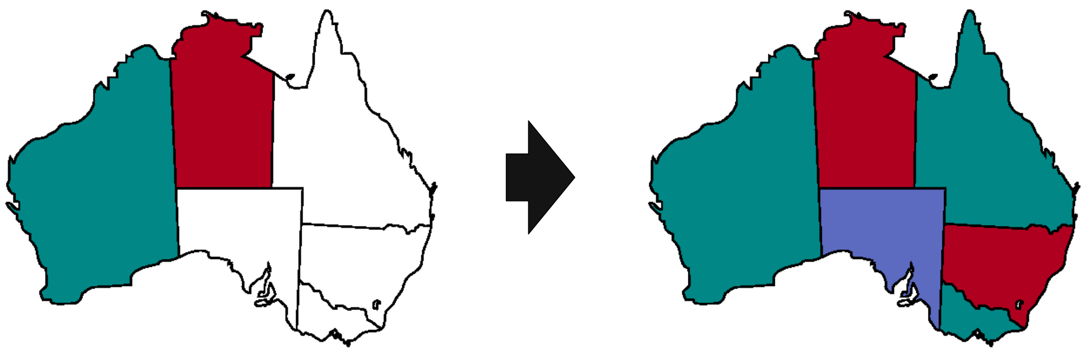


Figure 2.2: An example of a map coloring problem. Each section has to be filled with a color. However, no adjacent regions can have the same color. The initial state is on the left side, the solution can be seen on the right side[6].

2.2 Constraint Satisfaction Problems

The previous section described a motivating scenario where five Components had to be updated to gain a deployable, updated application deployment model. An algorithm automatically updates this model, it has to observe which Component versions reach a valid connection. To express invalid connections, constraints can be used. By following the rules of constraints, an algorithm is able to reach a valid solution. Such problems can be expressed as Constraint Satisfaction Problems. They describe a set of variables that contain a single value from a given domain. However, the values of variables can not be chosen arbitrarily from the domain. They underlie constraints, which reflect properties of the problem to solve. If all constraints are fulfilled, the solution has been found.

An example of a Constraint Satisfaction Problems problem is a map coloring problem, shown in Figure 2.2. The task demands that all regions need to have assigned a color from a domain of three colors, green, blue, and red. However, adjacent regions can not have the same color, as shown in the solution in Figure 2.2. When expressing this task as a Constraint Satisfaction Problem, each region is assigned a variable containing the color. The constraints assert that each adjacent region pairs do not share the same color. Algorithms which have the task of solving such a problem need to consider all constraints to solve it. They limit the domain for a single variable, e.g., if the leftmost region is filled with the color green, as shown in Figure 2.2, the two regions adjacent to the green region can not have the same color. Their possible domain does not include the color green anymore. This simple example only has 3^6 possible states, but as the domain and the number of regions grow so do the possible states. The running example Figure 2.1 has 7 Components, if the number of possible Component Types is 20 there are 7^{20} possible states of which only a subset is a deployable model. To solve CSPs efficiently, algorithms have to be used that exploit constraints to find a solution much faster than conventional brute force solving.

The map can be expressed as a graph, shown in Figure 2.3. Connections between the graph describe adjacent regions of the map. Constraints define which node connections are valid depending on the color they have in the current state. The resulting graph is similar to an application topology graph

[6]`https://commons.wikimedia.org/wiki/Maps_of_Australia#/media/File:Australia_locator-MJC.png` CC BY-SA 3.0, last viewed November 4, 2019

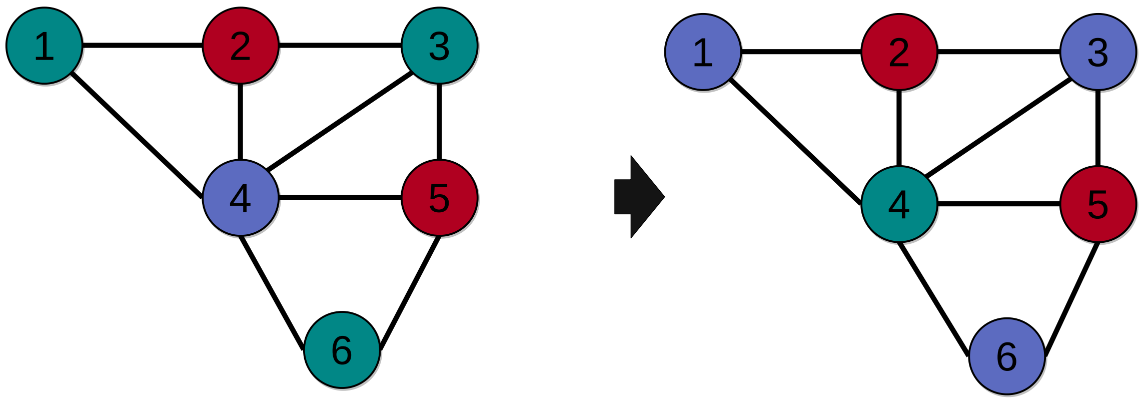


Figure 2.3: The left graph shows the solution to the coloring problem of Figure 2.2. On the right side is another solution, that can be created by changing the left graph while preserving all constraints.

in a deployment model, by replacing colors with Component Types and nodes with Components. Dependencies are encoded as matching Requirements and Capabilities in constraints. Only if all dependencies are resolved a solution has been found. Therefore automatic updating of deployment models can be achieved by encoding matching Requirements and Capabilities and the topology of an application deployment model to a Constraint Satisfaction Problem.

For solving the updating problem planning is used, as planning problems can be converted to constraint satisfaction problems [FS92]. The next section will explain how planning works by using the same graph coloring problem, as shown in figure Figure 2.3.

2.3 Planning

The word *planning* is a term used to describe the act of finding a plan, to achieve a desired goal [RN09]. A plan consists of a sequence of actions, which change an initial state until the goal state is reached [RN09]. For example, a sequence of actions to change the coloring of the graph shown in Figure 2.3, preserves the constraints that each pair of connected nodes can not have the same color in the goal state. *Planners* are dedicated programs that derive a sequence of predefined actions to reach a defined goal state from a given initial state. The initial state is encoded as a problem, actions and the goal are defined in a domain. Both of them are formulated using First Order Logic, which will be explained in Section 2.3.1. From this information pool a planner is able to derive a sequence of actions to achieve the goal by altering the initial state with the provided actions until the goal state is reached.

In the last section CSPs were introduced with the map coloring problem, as pictured in Figure 2.2. By mapping each district to a node and each adjacent district pair to a connection, the problem can be expressed in a graph, where each node has an assigned color value from the domain space. Two of these graphs can be seen in Figure 2.3. The graph on the left shows one of the possible solutions to the map coloring problem. The task is now to change the color of node 4 to the color green while considering the constraint that every pair of connected nodes can not have the same color. However, node 1, node 3, and node 6 are connected to node 4 and are green, subsequently they have to be changed to reach the desired goal. A planner has to find a sequence of actions which change the graph as pictured on the left side to the graph shown on the right side in Figure 2.3. Therefore the initial state is the graph on the left side. The goal state contains the information that node number 4 needs to have the color green. Only a single action is needed to solve problems of this type, changing the color of a node.

However, to prevent a planner from taking an action violating the constraints of the graph, to goal state has to be used. The goal state has to contain the constraint that each connected node pair can not have the same color. Preconditions of actions have a similar effect, they prevent the planner from taking an action if those conditions are not met in the current state. But as we are only interested in the goal state, preconditions are not needed in the current example. Actions also consist of parameters and an effect. The parameters for the `change_color` action in the example is the node and the color which is assigned to it. The effect then has to encode all changes to the state after the action has been taken.

The colored graph is similar to the topology inside the deployment model which should be updated. Furthermore, the coloring problem is related to the problem of updating deployment models. Nodes can be seen as Components whereas colors function as Component Types that can be changed. However, as Chapter 4 will show, some additional actions and constraints have to be considered. The graph coloring problem can be solved as a planning problem or as a CSP. The planning approach was chosen because planning has already been used in deployment models, which will be shown in Section 3.3. Furthermore, planning problems can be formulated as a Constraint Satisfaction Problem by mapping actions to CSP variables, while adding preconditions, effects, the initial state, and the goal state as constraints [DK00]. Therefore planning problems can be solved by using algorithms which solve Constraint Satisfaction Problems. The next section shows how planning problems are encoded to be read by a planner.

2.3.1 PDDL

Most Planners use the Planning Domain Definition Language, first designed in 1998 in an effort to standardize planning languages [GHK+98]. It uses First Order Logic and a standardized syntax to formulate the initial state, the goal state, and actions in a consistent manner. PDDL is a declarative language [Ede04], it does not tell the planner how to solve the problem, only the initial state, actions and the goal state are given to a planner. Both states are stored in a problem file and actions to change the state in a domain file. This ensures reusability of similar problems with different initial states or goals.

The domain file begins with the domain name, followed by requirements needed to correctly parse both the problem and the domain. Domains were introduced to enable different planners to focus on supporting different aspects of the PDDL language, as a planner that implements all requirements would be complex and can not be adapted to specific subsets of the PDDL language which can lead to a faster computation time. Only few planners should support every requirement as described in the initial specification [GHK+98]. Therefore dedicated planners can focus on supporting a subset of the PDDL language and do not have to be as complex as otherwise necessary, furthermore they can be optimized for specific subsets.

Listing 2.1 Change color action example

```
(:action change_color
 :parameters (?n - node ?c - color)
 :precondition (not (has_color ?n ?c))
 :effect (has_color ?n ?c))

```

Listing 2.2 Prevent adjacent colors derived predicate example

```
1 (:derived (prevent_adjacent_colors)
2   (forall (?n0 - node)
3       (forall (?n1 - node)
4           (or (not(connected_with ?n))
5               (not(exists(?c - color)
6                       (and(has_color ?n0 ?c)
7                           (has_color ?n1 ?c)))))))
8
```

The next section of the domain file specifies object types, used by variables. For example, the object `node_four - node` of type `node`, whose name is `node_four`. Subsequently, constants and predicates can be declared. Predicates are functions with parameters which can be evaluated to either true or false. All predicates not contained in the problem are false. Predicates can also be used to model the goal state or to assert the effect of an action, as shown in Listing 2.1. They are used with variables of corresponding types, e.g., `(change_color ?n - node ?c - color)` declares the predicate `(change_color node_four green)`, where the objects `node_four` and `green` of type `node` and `color` are used.

From now on, predicates in examples are denoted inline as `<p>`. PDDL is written in First Order Logic, which means predicates such as `(and <p> <p>)`,`(or <p> <p>)` and `(not <p>)` are predefined and can be used to conjoin, disjoin, or negate other predicates. Further predicates such as `(forall (?v - type)(<p>))` or `(exists (?v - type)(<p>))` can be added by Requirements. The `forall` predicate is a shortcut to assert all predicates inside it true for all objects of the type specified, it is a universal quantifier. The `exists` predicate asserts that at least one of the objects, of the type specified has to fulfill the predicates inside the `exists` predicate, it acts as an existential quantor.

Furthermore, derived predicates will be declared like conventional predicates. They are used to enhance the expressiveness of PDDL and help in efficiency, as stated in [THN05]. A derived predicate can contain several connected predicates, as shown in Listing 2.2. Asserting the example predicate in the goal state enforces the rule of the example given in the previous section, no connected nodes can have the same color. This is achieved by using two nested `forall` predicates which assert that all node pairs are either not connected (Line 4) or they are connected but there does not exist (Line 5) a color that both nodes have in common (Line 6).

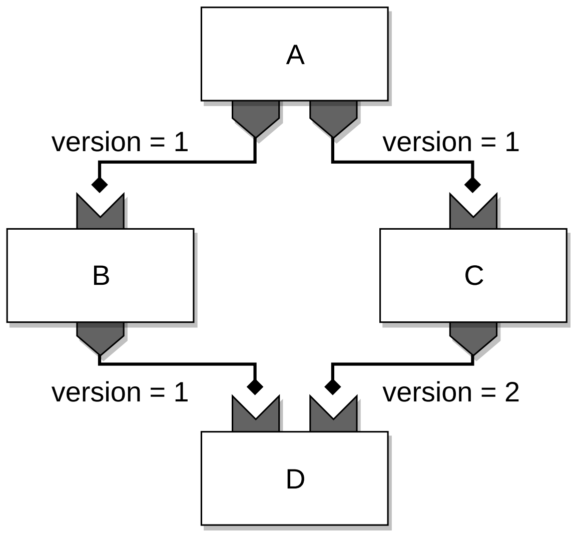


Figure 2.4: The diamond dependency problem includes four Components, where Component A depends on Component B and C, that depend on Component D. The edges refer to the requested version specified by the source. A conflict can be seen between the version requested by Component B and C.

2.4 Dependency Resolution

The Problem of Dependency Resolution occurs in domains, such as package managers in operating systems, e.g., Aptitude[7]. However, they are not only used in operating systems. Several programming languages use package managers to reliably install third party software Components. Examples of these are Python's pip[8] and Ruby's Bundler[9]. Another example is Dependency Resolving in Eclipse's Plugin architecture. The project is called p2[10]. All of them install different kinds of software packages, which from now on are referred to as Components to generalize the problem.

Components depend on each other, e.g., the Java Webshop shown in Figure 2.1 needs a Java runtime to operate. This quickly leads to transitive dependencies across a majority of them. Because a single Component can depend on several others, which in turn will depend on a multitude of other Components, leading to an exponential increase. Package managers have to find a plan that fulfils the requirements of all Components.

Furthermore, most package managers allow programmers to specify a version (or sometimes version ranges), their Components depend on. This leads to a quite difficult problem, as several Components can depend on the same software package but can specify different versions of them. The diamond dependency problem shows this dilemma, as illustrated in Figure 2.4. The figure shows 4 components, the uppermost component A depends on Component B and C, which both depend on a single Component, called D. The connections show the requested version of each

[7]https://wiki.debian.org/Aptitude
[8]https://pypi.org/project/pip/
[9]https://bundler.io/
[10]https://www.eclipse.org/equinox/p2/

connection source, e.g., component A expects version 1 from Component B to function properly. Figure 2.4 shows that Component B needs version 1 from Component D to function, but Component C needs version 2 of Component D to function. This problem is called the diamond dependency problem. Other examples of similar but resolvable issues are more common. For example, if Component C in Figure 2.4 relies on any version of D in the range 1.1 to 2.0 and Component B depends on any version between 1.6 to 3.2. then, a package manager can decide to use version 1.1 of Component D after examining the dependencies of C. However, as soon as it reaches Component B, the assumption breaks and it has to reconsider its decision by evaluating both Component dependencies. Moreover, only version 1.0 of C actually supports this range, another version might change the range, subsequently, dependencies of C and B can not be resolved. Then the version of C has to be changed. As the number of Components grow, the possibilities increase exponentially.

Resolving such dependencies is NP-complete [ACTZ12]. However, some systems allow the concurrent installation of the same package with different versions. This means solutions can be found more quickly, but unfortunately the solution is not always minimal [Cox]. Moreover, it is not feasible to install many versions of the same Component in most systems.

The Eclipse plugin framework uses a satisfiability (SAT) solver to resolve its dependencies. As Eclipse's plugin infrastructure faces the same issues as conventional Component based systems regarding dependency resolution [LR09], Components and its dependencies have to be modeled as a Boolean satisfiability problem. Such a problem consists of a Boolean function. The corresponding solution are the values of each variable inside the function. SAT solvers are algorithms which are built for the single purpose of solving these problems. Most modern dependency resolutions use a SAT solver as stated in [Cox]. Constraint Satisfaction Problems can be solved by SAT solvers as well, as described in [TTB10]. Other solutions include tree search combined with heuristics, as used in Aptitude. Bundler uses a backtracking algorithm from the Molinillo project[11], similar approaches are explained in Chapter 5.

Updating cloud deployment models closely relates to classical dependency resolution in package managers, as a topology acts similar to a dependency graph. This can be seen in the Aeolus deployment model, which allows to model more fine grained Components, such as, packages in the Debian package management system [CDE+13]. The Aeolus deployment model will be explained further in Chapter 3. This thesis uses a planning approach to solve dependencies of deployment topologies. Planning problems can be solved by classical SAT solvers [KS92], subsequently similar results can be achieved as described in Chapter 4.

[11] https://github.com/CocoaPods/Molinillo

3 Related Work

The following sections showcase different topics which have already been covered by previous works and how they relate to the approach presented in this thesis. At first, Section 3.1 explains techniques for validation. Afterwards, Section 3.2 analyzes how previous works adjust deployment models. Lastly, several works are presented in Section 3.3 which already utilize a planning approach in the domain of cloud deployment models.

3.1 Validation of Application Topologies

Validation of topologies inside deployment models can be done in various ways. The tool Sommelier [BTS18], first introduced in the year 2017 aims to validate cloud topology connections between Components. Its first prototype was explained in the article “Validating TOSCA Application Topologies” by Brogi et al. [BTS18]. Sommelier builds on top of the TOSCA OASIS standard [TOSCA-v1.0] and uses the tools which were already developed at the time to use TOSCA based deployment models. Namely TOSCA engines which enable users to deploy TOSCA deployment models on different infrastructures. As TOSCA engines do not include validation [BTS18], only the actual deployment itself, Sommelier is used in a previous step to verify correctly connected Node Types. To verify Capabilities and Requirements of all Node Types inside the topology, the authors extracted all necessary conditions from the standard. After formalizing them an algorithm was built which can tell cloud orchestrators if the specified topology is correctly declared. If that is the case, any TOSCA engine is able to deploy the TOSCA model, as long as all Node Types used in the deployment model are supported by the engine.

Further efforts, including detecting problems by using architecture and design patterns, have been made by Saatkamp et al. [SBKL19]. Formalizing these patterns has been done using First Order Logic. By evaluating the topology with the corresponding predicates, problems can be highlighted. Then a pattern is proposed to the cloud application developer, which is able to resolve the presented problems. This enables cloud application developers to quickly find solutions to problems which need specialized knowledge to solve them. Further works by Saatkamp et al. [SBF+19] show how such problems can be solved in an automated manner, as will be explained in the next section.

These methods of validating applications topologies focus on the TOSCA OASIS standard [TOSCA-v1.0], but do not work for a variety of deployment technologies. The approach presented in this thesis uses the Essential Deployment Metal Model which enables the use of multiple deployment models by mapping them to the EDMM model [WBF+19]. Furthermore, the complexity of correcting topologies increases, as the size of the model grows. Cloud application developers need to have specific knowledge about many Components to solve the problems described by the approaches presented in this section. Therefore automatic updating of deployment models is needed to prevent the necessity of domain specific knowledge.

3.2 Automatic Adjustment of Application Topologies

Changing cloud deployment models in an automated manner has been proposed several times with various approaches and different results [HBBL14][SBK+18][SBLW17][SBKL19]. An example of this is the automatic completion of partial topologies in TOSCA [TOSCA-v1.0] cloud applications by Hirmer et al. [HBBL14]. The approach covers automatic resolution of Requirements by adding all necessary Components to the topology until all Requirements are fulfilled. Therefore topologies which contain nodes whose dependencies are not met can be adjusted until all dependencies are resolved.

Another concept explained using TOSCA is injection of Node Templates or partial topologies. The main concept is explained by Saatkamp et al. [SBK+18] [SBLW17]. By injecting Node Templates or even other partial topologies into any incomplete topology a deployable topology can be achieved. A provider repository which contains Node Types and partial topologies is being used. By comparing available Capabilities from the repository with the necessary Requirements of the initial topology a deployable result can be achieved.

Further changes can be made by splitting topologies into different sub graphs as described by Saatkamp et al. [SBKL17]. Components inside the initial topology will be labeled with attributes, which describe how a topology should be split. The target labels correspond to a target cloud provider or any virtual environment, e.g, an Ubuntu virtual machine. However, target labels are not limited to a single Component, different levels of granularity can be applied. As the splitting occurs, labels on Components are checked and their respective Requirements will be fulfilled by other Components corresponding to the attached target labels. Another approach by Saatkamp et al. is based on validating applications with First Order Logic [SBKL19] as mentioned in the previous section. After detecting problems and suggesting a pattern to solve it, a fitting Topology Adaptation Algorithm is found which transforms the topology as desired. In this work validation has been taken even further. Context aware validation is presented, which detects problems regarding the context in which Components operate, e.g., an insecure connection over a public network. Those problems can occur after splitting a topology. In order to solve them, a fitting algorithm is found which changes the topology accordingly.

However, none of these works are able to update deployment models which have not been used for long periods of time. As Component versions age, security Requirements and compliance rules can not be satisfied and the topology has to be adjusted. By using First Order Logic and planning together with meaningful actions this can be achieved as explained in this thesis. Furthermore, problems, such as the diamond dependency problem are not addressed by the above mentioned approaches. Which can lead to unnecessary Components being used. The planner used in the method presented in this thesis solves this problem, as described in Chapter 4.

3.3 Planning in Cloud Deployment Models

Planning is used to generate plans on how to successfully deploy TOSCA [TOSCA-v1.0] models, as depicted by Breitenbücher [Bre16]. In the work, *planlets* are introduced. They perform one or several management operations, such as installing a Component on the corresponding operating system. Planning is used to find a valid sequence of planlets to successfully deploy applications. They behave as actions, whereas the initial state is modelled from provided deployment models.

Further deployment techniques involving planning are demonstrated by Di Cosmo et al. [DLT+14]. This approach involves the *Aeolus* Component model [CDE+13] in which Components contain state machines which indicate the state a Component can reside in, e.g., *installed* or *running*. Aeolus does not only model coarse grained Components, such as the ones shown in the running example but also more fine grained Components, such as packages. This generalizes the overall problem of dependency management. During deployment internal states of the Components are being changed. Plans to deploy Components are given by actions which involve the Component Type and the Component state in which it should operate. States change during the deployment phase, e.g., the initial state of a Java application is *uninstalled*, after a runtime Component reaches the state *installed* the Java application can transition to the same state as its Requirements are fulfilled. The goal state is a complete deployment of all requested Components in the desired Component state. Actions include creating, removing, binding, unbinding, and changing the state of the Component. As the deployment takes place, actions which can be performed are limited, as the states of Components constrain the planner in taking actions, such as *bind*. E.g the Component MySQL can only be connected to an application if it already reached the state *installed*. Plans are generated to ensure a successful deployment by following all actions in the sequence provided by the planner.

A different idea is presented by Georgievski et al. [GNLA17] which uses Hierarchical Task Network planning to solve the deployment of Aeolus Component models. In contrast to classical planning which was described in Section 2.3, HTN planning receives an initial state and a task network as input. Tasks can be executed by operators. But a single task may include many sub tasks which can not be executed in a sequence. Subsequently, methods are being used to declare how tasks can be decomposed. Both tasks and methods will be provided initially. The problem is solved as soon as all tasks are decomposed and can be run by operators [GNLA17]. The article proves that HTN planning can be used to successfully deploy Aeolus models.

However, all of the approaches described only encompass the deployment phase, not the modeling itself. In order to update deployment topologies changes have to take place during the modeling phase. Breitenbücher [Bre16] describes automatically correcting problems which occur during the modeling phase by using patterns, but planning is only used to deploy the model. By extending the usage of planning to the modelling phase, automatic updates can be accomplished and necessary dependencies will be resolved, which is depicted in the following approach.

4 An Approach to Update Deployment Models

The motivating scenario in Section 2.1.1 describes a severely outdated application deployment model. By using the approach presented in the following, cloud applications developers are able to update such models. The approach is based on planning. A planner takes over the entire update process. However, pre- and post processing steps have to be made in order to use such a planner. The initial topology has to be represented in PDDL, which was explained in Section 2.3.1. Furthermore, rules have to be set which tell the planner how valid topologies are defined. Afterwards, the generated plan is applied to the initial topology and the updated version can be extracted.

4.1 Overview

To update existing deployment models by considering constraints between different versions used in the deployment model, several steps have to be taken, they are shown in figure Figure 4.1. An existing deployment model is analyzed based on compliance rules by a cloud application developer. The deployment model itself was created several years ago. Subsequently, Ubuntu version 12.04 does not satisfy the compliance rules. This Component Type needs to be interchanged with another type using a newer version. The cloud application developer detects that this Component has to be changed. The figure shows a topology which uses this outdated Component *Ubuntu 12.04*. Compliance rules only allow version *Ubuntu 18.04*.

The topology itself is provided in form of the EDMM YAML specification[1]. The specification is used in the approach as many different automatic deployment technologies can be mapped to EDMM. Therefore technologies, such as Chef or TOSCA can be used with the approach presented.

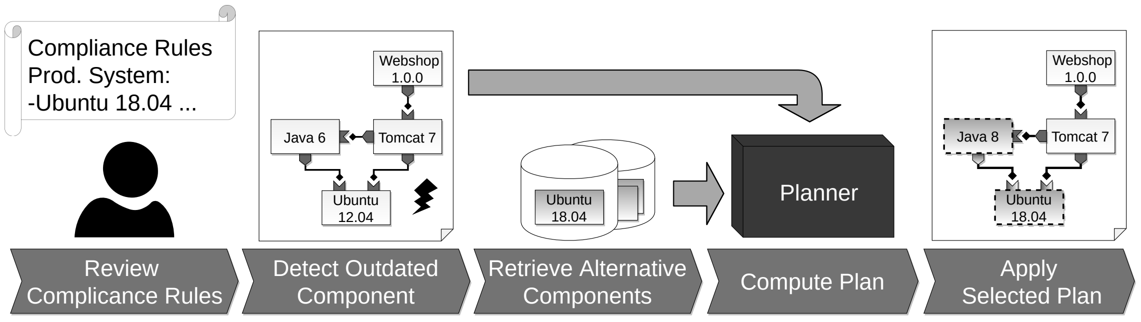


Figure 4.1: Overview of the approach

[1]https://github.com/UST-EDMM/spec-yaml

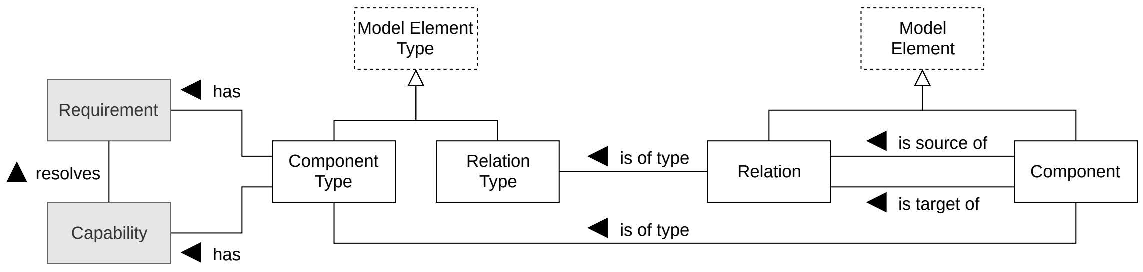


Figure 4.2: Extension of the EDMM model

In order to update the topology, information on which versions can be used, has to be gathered. A database contains all Component Types and their dependencies. Among them is the requested *Ubuntu 18.04* Component Type. It contains a list of all matching Requirement and Capability pairs and a list of possible Component Types. With this information the planner is able to decide which Components resolve a Requirement of another one by looking at its Capabilities and checking if one of them matches with the Requirement to resolve. As soon as the deployment model, the additional types, and their respective Requirements and Capabilities are gathered the planner will be used.

The planner receives the data from the deployment model and the database. By solving the provided problem several plans will be generated which update the topology according to the needs of the cloud application developer. Figure 4.1 shows that the Ubuntu Component has received a new Component Type. As requested by the cloud application developer, the Component Type *Ubuntu 18.04*, is now used inside the topology. However, as the Component Type changed, the Capabilities of the Component itself changed as well. Components which depend on the initial *Ubuntu 12.04* have to be analyzed if their Requirements are still met. The planner notices that the Requirement of *Java 6* is not fulfilled anymore, but that the Component is still needed by *Tomcat 7*. Subsequently, the planner updates the Java Component as well, because the Requirements of *Java 8* match the Capabilities of the new *Ubuntu 18.04* Component. The previous Java version is not compatible with the newly requested Ubuntu version.

Each action used by the planner to achieve an updated deployment model with resolved dependencies is contained inside the plan. Many plans can be generated by the planner, but all of them result in a valid deployment model which follows the Requirements given by the database. One of the plans is chosen, as explained in Section 4.4. In this case the action change version was used twice to update the topology inside the deployment model. This plan is then applied to the topology itself. The result is a new deployment model which satisfies the needs specified earlier. Then the cloud application developer receives the updated model in the EDMM format. If another technology was used before and then mapped to EDMM, the developer is able to update their deployment model according to the resulting EDMM model. By comparing the initial EDMM model and the resulting one, necessary changes can be seen and applied by hand to any technology which can be mapped to EDMM. The following sections will explain all steps described in further detail.

4.1.1 Information encoded in the initial state

The Essential Deployment Meta Model specifies how information about the initial deployment model and its internal topology is encoded. All Components and their respective Relations are contained in a EDMM compliant YAML file. They are extracted, together with all Component

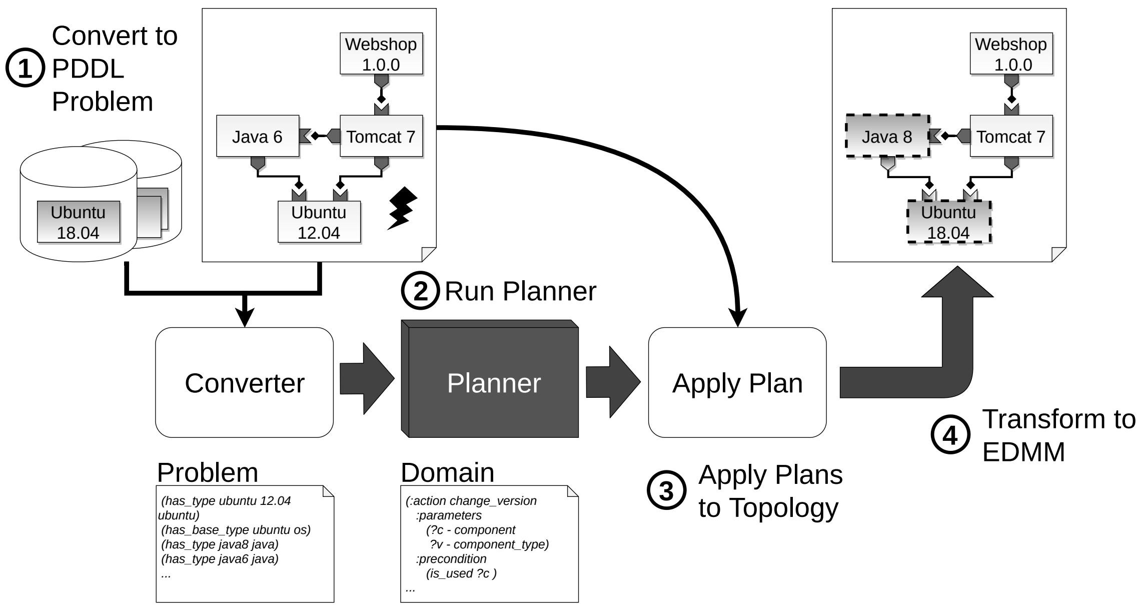
Types. However, this approach uses only some parts of the EDMM model which relate to the topology, shown in Figure 4.2. Components, and Relations with their respective types are used, all other parts of the model will be ignored by the approach as they are not modeling the topology itself. However, the model does not provide information on which Component Types can be connected to each other. As a result, dependencies can not be analyzed with the EDMM alone. A dependency will be expressed as a tuple of a Requirement and a Capability where the Capability resolves the Requirement. A Requirement means a Component Type needs another Component in order to function, whereas the Capability of another Component can fulfill it. The terms Requirement and Capability are inspired by the TOSCA standard by OASIS. This thesis uses the same concept to resolve dependencies. Component Types of the model defined by EDMM are extended to contain both Requirements and Capabilities as shown in Figure 4.2.

Additional types might be needed, they are contained in the Component Type database. For example, the topology uses Components of type Ubuntu 12.04, such as the running example in Figure 2.1 but the type Ubuntu 18.04 should also be considered by the planner. Then the Component Type database will additionally contain the Component Type Ubuntu 18.04. Even more data is needed to determine which types can be used together. Java version 6 can not be used together with Ubuntu 18.04, as explained in the documentation[2]. Which means that both Components are not compatible. The database also has to contain all Requirements and Capabilities of Component Types. A translator will transform the data from the database and EDMM file into an internal representation, which follows the model in Figure 4.2. The internal model ensures the possibility of interchanging translators, so that translators for different deployment models can be developed. This model can then be exported to the PDDL problem file. The second task a transformer has, is to manage converting the internal representation back into EDMM. As EDMM supports more information than needed by the planner, only a subset will actually be translated into the internal model. In order to preserve the whole data pool of EDMM the initial deployment model has to be kept. From this data additional data which is not represented in the internal model can be reconstructed. Newly included Components, or even ones that were fundamentally changed will only contain information which is stored inside the model. Additional data, such as operations have to be added by the maintainer or another algorithm, as this information is not available to the prototype, explained in Chapter 5.

4.1.2 Updating Deployment Models as a Planning Problem

Finding matching Component Types by resolving all necessary dependencies of them can be thought of as a Constraint Satisfaction Problem. For instance, if a particular Component of type Java 6 is used, all its dependencies have to be resolved, as can be seen in Figure 2.1. These dependencies can be thought of as a constraint, e.g., a Component which satisfies the dependency of Java 6, such as Ubuntu 12.04, as shown in the example, has to be contained in the deployment model. If other Components also need an operating system to run on, they have to support Ubuntu 12.04 as well. If that is not the case, a new Component which satisfies the Requirement has to be added to the topology of the deployment model or the current type of the Component has to be changed. A decision has to be made. Planners take decisions by using actions, which alter the initial state.

[2]https://www.oracle.com/technetwork/java/javase/system-configurations-135212.html



```
 (has_type ubuntu 12.04
ubuntu)
 (has_base_type ubuntu os)
 (has_type java8 java)
 (has_type java6 java)
 ...
```

```
(:action change_version
   :parameters
      (?c - component
       ?v - component_type)
   :precondition
      (is_used ?c )
...
```

Figure 4.3: Detailed overview of the approach

Furthermore, planning problems can be transformed to a Constraint Satisfaction Problem, as shown in Section 2.2. As planning already has a variety of applications in deployment models this thesis extends its usage to the modelling phase to update them.

As information about the initial topology is provided by an EDMM compliant format and further types are being stored inside the database, all necessary information can be extracted and converted into an initial state in a PDDL problem, as depicted in step (1) in Figure 4.3. Goals specify that all dependencies of every Component have to be resolved, to solve the problem. Even further goals can be formulated to change the initial topology as desired, e.g., all Ubuntu Components should be of type Ubuntu 18.04, such as in the example shown in Figure 4.1. The initial state of the topology, corresponding objects, dependencies of them, and the goal belong to the PDDL problem.

The domain contains necessary constraints to build a working deployment model. They are expressed as derived predicates and ensure valid dependencies between Components. In order to keep the problem file short and readable, all constraints are collected inside a single derived predicate, which has to be added to the goal. As a result, the planner always has to consider them. The second part of the domain file are actions to change the state. The most important action is changing the type of a Component so that in the running example Figure 2.1 Ubuntu 12.04 can be changed to a newer version, such as Ubuntu 18.04. Further actions which are needed to update the deployment model are explained in Section 4.3.2.

A planner then generates a plan based on the problem and domain file in step (2) in Figure 4.3. In step (3) the plan is applied by an external program to the model discussed in the previous Section 4.1.1. After the model has been updated step (4) begins. The model is converted to an EDMM compliant model and can be used by the cloud application developer.

4.2 Representing Deployment Models in PDDL

This section explains the PDDL problem file and all of its contents. The file itself contains all necessary information for the planner to generate a plan which successfully updates a topology, if a solution is possible, as shown in Figure 4.3. The problem has to contain the initial topology graph as well as all possible dependencies which are expressed in Requirements and Capabilities depending on the Component Type used. Lastly, a PDDL problem, based on the running example is presented.

4.2.1 Establishing the Topology Graph

The deployment model has to be expressed in PDDL to update the given topology. Components are expressed as PDDL objects, of type `component`. They can be connected with each other, which is achieved by using the `(connected_with ?n0 - component ?n1 - component)` predicate, which takes two Components as input. The first Component has a Requirement which is fulfilled by the second Component. Components in the PDDL model have Component Types as well, e.g., *Java 6*. Initially the number of Components is the same as the number of Components used in the provided topology. However, as further Components might be needed in the updating process more of them have to be added. In most deployment models, the updating process does not require more than two or three extra Components than initially needed. However, larger topologies inside deployment models might require more extra Components.

Components can also be removed, which is why the planner has to have the ability to remove Components from the topology entirely. For example, if a Component has been updated and one or several Requirements are not needed anymore and the Components which fulfill those Requirements are not needed elsewhere, they should be deleted, as they serve no purpose. To express if Components are needed, the predicate, `(is_used ?n - component)` indicates that this is the case. In the initial state all Components inside the topology are declared as being used. Furthermore, the connections between them are added to the PDDL problem file as well by using the `(connected_with ?n0 - component ?n1 - component)` predicate. The predicates enable cloud application developers to specify which Components should be kept and which connections should be upheld in any case.

4.2.2 Declaring Component Types

As the topology contained inside a deployment model is a typed, directed graph a type hierarchy has to be established in PDDL. Each Component has to be of a Component Type. However, not every type is actually deployable. Some of them serve abstract purposes to group similar Component Types, e.g., the types *Ubuntu 12.04* and *Ubuntu 18.04* are of type *Ubuntu*, but the Ubuntu type itself can not be deployed as no version is specified. Subsequently, deployable and non deployable Component Types have to be distinguished, see Figure 4.4. PDDL objects of different types are used to model the deployable trait. Component Types which can be deployed are objects of type `component_type`, whereas abstract Component Types which can not be deployed are referred to as `abstract_component_type`.

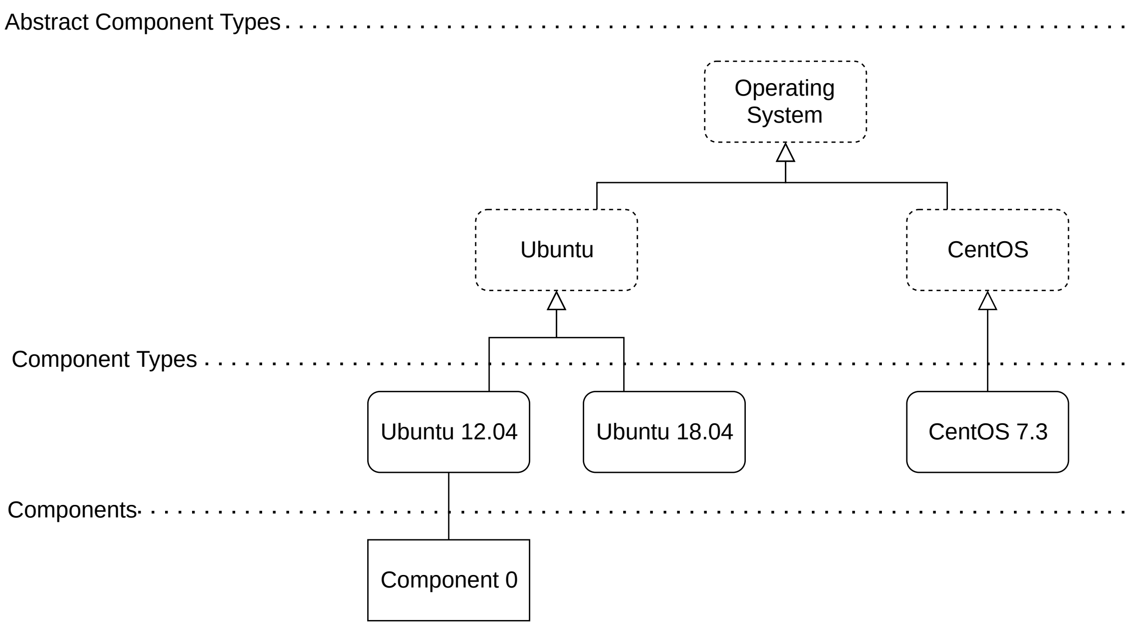


Figure 4.4: Inheritance and type overview

The inheritance between a `component` and its `component_type` is modelled by the `(has_type ?n - component ?v - component_type)` predicate. If a Component version is changed, this predicate has to be changed to the new Component Type which is now being used. For example, `(has_type component0 ubuntu-1204)` can not be true after an action has been chosen to change the type to Ubuntu 18.04. Instead the predicate `(has_type component0 Ubuntu-1804)` is now true in the current state.

Distinguishing between abstract and deployable Component Types enables creating two different actions. Changing the version of a Component only allows to change to a Component Type which is a direct sibling of the current one, see Figure 4.4. And changing to a Component Type which is only connected to the current one through a common ancestor, for example, changing Ubuntu 12.04 in the running example to Ubuntu 18.04 only requires the first action as they share the parent Ubuntu. However, to change to another operating system, e.g., CentOS the second action has to be used. As CentOS does not share an immediate ancestor with Ubuntu 12.04, as shown in Figure 4.4. Both share the ancestor Operating System, from which the parent of Ubuntu 12.04, Ubuntu inherits. By distinguishing between changing to a type with the same parent and changing to another type which only shares some ancestor with the current one, the planner can be controlled more precisely, as will be explained in Section 4.3.2.

Corresponding abstract Component Types, from which Components Types inherit, are modelled with the predicate `(has_abstract_type ?v - component_type ?t - abstract_component_type)`. However, as abstract Component Types can also have parents another predicate is needed `(has_parent_type ?t0 - abstract_component_type ?t1 - abstract_component_type)`. As a result, complete multi-level inheritance between abstract Component Types is achieved while still distinguishing between deployable Components and their abstract Components.

The predicates also help application developers to model the goal state. They can be used to force a planner to use a specified Component Type. E.g Ubuntu version 18.04 should be used for a fixed Component. Furthermore, individual abstract Component Types can be set, whose children should be used, for example, only using operating systems of type Ubuntu.

4.2.3 Expressing Dependencies

In order to validate dependencies, they have to be modeled in PDDL. The approach presented in this thesis uses terminology from the TOSCA OASIS standard [TOSCA-v1.0] to model them. A dependency consists of a single Requirement and Capability. They can be seen in the running example Figure 2.1. Requirements and Capabilities of Components are inherited from their corresponding Component Type. If a Capability matches a Requirement, the connection in the topology can be established and the dependency is resolved. If a Requirement fits to a particular Capability, as defined in the TOSCA standard, can be calculated as depicted by Brogi et al. [BTS18]. This approach requires a list of Requirements and Capabilities which do match together. Such a list is provided by the database, shown in Figure 4.3.

A PDDL object of type `reqcap` is used to decide whether Requirements and Capabilities match. If they share the same object, the dependency is valid. To decide which reqcap object belongs to which Component Type, two predicates are being used. The `(has_requirement ?v - component_type ?c - reqcap)` predicate for Requirements, and `(has_capability ?v - component_type ?c - reqcap)` for Capabilities. Component Types can have as many of them as needed to express the whole topology and its dependencies.

4.2.4 Defining a Goal State

The goal state specifies which Components should be changed to achieve the desired result formulated by the cloud application developer. Predicates established earlier enable the developer to design a variety of goal states. They can even be generated by an algorithm as their notation is the same as the one used in other parts of the problem or domain file. In the running example, the Ubuntu operating system should be updated to the newer version 18.04, as the current version is severely outdated. To achieve this three predicates connected by an `and` statements are required. The first one ensures that a deployable topology is produced. The other two tell the planner to use the type Ubuntu 18.04 for the existing Components, currently of type Ubuntu 12.04.

However, the Capabilities of First Order Logic allow to specify more complicated goals, such as using at least version 16.04. By using `or` statements which contain all valid versions to use.

4.2.5 PDDL Problem for the Running Example

Listing 4.1 shows an excerpt of the problem file which is used for the running example. All concepts described in previous sections are applied. As mentioned before the problem file consists of objects, the initial state, and the goal. Line 3 to Line 11 contain the necessary Component Types, abstract types, the Components themselves, and `reqcap` objects to identify valid connections between

Listing 4.1 Extract of the problem file for the running example

```
1  (define (problem running_example)
2  (:domain deployment_domain)
3  (:objects
4   ubuntu - abstract_component_type
5   ubuntu-1204 - component_type
6   ubuntu-1804 - component_type
7   ...
8   comp0 - component
9   ...
10  hosted-on-ubuntu-18 - reqcap
11  ...)
12 (:init
13  (has_abstract_type ubuntu-1204 ubuntu)
14  (has_abstract_type ubuntu-1804 ubuntu)
15  (has_abstract_type java-6 java)
16  (has_abstract_type java-8 java)
17
18  (has_requirement java-6 hosted-on-ubuntu-12)
19  (has_requirement java-8 hosted-on-ubuntu-18)
20  ...
21
22  (has_capability ubuntu-1204 hosted-on-ubuntu-12)
23  (has_capability ubuntu-1804 hosted-on-ubuntu-18)
24  ...
25
26  (is_used comp0)
27  (has_type comp0 ubuntu-1204)
28  ...
29 )
30 (:goal
31  (and (check_all_nodes)
32    (has_type comp0 ubuntu-1804)
33    (has_type comp6 ubuntu-1804))))
34
```

Components. In the initial state (Line 12 to Line 29) defines the topology as pictured in the running example Figure 2.1. The Requirement of Java 6 can be seen in Line 18, and the corresponding Capability of Ubuntu 12.04 in Line 22.

The goal state begins with an `and` predicate which contains the `check_all_nodes` derived predicate from the domain file. It always has to be present for the planner to follow instructions to ensure a valid topology in the goal state. By declaring that both Components have to be of type Ubuntu 18.04 the planner will update the topology accordingly. A plan which achieves all goals specified can have multiple forms. The planner chooses to change the type of Java 6 to Java 8 and of both Ubuntu Components to type Ubuntu 18.04. All plans produce a deployable topology according to the needs of the cloud application developer.

Listing 4.2 Resolve Requirements predicate

```
1 (:derived (resolve_requirements ?n - component ?v - component_type)
2   (forall (?c - reqcap)
3     (or (not(has_requirement ?v ?c))
4         (exists (?n1 - component)
5           (and (resolve_specific_relation ?n ?n1 ?c)
6                (connected_with ?n ?n1))))))
```

4.3 Updating Deployment Models with Planning

This section states how the updating process is realized by explaining the domain file, which is used to tell the planner under which conditions a topology is valid. In addition to validating the topology graph the planner also needs to have a set of actions which enables the algorithm to change an initial state until the desired goal state is reached. The domain file as a whole can be seen in Appendix A.1.

4.3.1 Modelling Dependencies as Constraints

To check if the topology is valid, a predicate must be declared in the goal state. A single derived predicate is used to bundle all properties which ensure a deployable topology. The predicate is called `check_all_nodes`. It contains a universal quantifier over all Components. It declares all properties inside the quantifier to be true for each Component which is used in the topology. Two properties are checked, (1) all Components which are used need to have a type and (2) they need to have at least one connection.

The first step does not only validate if a type is set for each Component but also tells the planner that the Requirements of the corresponding type need to be fulfilled by calling the `resolve_requirements` predicate, shown in Listing 4.2. A universal quantifier checks all objects of type `reqcap` whether they are used as a Requirement for the Component Type. If a Requirement has been found, Line 4 to Line 6 confirm that another Component exists which is connected to the initial Component and that the Relation has to be valid (Line 5). The predicate `resolve_specific_connection` validates that Capability and Requirements of the connected Components share the same `reqcap` object.

The second step (2) has the purpose of validating already established connections, it tells the planner that if a connection does exist, the Requirements and Capabilities have to match in the same manner as described in step (1). Furthermore, it has the effect of telling the planner that every Component needs to have at least one connection. Subsequently, unnecessary Components will be deleted, as they are not used by any others. This is based on the assumption that cloud application developers will not try to update a topology which only contains a single Component, as the updating process is fairly simple in that case.

Listing 4.3 Change type action from the domain

```
(:action change_type
:parameters (?i - component ?nV - component_type ?oV - component_type)

:precondition (and  (has_type ?i ?oV)
                    (exists (?t - abstract_component_type)
                            (and (has_abstract_type ?oV ?t)
                                 (has_abstract_type ?nV ?t))))

:effect (and (has_type ?i ?nV)
             (not(has_type ?i ?oV))))

```

Listing 4.4 Action to add a new Component to the deployment model

```
(:action add_component
:parameters (?i - component ?v - component_type)

:precondition (not (is_used ?i))

:effect (and (is_used ?i)
             (has_type ?i ?v)))

```

4.3.2 Defining Actions to Change Deployment Models

In order to generate a plan which changes a topology until the goal state is reached, actions have to be defined. They change the initial state according to their effects, giving a planner the ability to build a valid topology demands several actions.

The most obvious one is depicted in Listing 4.3. Three parameters are necessary to change the type of a Component. The Component object to change, its previous type, and the new desired type. In order to use this action preconditions have to be satisfied. In this case the current type has to be the type of the Component to change. And both Components types have to share a mutual parent, e.g., Java 6 in the running example can only be changed to any Component Type which has an abstract type of Java, such as Java 8.

There are cases in which it is desirable to change to a Component Type which has a different abstract type. Such as changing Ubuntu 12.04 to another type of operating system, such as Cent OS version 7.2. For this case another action is used, called `change_type_by_ancestor` which essentially does the same as `change_type` but does allow changing to another abstract Component Type as long as some shared abstract parent between the Component Types exists. By using two different actions which change Component Types the planner is able to be controlled more precisely. In many cases it is not desired to drastically change Component Types, such as changing Ubuntu 12.04 to CentOS 7.2. By distinguishing between sibling Components and those that only share some ancestor, weights can be introduced in the future. A bigger weight is chosen for the action `change_type_by_ancestor`. If the planner is then told to minimize action costs, it will try to prevent using the `change_type_by_ancestor` action while preferring smaller changes to the topology.

If the type of a Component has been changed, new Requirements can occur. In case they can not be satisfied a new Component has to be included. Its type has to satisfy the Requirements, by having the same `reqcap` object as Capability. Therefore an action which adds Components to the topology has to be included, as shown in Listing 4.4. And vice versa Requirements can be removed after a type has been changed. An action to remove Components is used for this case. Both actions function in the same manner. When adding a new Component to the topology inside the deployment model the Component can not exist in the current state. Subsequently a precondition which states that a Component is currently not used is implemented. When removing a Component a precondition has to be defined as well, it states that the Component is used in the current state. The effects of both actions either include the new component with the specified type or remove it. In case of including a new Component the requested Component Type is contained as a parameter.

Lastly, connections between nodes have to be established or removed. In order to check if Requirements of Components are fulfilled, their respective dependencies have to be found and a connection has to be established. Like the previous action, connections can be incorporated or removed. For connecting nodes the Requirements and Capabilities have to be of the same `reqcap` object.

4.4 Using the Plan

After running the planner one or several plans have been generated. Plans are not specified in the PDDL standard [GHK+98]. Some planners generate a dedicated file to every plan, such as the Fast Downward planner [Hel06], but others just print it on the standard output, such as Marvin [CS11]. This is why a wrapper class has to be written to correctly interpret plans from different planners. Afterwards, a general algorithm is used which applies all possible actions, defined in the PDDL domain to the initial topology. It mirrors the semantics provided by all actions from the domain file.

If multiple plans are generated one of them has to be chosen. However, as all of them provide a valid solution to the proposed problem in the initial state, an arbitrary choice can be made. However, heuristics could be implemented which reflect preferences of the developer, e.g., use the newest possible versions or change as little as possible. This will be discussed further in Chapter 5.

5 Prototype and Evaluation

This chapter explains how the prototype[1] can be used and some design decisions which had to be made. Section 5.1 describes a workflow using Eclipse Winery [KBBL13] which integrates the prototype. Afterwards, Section 5.2 compares two different planners used within the prototype. Lastly general concepts of the prototype are discussed and how future implementations can extend the current prototype.

5.1 Using the Prototype with Eclipse Winery

During the modelling phase of TOSCA [TOSCA-v1.0] deployment models Eclipse Winery [KBBL13] can be used. It enables developers to build TOSCA deployment models in a graphical environment. Repositories provide such deployment models in the TOSCA packaging format. They can be imported and exported as desired. Furthermore, EDMM exports are supported as well. Subsequently, TOSCA deployment models can be updated with the prototype by transforming them to EDMM compliant files inside Eclipse Winery. Other features of Eclipse Winery include a topology modeler and type management, which enables developers to change TOSCA topologies and their properties. This is an example of integrating the prototype in a workflow which deploys the updated deployment model in the end. However, the database containing a list of all types has to be provided, this can be achieved by using Sommelier [BTS18].

By using the prototype developed according to the approach presented in Chapter 4, together with Eclipse Winery, a complete workflow to update deployment models in the TOSCA format can be accomplished, as shown in Figure 5.1. A cloud application developer has an outdated deployment model packaged with the TOSCA packing format. Step(1) in Figure 5.1 shows that this model can be imported to Eclipse Winery and then be exported in the EDMM YAML specification. Afterwards

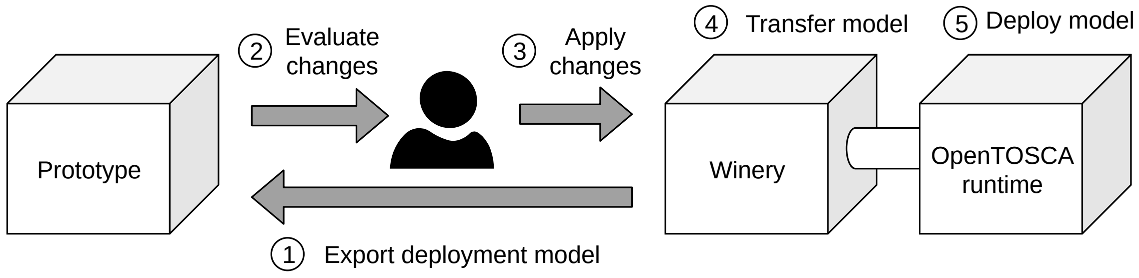


Figure 5.1: Overview of the workflow with Eclipse Winery

[1]https://github.com/FlxB2/Constraint-Based-Automated-Updating-of-Application-Deployment-Models

a developer specifies the goal state of the planner and provides the EDMM model, together with a database of all possible Component Types and their Requirements and Capabilities, to the planner. After starting the planner, updating of the deployment model takes place and the new model is received on the output which follows the EDMM YAML specification. In step (2) the developer evaluates these changes and is now able to apply the same changes to the TOSCAs deployment model in Eclipse Winery (step 3). Changes are visualized in a graphical user interface which shows the initial topology and the updated result. By evaluating both graphs the developer is able to apply the same changes inside Eclipse Winery. Then Eclipse Winery transfers the deployment model in TOSCA format to the OpenTOSCA runtime, which finally deploys the updated deployment model in step (5).

In future implementations, manual refinement of TOSCA deployment models is not necessary, which removes the developer from the workflow, if goal states are already defined. The prototype can be built as a service which receives inputs in the form of TOSCA deployment models, together with repositories from Eclipse Winery which provide all necessary Node Types. As TOSCA deployment models can be mapped to EDMM the planner is able to work with TOSCA deployment models as well, by changing the prototype to interpret the TOSCA deployment model format. A list of valid connections between Requirements and Capabilities can be achieved by using Sommelier [BTS18] on all Node Types provided from repositories.

5.2 Selection of a Planner

Since 1998 the International Planning Competition has taken place every two years. Researchers can register different planners which compete against others. For the implementation of the prototype, two planners can be used, as both support all necessary Requirements and both competed in the IPC, which means they can be compared to other planners by looking at the results of the competition.

Planning problems can be solved by searching a state space graph which describes the state after each action as a node in a graph [RN09]. Considering that the goal state is already defined when planning takes place two approaches of searching the state space can be considered, searching backwards from the goal state until the initial state is reached or searching forwards from the initial state [RN09]. Efficient searching in both directions requires heuristics. They estimate the distance between the current state and the goal state (or the initial state depending on the search direction). Heuristics can be domain independent and thus be automatically applied by a planner or they can be domain specific, which means they have to be derived by an analyst. Planners differ in their search strategies and in which heuristics are used. This leads to different execution times when using the domain and problem presented in Chapter 4.

As explained in Section 2.2, the number of actions a planner can perform increases significantly once the number of Components and the number of possible Component Types rises. This means more calculation has to occur inside the planner to consider the implications of each action. As a result, the planner will take a longer time for bigger deployment models and if more Component Types can be selected. Using the planner Fast Downward [Hel06] to solve the problem shown in figure Figure 2.1, requires $956.373s$ to run, while using A* search without domain specific knowledge[2]. A* search is a path search algorithm, which uses heuristics to optimize performance. However, by excluding the components MySQL DB 5.1, MySQL DBMS 5.1, and the Ubuntu Component on the right, the execution time decreases significantly. Using the same search algorithm and environment as before with the Fast Downward planner results in an execution time of $0.0163993s$. The performance varies with different planner implementations and with different search strategies, but all of them increase in execution time if the number of Components or Component Types increases. Chapter 6 discusses how this increase in execution time can be limited by running several planner instances simultaneously on subsets of the deployment model. Further improvements can be achieved by implementing domain specific heuristics and using other search algorithms.

The prototype was built in a modular fashion to enable the usage of various planners which implement different search strategies and heuristics. Two planners are already implemented, namely Marvin [CS11] and Fast Downward [Hel06]. The Fast Downward planner itself is built modular as well, which allows to use a variety of search algorithms and heuristics. The only prerequisite for a planner which uses the domain presented in section Chapter 4 is that all PDDL Requirements have to be supported, namely `:derived-predicates`, `:existential-preconditions`, and `:equality`. In future work, data can be collected which compares the execution times between different planners, search strategies and heuristics.

5.3 Discussion

This section explains properties of the prototype which have not been described in previous sections. Furthermore, future implementations and optimizations are discussed. Following the approach in Chapter 4, a prototype was built which is not only able to update deployment models but is also capable of automatically completing non deployable models which contain Requirements not satisfied. This is possible because the domain includes constraints which make the model deployable, as described in Section 4.3.1. If a cloud application developer provides any incomplete deployment model to the planner, it is able to complete it, as it has to fulfill all constraints. Furthermore, the usage of FOL in the goal state, enables developers to define many different rules for transforming the topology, which leads to more elaborate transformations of the deployment model, such as prohibiting the usage of a Component Type, never establishing a connection between certain Components etc.. Even specific forms of the topology can be achieved, like setting up the existing MySQL database on an external server. This means that the prototype is able to do a variety of transformations regarding deployment models by formulating or generating sophisticated goals which help cloud application developers in building deployable applications in many different forms.

Furthermore, the implementation is built modular, which makes it possible to change it without having to change the whole prototype. It enables developers to implement algorithms which map their specific deployment model to the model shown in Figure 4.2 without having to convert it to the Essential Deployment Meta Model. In addition, algorithms which apply the list of actions provided

[2]The prototype was run on an Intel(R) Core(TM) i5-5200U CPU @ 2.20GHz with 8 GB of RAM. The execution time was measured by the Fast Downward planner, the argument `--search 'astar(blind())'` was used to specify the search strategy.

by the planner have to be modified as well. Other possibilities include integrating other planners which use different algorithms or heuristics to increase the performance. Integrating the prototype directly into existing technologies, such as Eclipse Winery [KBBL13], which help cloud application developers in modelling deployment models, is a possibility as well.

6 Summary and Future Work

This thesis presented an approach to automatically update deployment models by using constraints, which define if a model can be deployed and how it should be updated. It describes how updating can be achieved by using an approach involving artificial intelligence planning. A prototype was built, it can be included in the modelling step of cloud application developers.

To ease the deployment process, maintainability and reproducibility of applications, cloud application developers use declarative deployment models. They ease maintainability and deployment efforts. Because of frequent updates, which relate to security and functionality, such deployment models need to be updated frequently to conform with existing compliance rules. The developer has to update the model manually. This process requires domain specific knowledge about all Components, such as an Ubuntu operating system version or a specific version of Java. A developer has to know which versions can work together successfully to provide the updated cloud application. To ease the updating process of cloud application developers, this thesis presented an automatic approach to update deployment models. The approach can be included in the modelling workflow of developers.

The problem of the updating process has been thoroughly analyzed and identified as a Constraint Satisfaction Problem problem, which can be solved with planning. Therefore the approach uses a planner to provide a sequence of actions on how to update deployment models. To support a variety of declarative deployment models, the Essential Deployment Meta Model is used. Therefore, commonly used deployment models, such as Puppet, Chef, or TOSCA, can be mapped to EDMM. Other related works have been described, including the validation of deployment models and their automatic adjustment. Afterwards works that use planning in the domain of deployment models were presented. It was found that none of these works are able to update deployment models automatically.

Subsequently, Chapter 4 contains the concept of the approach that enables automatic updating. It describes an overview of the process, starting with a cloud application developer who analyzes an outdated deployment model and identifies all Components which have to be updated. The developer encodes Components that need to change, to conform with all compliance rules. Afterwards the deployment model, further Component Types, and their compatibilities are gathered and encoded in a planner problem. This problem is then solved by a planner, that provides one or many plans to update the deployment model. One of the plans is interpreted and applied to the initial deployment model, the result is the updated model, which is exported and can be used by the cloud application developer.

Lastly a prototype is presented that can be included in the modelling process of TOSCA deployment models. A developer is able to export a model from Eclipse Winery into a EDMM compliant deployment model. This modell is then updated automatically by the prototype. Afterwards the cloud application developer is able to reproduce the update steps and can apply the same changes inside Eclipse Winery, which is connected to a deployment engine. The model can then be deployed by the deployment engine connected to Eclipse Winery.

Future Work

This thesis presented an approach to automatically update deployment models using constraints. A workflow was presented which includes the prototype shown and Eclipse Winery, a modelling tool for TOSCA deployment models. In the future the prototype could be implemented directly into such modelling programs, either directly or by building a service which modelling programs can query to receive updated cloud deployment models.

Furthermore, increased execution time of deployment models which include many Components, could be limited in the future, by splitting them into several partial models, multiple planner instances can be run in parallel, as described by Breitenbücher [Bre16]. Paralellizing the prototype can be done in four steps. Step (1) is splitting the provided deployment model into multiple submodels. Splitting of deployment models was already described by Saatkamp et al. [SBKL17]. In step (2) multiple planner instances are run in parallel which update each sub-deployment model. Step (3) merges the updated submodels back together. At last, step (4) runs a single planner instance on the whole deployment model to validate connections between sub-deployment models. The rejoined result can need less time to execute in step (4) as several steps were already taken by previous instances, which limits the overall number of steps which have to be taken and subsequently the number of calculations a planner has to execute. Parallelizing the prototype in this fashion can be done in future implementations. Further optimizations to increase the performance can be done in the domain, which is found in Appendix A.1.

Lastly the overall approach could be changed to directly solve the updating problem by encoding it as a Constraint Satisfaction Problem. Programs which are able to solve CSPs are then able to provide an updated deployment model by interpreting the results. The execution time of this approach can be compared to the execution time of the prototype presented in this thesis. The expressiveness of the goal state can still be maintained, by including goal states in the encoding of the CSP. Future works can also automatically generate such goal states to adjust deployment models in a variety of forms, such as migrating Components to different virtual machines or removing Components of a specific type, like Ubuntu, entirely.

Bibliography

[ACTZ12] P. Abate, R. D. Cosmo, R. Treinen, S. Zacchiroli. “Dependency solving: A separate concern in component evolution management”. In: *Journal of Systems and Software* 85.10 (2012). Automated Software Evolution, pp. 2228–2240. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2012.02.018. URL: http://www.sciencedirect.com/science/article/pii/S0164121212000477 (cit. on p. 23).

[Bre16] U. Breitenbücher. “Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements”. Dissertation. University Stuttgart, 2016. URL: http://dx.doi.org/10.18419/opus-8764 (cit. on pp. 14, 27, 46).

[BTS18] A. Brogi, A. Tommaso, J. Soldani. “Sommelier: A Tool for Validating TOSCA Application Topologies”. In: July 2018, pp. 1–22. ISBN: 978-3-319-94763-1. DOI: 10.1007/978-3-319-94764-8_1 (cit. on pp. 13, 25, 35, 41, 42).

[Cana] Canonical. *Ubuntu 12.04.5 LTS (Precise Pangolin)*. http://releases.ubuntu.com/12.04/ (cit. on p. 17).

[Canb] Canonical. *Ubuntu Bionic Packages mysql-server-code-5.7*. https://packages.ubuntu.com/bionic/mysql-server-core-5.7 (cit. on p. 17).

[Canc] Canonical. *Ubuntu Bionic Packages openjdk-8-dbg*. https://packages.ubuntu.com/bionic/openjdk-8-dbg (cit. on p. 17).

[CDE+13] M. Catan, R. Di Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, J. Zwolakowski. “Aeolus: Mastering the Complexity of Cloud Application Deployment”. In: *Service-Oriented and Cloud Computing*. Ed. by K.-K. Lau, W. Lamersdorf, E. Pimentel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–3. ISBN: 978-3-642-40651-5 (cit. on pp. 16, 23, 27).

[Cox] R. Cox. *Version SAT*. https://research.swtch.com/version-sat (cit. on p. 23).

[CS11] A. Coles, A. Smith. “Marvin: A Heuristic Search Planner with Online Macro-Action Learning”. In: *CoRR* abs/1110.2736 (2011). arXiv: 1110.2736. URL: http://arxiv.org/abs/1110.2736 (cit. on pp. 39, 43).

[DK00] M. Do, S. Kambhampati. “Solving Planning-Graph by Compiling It into CSP.” In: Jan. 2000, pp. 82–91 (cit. on p. 20).

[DLT+14] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, A. Agahi. “Automated Synthesis and Deployment of Cloud Applications”. In: *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ASE ’14. Vasteras, Sweden: ACM, 2014, pp. 211–222. ISBN: 978-1-4503-3013-8. DOI: 10.1145/2642937.2642980. URL: http://doi.acm.org/10.1145/2642937.2642980 (cit. on p. 27).

[EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. *Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications*. Feb. 2017 (cit. on p. 15).

[Ede04] S. Edelkamp. "PDDL2. 2: The language for the classical part of the 4th international planning competition". In: (Jan. 2004) (cit. on p. 20).

[FS92] B. Fromont, D. Sriram. "Constraint Satisfaction as a Planning Process". In: *Artificial Intelligence in Design '92*. Ed. by J. S. Gero, F. Sudweeks. Dordrecht: Springer Netherlands, 1992, pp. 97–117. ISBN: 978-94-011-2787-5. DOI: 10.1007/978-94-011-2787-5_6. URL: https://doi.org/10.1007/978-94-011-2787-5_6 (cit. on p. 19).

[GHK+98] M. Ghallab, A. Howe, C. Knoblock, D. Mcdermott, A. Ram, M. Veloso, D. Weld, D. Wilkins. *PDDL–The Planning Domain Definition Language*. 1998. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.212 (cit. on pp. 20, 39).

[GNLA17] I. Georgievski, F. Nizamic, A. Lazovik, M. Aiello. "Cloud Ready Applications Composed via HTN Planning." In: *SOCA*. IEEE Computer Society, 2017, pp. 81–89. ISBN: 978-1-5386-1326-9. URL: http://dblp.uni-trier.de/db/conf/soca/soca2017.html#GeorgievskiNL017 (cit. on p. 27).

[Har18] L. Harzenetter. "Versioning of Applications Modeled in TOSCA". In: 2018 (cit. on pp. 13, 16).

[HBBL14] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann. "Automatic topology completion of TOSCA-based cloud applications". In: *Informatik 2014*. Ed. by E. Plödereder, L. Grunske, E. Schneider, D. Ull. Bonn: Gesellschaft für Informatik e.V., 2014, pp. 247–258 (cit. on p. 26).

[Hel06] M. Helmert. "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research* 26 (2006), pp. 191–246 (cit. on pp. 39, 42, 43).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "Winery – A Modeling Tool for TOSCA-Based Cloud Applications". In: *Service-Oriented Computing*. Ed. by S. Basu, C. Pautasso, L. Zhang, X. Fu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 700–704. ISBN: 978-3-642-45005-1 (cit. on pp. 41, 44).

[KS92] H. Kautz, B. Selman. "Planning As Satisfiability". In: *Proceedings of the 10th European Conference on Artificial Intelligence*. ECAI '92. Vienna, Austria: John Wiley & Sons, Inc., 1992, pp. 359–363. ISBN: 0-471-93608-1. URL: http://dl.acm.org/citation.cfm?id=145448.146725 (cit. on p. 23).

[LR09] D. Le Berre, P. Rapicault. "Dependency management for the eclipse ecosystem: Eclipse p2, metadata and resolution". In: *IWOCE'09 - Proceedings of the 1st International Workshop on Open Component Ecosystems* (Jan. 2009). DOI: 10.1145/1595800.1595805 (cit. on p. 23).

[NHS+18] C. Neuefeind, L. Harzenetter, P. Schildkamp, U. Breitenbücher, B. Mathiak, J. Barzen, F. Leymann. "The SustainLife Project - Living Systems in Digital Humanities". Englisch. In: *Papers From the 12th Advanced Summer School of Service-Oriented Computing (SummerSOC 2018)*. IBM Research Division, Oct. 2018, pp. 101–112. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2018-35&engl=0 (cit. on p. 13).

[Ora] Oracle. *Oracle Java SE Support Roadmap*. https://www.oracle.com/technetwork/java/java-se-support-roadmap.html (cit. on p. 17).

[RN09] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 9780136042594 (cit. on pp. 19, 42).

[SBF+19] K. Saatkamp, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann. "An Approach to Determine & Apply Solutions to Solve Detected Problems in Restructured Deployment Models Using First-Order Logic". English. In: *Proceedings of the 9th International Conference on Cloud Computing and Services Science (CLOSER 2019)*. SciTePress, Apr. 2019, pp. 495–506. ISBN: 978-989-758-365-0. DOI: 10.5220/0007763204950506. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2019-11&engl=1 (cit. on p. 25).

[SBK+18] K. Saatkamp, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann. "OpenTOSCA Injector: Vertical and Horizontal Topology Model Injection". English. In: *Service-Oriented Computing - ICSOC 2017 Workshop*. Vol. 10797. LNCS. Springer International Publishing, Jan. 2018, pp. 379–383. ISBN: 978-3-319-91764-1. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2018-25&engl=1 (cit. on p. 26).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Topology Splitting and Matching for Multi-Cloud Deployments". English. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258. ISBN: 978-989-758-243-1. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2017-25&engl=1 (cit. on pp. 26, 46).

[SBKL19] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "An approach to automatically detect problems in restructured deployment models based on formalizing architecture and design patterns". English. In: *SICS Software-Intensive Cyber-Physical Systems* (Feb. 2019), pp. 1–13. DOI: 10.1007/s00450-019-00397-7. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2019-03&engl=1 (cit. on pp. 13, 25, 26).

[SBLW17] K. Saatkamp, U. Breitenbücher, F. Leymann, M. Wurster. "Generic Driver Injection for Automated IoT Application Deployments". English. In: *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services; Salzburg, Austria, December 4-6, 2017*. Ed. by M. Indrawan-Santiago, I. L. Salvadori, M. Steinbauer, I. Khalil, G. Anderst-Kotsis. ACM, Dec. 2017, pp. 320–329. ISBN: 10.1145/3151759.3151789. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2017-67&engl=1 (cit. on p. 26).

[SCEE17] I. O. for Standardization, I. E. Commission, I. of Electrical, E. Engineers. *ISO/IEC/IEEE 24765: 2017(E): ISO/IEC/IEEE International Standard - Systems and Software Engineering–Vocabulary*. IEEE Std. IEEE, 2017. ISBN: 9781504441186. URL: https://books.google.de/books?id=NS02tAEACAAJ (cit. on p. 16).

[SSS+12] R. Schwarzkopf, M. Schmidt, C. Strack, S. Martin, B. Freisleben. “Increasing virtual machine security in cloud environments”. In: *Journal of Cloud Computing* 1 (Jan. 2012). DOI: 10.1186/2192-113X-1-12 (cit. on p. 13).

[THN05] S. Thiébaux, J. Hoffmann, B. Nebel. “In defense of PDDL axioms”. In: *Artificial Intelligence* 168.1 (2005), pp. 38–69. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2005.05.004. URL: http://www.sciencedirect.com/science/article/pii/S0004370205000810 (cit. on p. 21).

[TOSCA-v1.0] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. OASIS Standard. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (visited on 07/01/2019) (cit. on pp. 13, 16, 25–27, 35, 41).

[TTB10] N. Tamura, T. Tanjo, M. Banbara. “Solving Constraint Satisfaction Problems with SAT Technology”. In: *Functional and Logic Programming*. Ed. by M. Blume, N. Kobayashi, G. Vidal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 19–23. ISBN: 978-3-642-12251-4 (cit. on p. 23).

[WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The essential deployment metamodel: a systematic review of deployment automation technologies”. In: *SICS Software-Intensive Cyber-Physical Systems* (2019), pp. 1–13 (cit. on pp. 15, 25).

All links were last followed on November 4, 2019.

A Appendix

A.1 PDDL Domain

```
(define (domain deployment_domain)
 (:requirements :typing :derived-predicates :strips :existential-preconditions :equality)
 (:types component_type abstract_component_type component reqcap)
 (:constants )
 (:predicates
  (is_used ?c - component)
  (has_parent_type ?t0 - abstract_component_type ?t1 - abstract_component_type)
  (has_abstract_type ?v - component_type ?t - abstract_component_type)
  (has_type ?c - component ?v - component_type)
  (has_requirement ?v - component_type ?r - reqcap)
  (has_capability ?v - component_type ?r - reqcap)
  (connected_with ?c0 - component ?c1 - component)
  (has_supertype ?t0 - abstract_component_type ?t1 - abstract_component_type)
  (share_supertype ?t0 - abstract_component_type ?t1 - abstract_component_type)
  (of_abstract_type ?n - component ?t - abstract_component_type)

  (check_all_nodes)
  (resolve_type ?n - component)
  (have_some_relation ?n - component)
  (resolve_requirements ?n - component ?v - component_type)
  (resolve_relation ?n0 - component ?n1 - component)
  (resolve_specific_relation ?n0 - component ?n1 - component ?r - reqcap))

 (:derived (check_all_nodes)
  (forall (?n - component)
    (or (not(is_used ?n))
     (and (resolve_type ?n)
       (have_some_relation ?n)))))

 (:derived (of_abstract_type ?n - component ?t - abstract_component_type)
  (exists (?v - component_type)
   (and (has_type ?n ?v)
     (has_abstract_type ?v ?t))))

 (:derived (resolve_type ?n - component)
  (exists (?v - component_type)
    (and (has_type ?n ?v)
      (resolve_requirements ?n ?v))))

 (:derived (have_some_relation ?n - component)
  (forall (?n1 - component)
    (or (not(connected_with ?n ?n1))
     (resolve_relation ?n ?n1))))

 (:derived (resolve_requirements ?n - component ?v - component_type)
  (forall (?r - reqcap)
    (or (not(has_requirement ?v ?r))
     (exists (?n1 - component)
      (and (resolve_specific_relation ?n ?n1 ?r)
        (connected_with ?n ?n1))))))

 (:derived (resolve_relation ?n0 - component ?n1 - component)
  (exists (?v0 - component_type ?v1 - component_type)
   (and (has_type ?n0 ?v0)
     (has_type ?n1 ?v1)
     (is_used ?n0)
     (is_used ?n1)
     (exists (?r - reqcap)
        (and (has_requirement ?v0 ?r)
           (has_capability ?v1 ?r))))))

 (:derived (resolve_specific_relation ?n0 - component ?n1 - component ?r - reqcap)
  (exists(?v0 - component_type ?v1 - component_type)
   (and (is_used ?n0)
     (is_used ?n1)
     (has_type ?n0 ?v0)
     (has_type ?n1 ?v1)
     (has_requirement ?v0 ?r)
     (has_capability ?v1 ?r))))

 (:action connect_components
  :parameters (?c0 - component ?v0 - component_type ?c1 - component ?v1 - component_type)

  :precondition (and (not(connected_with ?c0 ?c1))
         (has_type ?c0 ?v0)
         (has_type ?c1 ?v1)
         (exists (?r - reqcap)
           (and (has_requirement ?v0 ?r)
             (has_capability ?v1 ?r))))

  :effect (connected_with ?c0 ?c1))

 (:action disconnect_components
  :parameters (?c0 - component ?c1 - component)

  :precondition (connected_with ?c0 ?c1)

  :effect (not(connected_with ?c0 ?c1)))


 (:action change_type
  :parameters (?i - component ?cV - component_type ?oV - component_type)

  :precondition (and  (has_type ?i ?oV)
       (exists (?t - abstract_component_type)
        (and (has_abstract_type ?oV ?t)
          (has_abstract_type ?cV ?t))))

  :effect (and (has_type ?i ?cV)
      (not(has_type ?i ?oV))))

 (:action change_type_by_ancestor
  :parameters (?i - component ?cV - component_type ?cT - abstract_component_type ?oV -
component_type ?oT - abstract_component_type)

  :precondition (and  (has_type ?i ?oV)
       (has_abstract_type ?cV ?cT)
       (has_abstract_type ?oV ?oT)
       (not (= ?oT ?cT))
       (share_supertype ?cT ?oT))

  :effect (and (has_type ?i ?cV)
      (not(has_type ?i ?oV))))

 (:action add_component
  :parameters (?i - component ?v - component_type)

  :precondition (not (is_used ?i))

  :effect (and (is_used ?i)
      (has_type ?i ?v)))

 (:action remove_component
  :parameters (?i - component ?v - component_type)

  :precondition (and (is_used ?i)
          (has_type ?i ?v))

  :effect (and (not(is_used ?i))
      (not(has_type ?i ?v))))
)
```

Listing A.1: ”PDDL Domain defined in the approach and used in the prototype”

Declaration

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

place, date, signature