Hosting-Aware Pruning of Components in Deployment Models

Miles Stötzner[a], Sandro Speth[b] and Steffen Becker[c]
Institute of Software Engineering, University of Stuttgart, Stuttgart, Germany

Keywords: Pruning, Hosting, Constraints, Deployment Models, Variability Management, TOSCA, EDMM, VDMM.

Abstract: The deployment of modern composite applications, which are distributed across heterogeneous environments, typically requires a combination of different deployment technologies. Besides, applications must be deployed in different variants due to varying customer requirements. Variable Deployment Models manage such deployment variabilities based on conditional elements. To simplify modeling, elements, such as incomplete relations or hosting stacks without hosted components, are pruned, i.e., automatically removed from the model and, therefore, from the deployment. However, components whose hosting stack is absent are not automatically removed. Manually ensuring the absence of these components is repetitive, complex, and error-prone. In this work, we address this shortcoming and introduce the pruning of components without a hosting stack. This hosting-aware pruning must be correctly integrated into the already existing pruning concepts since, otherwise, a major part of the deployment is pruned unexpectedly. We evaluate our concepts by implementing a prototype and by conducting a case study using this prototype.

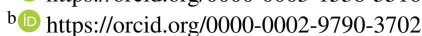
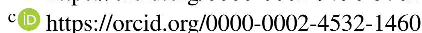
1 INTRODUCTION

Manually managing the deployment of applications is error-prone and complex (Oppenheimer et al., 2003; Oppenheimer, 2003). Therefore, deployment technologies, such as Terraform and Ansible, automate their deployment. Modern applications consist of multiple components that are distributed across heterogeneous environments (Brogi et al., 2018). Their deployment typically requires a combination of different deployment technologies (Guerriero et al., 2019; Wurster et al., 2021), which all have their area of application. Besides, applications must be deployed in different variants due to varying customer requirements, such as costs, elasticity, or required features. This further increases the complexity when deploying modern applications.

Variable Deployment Models (Stötzner et al., 2022, 2023a) manage such deployment variabilities. A Variable Deployment Model represents all possible deployment variants of an application based on conditional elements, i.e., application components, relations, configurations, and component implementations. Conditions assigned to elements specify when the elements are present in the model and, therefore, in the deployment. To reduce manual modeling efforts, elements are pruned, i.e., automatically removed from the deployment due to consistency issues and semantic aspects (Stötzner et al., 2023c). For example, a hosting component, such as Kubernetes, is removed once no hosted components are present, such as web applications, web servers, or databases.

However, hosted components themselves are not automatically removed once their hosting stack is absent. Kubernetes typically hosts not only application components but also operational components, such as monitoring agents, logging agents, or reverse proxies. Such operational components must be only present if Kubernetes is present. Also, if application components are absent but operational components are not, then Kubernetes is not pruned. Manually modeling conditions at all operational components to check for the presence of Kubernetes is repetitive and error-prone. Moreover, simply checking for the presence of Kubernetes leads to circular conditions, which introduces ambiguity and leads to the unexpected pruning of elements (Stötzner et al., 2023c).

To address this manual modeling effort and the ambiguity, we introduce the hosting-aware pruning of components, which prunes hosted components once their hosting components are absent while being interoperable with existing pruning concepts. Our contributions are as follows.

[a] https://orcid.org/0000-0003-1538-5516

[b] https://orcid.org/0000-0002-9790-3702

[c] https://orcid.org/0000-0002-4532-1460

Stötzner, M., Speth, S. and Becker, S.
Hosting-Aware Pruning of Components in Deployment Models.
DOI: 10.5220/0012671900003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 65-76
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

(i) *Hosting-Aware Pruning:* We introduce the hosting pruning condition to prune components without hostings. Therefore, we model persistent components and constraints to ensure that components and relations are not unexpectedly pruned. Moreover, we minimize the number of components to address ambiguity.

(ii) *Evaluation:* We evaluate our concepts by implementing an open-source prototype and by conducting a case study.

The remainder of this work is structured as follows. In Section 2, we provide the required background and introduce our motivating scenario. We contribute required building blocks in Section 3 and present the final method for the hosting-aware pruning of components in Section 4. In Section 5, we evaluate our concepts by implementing a prototype and by conducting a case study. Finally, in Section 6, we discuss related work and conclude our work in Section 7.

2 BACKGROUND & MOTIVATION

In the following, we introduce the required background and our motivating scenario. Based on this scenario, we illustrate the shortcomings of pruning components considering absent hostings.

2.1 Background

Manually managing the deployment of applications is error-prone and complex. Therefore, deployment technologies automate their deployment. However, we also need to manage deployment variabilities due to varying requirements.

2.1.1 Essential Deployment Models

Declarative deployment models (Endres et al., 2017) automate the deployment of applications. These models require only modeling *what* should be deployed by declaring the desired state and not *how*. Deployment technologies, such as Terraform and Ansible, automatically derive required deployment steps. However, modern applications consist of multiple components which are distributed across heterogeneous environments. Their deployment typically requires a combination of multiple deployment technologies. Therefore, *Essential Deployment Models* (Wurster et al., 2019) have been introduced. They are declarative deployment models, which can be executed while using multiple deployment technologies in combination (Wurster et al., 2021). The corresponding *Essential Deployment Metamodel (EDMM)* (Wurster et al., 2019) consists of the following core elements.

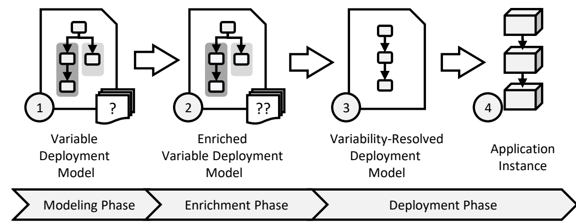


Figure 1: Managing deployment variability (simplified and based on Stötzner et al. (2023c)).

- *Components* represent application components, e.g., Node.js applications and virtual machines.
- *Relations* represent relationships between components, e.g., hosting and connection relations.
- *Properties* represent configurations of components and relations, e.g., RAM size and ports.
- *Deployment artifacts* represent implementations of components, e.g., Node.js files and binaries.

2.1.2 Variable Deployment Models

Applications must be deployed in different variants due to varying requirements, such as costs, elasticity, and enabled features. Deployment technologies typically support proprietary and non-interoperable variability modeling concepts. Since we use multiple deployment technologies in combination to deploy modern applications, we require a holistic variability modeling layer across heterogeneous deployment technologies.

We introduced *Variable Deployment Models* (Stötzner et al., 2022, 2023a) to manage such deployment variabilities based on modeling conditional EDMM elements. The overall process is given in Figure 1. In the modeling phase, the modeler creates the Variable Deployment Model of an application by modeling EDMM elements and assigning variability conditions to them. Variability conditions are expressions over variability inputs representing varying requirements. In the enrichment phase, the software component Condition Enricher automatically generates and assigns pruning conditions targeting consistency issues and semantic aspects to the elements of the Variable Deployment Model (Stötzner et al., 2023c). Therefore, the modeler must not model them. In the deployment phase, the operator assigns variability inputs, which the software component Variability Resolver uses to resolve variability automatically. The Variability Resolver evaluates all conditions, removes elements

SCIENCE AND TECHNOLOGY PUBLICATIONS

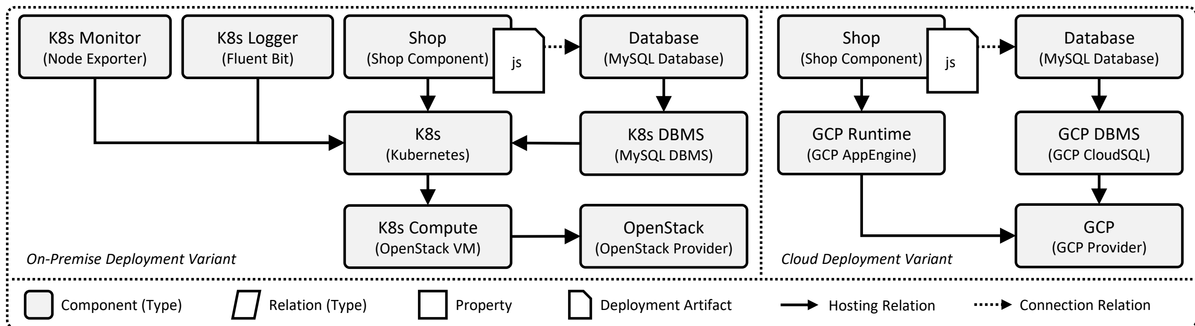


Figure 2: The motivating scenario is either deployed on-premise using Kubernetes on a local OpenStack instance (on the left) or in the cloud on the Google Cloud Platform (GCP) (on the right).

whose conditions do not hold, and conducts consistency checks. As a result, an Essential Deployment Model is generated, which is subsequently executed to deploy the application with multiple deployment technologies in combination. The corresponding *Variable Deployment Metamodel (VDMM)* (Stötzner et al., 2022, 2023a) extends EDMM as follows.

- *Conditional elements* represent elements, which might have conditions assigned, i.e., components, relations, configurations, and deployment artifacts.
- *Variability inputs* represent the context under which variability is resolved.
- *Variability conditions* represent Boolean expressions over variability inputs.
- *Manual conditions* represent manually assigned variability conditions.
- *Pruning conditions* represent automatically assigned variability conditions.

The following pruning conditions are generated and assigned to remove any inconsistent or semantically incorrect element from the deployment automatically.

- *Is any incoming relation present* for components with at least one incoming relation.
- *Is any deployment artifact present* for components with at least one deployment artifact.
- *Are the source and target present* for relations.
- *Is the container present* for properties and deployment artifacts.

2.2 Motivating Scenario

In our motivating scenario, we manage the deployment of a webshop application. The webshop consists of a shop component and a database. An overview is given in Figure 2. The webshop can be either deployed on-premise or on a public cloud provider and is based on our motivating scenario for pruning elements (Stötzner et al., 2023c). We use this scenario throughout our work to illustrate our concepts.

2.2.1 Deployment Variants

In the on-premise deployment variant, the webshop is deployed using Kubernetes on a virtual machine on a local OpenStack (OS) instance, as shown on the left of Figure 2. This deployment variant is only applicable if the virtual machine is capable of handling the expected workload and if, e.g., due to compliance reasons, the usage of a public cloud is not allowed. Due to operational requirements, a monitoring agent and a logging agent are additionally deployed on Kubernetes. These agents ship their monitoring data and logs to external servers, which are not shown.

In the cloud deployment variant, the webshop is deployed on the public cloud Google Cloud Platform (GCP), as shown on the right of Figure 2. Therefore, the shop component is deployed using GCP AppEngine, and the database using GCP CloudSQL. In this variant, we are not required to deploy a monitoring agent or a logging agent since we can rely on the respective capabilities provided by GCP.

There might be more deployment variabilities. Depending on the customer, the webshop might be differently implemented or configured (Stötzner et al., 2023a). For the sake of simplicity, we restrict our scenario to hosting variabilities.

2.2.2 Problem

The Variable Deployment Model of the webshop is given in Figure 3. The shop component and the database are present in all variants and have no conditions assigned. Some elements are automatically removed due to pruning and do not require manual conditions. For example, Kubernetes is automatically removed once all hosted components are absent. However, we must manually assign conditions to the monitoring agent and the logging agent to be only present

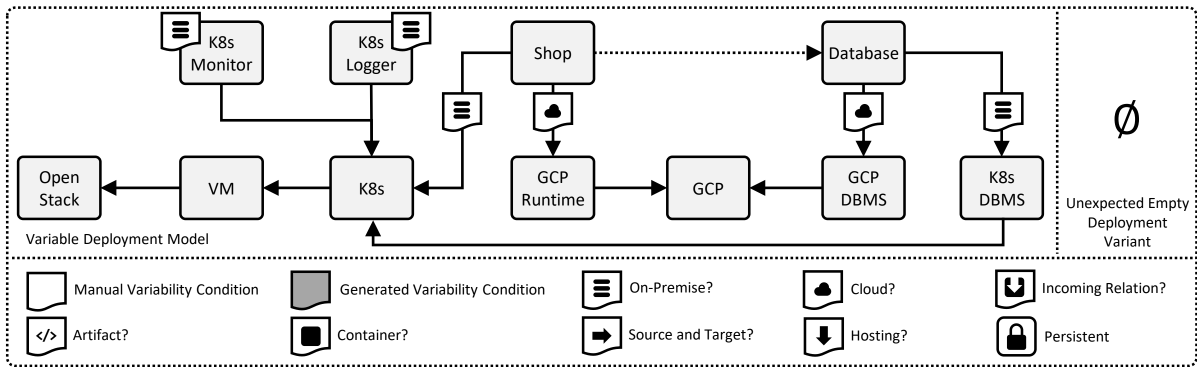


Figure 3: The Variable Deployment Model of our motivating scenario and the unexpected variant, which is derived independently of the variability inputs (simplified). The legend already standardizes the visualization of all pruning conditions.

for the on-premise deployment. This is repetitive, error-prone, and time-consuming. Thus, we automate this and prune elements without hostings.

However, simply combining a pruning condition checking for the presence of hosted components with a pruning condition checking for the presence of hosting components leads to the unexpected removal of the majority of the deployment model due to circular dependencies (Stötzner et al., 2023c). The Variability Resolver essentially tries to remove elements while complying with the conditions and might decide to remove Kubernetes. With Kubernetes being absent, the virtual machine and OpenStack are removed. When following the envisioned pruning of components without hostings, the monitoring agent, the logging agent, and the shop component are removed. As a result, every element is always pruned independently of the variability inputs. Therefore, in this paper, we present concepts to prevent this.

3 BUILDING BLOCKS

Before we extend our Pruning Method (Stötzner et al., 2023c) by the hosting-aware pruning of components, we contribute several building blocks. These blocks include, e.g., modeling constraints between elements and minimizing the number of components to remove components, which are not relevant.

3.1 Constraints

In the following, we introduce the concepts of constraints, which cross element boundaries to enforce the presence of elements. We introduce concepts to support modeling them, including the automated generation of hosting constraints.

```
1  function enrich(vdm: VariableDeploymentModel):
2    # Add relation constraints
3    for (relation of vdm.relations)
4      constraint = createRelationConstraint(relation)
5      vdm.constraints.add(constraint)
6
7    # Add hosting constraints
8    for (component of vdm.components)
9      constraint = createHostingConstraint(component)
10     vdm.constraints.add(constraint)
11
12   return vdm
```

Listing 1: The function for enriching a Variable Deployment Model with constraints.

3.1.1 Constraint Enricher

We introduce the *Constraint Enricher*. The Constraint Enricher is a software component that automatically processes a given Variable Deployment Model during the enrichment phase after the Condition Enricher. The goal is to generate constraints to reduce manual modeling efforts. For a better understanding, we briefly introduce the executed logic. The used concepts are introduced in the following sections.

The corresponding function *enrich* is given in Listing 1. On a given Variable Deployment Model, the Constraint Enricher first transforms relation constraints into variability constraints (Lines 3 to 5) to simplify modeling constraints in which a component implies a relation and then generates hosting constraints (Lines 8 to 10) to ensure that the hosting of a present component is present.

3.1.2 Variability Constraints

Modeling constraints between elements is a known concept, e.g., to model dependencies of features in feature models (Kang et al., 1990). We use this concept and introduce *variability constraints* between VDMM elements. For example, the shop component

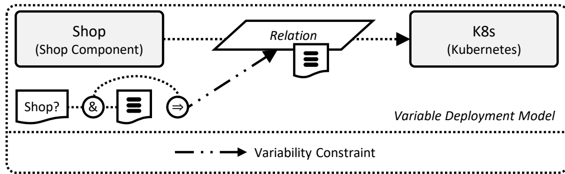


Figure 4: The Kubernetes hosting relation of the shop component is required for the on-premise variant (simplified).

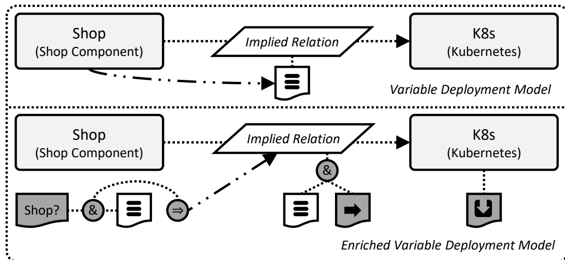


Figure 5: The Kubernetes hosting relation of the shop component modeled as implied relation (simplified).

requires the Kubernetes hosting for the on-premise variant. Therefore, we model the variability constraint, which implies the presence of the relation under the given condition, as given in Figure 4. However, we also need to model a variability condition at the relation to ensure that the relation is absent for the cloud variant. The modeled implication also ensures the presence of Kubernetes: If Kubernetes is absent, then the relation is pruned. This contradicts our modeled constraint and, thus, is not allowed.

In contrast to variability conditions, variability constraints can enforce the presence or the absence of *other* elements, while variability conditions only state the presence considering the element they are assigned to. We extend the Variability Resolver to evaluate constraints and require that all constraints must be fulfilled when resolving variability.

3.1.3 Relation Constraints

Modeling variability constraints for relations requires duplicating manual variability conditions, as shown in Figure 4. Therefore, we automate this and introduce *relation constraints* into VDMM. The Constraint Enricher derives them from the variability conditions assigned to relations, which are implied by their source. For example, the shop component implies the Kubernetes hosting relation for the on-premise deployment variant, as shown in Figure 5.

The corresponding function *createRelationConstraint* is given in Listing 2. On a given relation, the Constraint Enricher first checks if the relation is implied (Line 3) and ignores this relation if not. Otherwise, the Constraint Enricher constructs the relation constraint (Lines 6 to 9), which implies the presence of the relation if the relation source is present and manual conditions hold.

```
1   function createRelationConstraint(relation:
        Relation):
2     # Ignore relations that are not implied
3     if (!relation.isImplied()) return true
4
5     # Source and condition imply relation
6     conditions = relations.manualConditions
7     antecedent = {and: [relation.source.id,
          conditions]}
8     consequent = relation.id
9     constraint = {implies: [antecedent, consequent]}
10
11    return constraint
```

Listing 2: The function for creating the relation constraint.

3.1.4 Hosting Constraints

Each component with a hosting relation requires this hosting relation. Moreover, only a single hosting relation must be present. Manually ensuring these aspects is repetitive and error-prone. For example, we must model a corresponding constraint for the monitoring agent, logging agent, Kubernetes, etc. Therefore, we introduce *hosting constraints* into VDMM, which are automatically generated constraints.

The Constraint Enricher automatically generates and assigns hosting constraints to components with hostings. The corresponding *createHostingConstraint* is given in Listing 3. If the component has no hosting relations, then no constraint is generated (Line 3). Otherwise, all hosting relations are collected (Lines 6 to 8). To ensure that only a single hosting relation is present if the component is present, they are joined by an *xor* and implied by the presence of the component (Line 11). This also allows the Variability Resolver to select a hosting relation if no other conditions or constraints are modeled.

```
1   function createHostingConstraint(component:
        Component):
2     # Ignore components without hosting relations
3     if (component.hostingRelations.isEmpty()) return
          true
4
5     # Add hosting relations
6     hostings = []
7     for (relation of component.hostingRelations)
8       hostings.add(relation.id)
9
10    # Component implies exactly one hosting relation
11    return {implies: [component.id, {xor: hostings}]}
```

Listing 3: The function for creating the hosting constraint.

3.2 Pruning

To automatically remove components without hosts, we introduce the pruning condition checking for the presence of hostings. However, we require additional concepts such as not-prunable components and selecting the smallest deployment model.

3.2.1 Pruning Components

We extend the Condition Enricher to ignore not-prunable components and to generate conditions checking for the presence of hostings. For a better understanding, we briefly introduce the executed logic. Details are introduced in the following sections.

The corresponding function *createPruningCondition* extends the original function (Stötzner et al., 2023c) and is given in Listing 4. On a given component, the Condition Enricher checks if the component is persistent (Line 5) and ignores it. Otherwise, the Condition Enricher checks if the component has any hosting relations (Line 8). If that is the case, the pruning condition checking the presence of these relations is generated. From here on, the Condition Enricher proceeds unchanged. A pruning condition checking for the presence of any incoming relation is generated if the component has at least one incoming relation (Line 11), and a pruning condition checking for the presence of any deployment artifact is generated if the component has at least one artifact (Line 12). The component is only present if all conditions hold. Therefore, they are joined by an *and* (Line 15).

```
1  function createPruningCondition(component:
       Component):
2    conditions = []
3
4    # Ignore persistent components
5    if (component.isPersistent()) return true
6
7    # Add hosting pruning condition
8    if (!component.hostingRelations.isEmpty())
9      conditions.add(createHostingCondition(component))
10
11   # Add incoming relation pruning condition ...
12   # Add deployment artifact pruning condition ...
13
14   # All conditions must hold
15   return {and: conditions}
```

Listing 4: The function for creating the component pruning condition (based on Stötzner et al. (2023c)).

3.2.2 Hosting Pruning

Components whose hostings are absent should be automatically removed from the deployment. This is the ultimate goal of this paper. Therefore, we introduce the *host pruning condition* into VDMM, which checks *if any hosting relation is present*. The Condition Enricher generates and assigns this condition to components that have at least one hosting relation. For example, instead of manually assigning a condition to the monitoring agent checking for the on-premise deployment variant, no condition must be modeled, as shown in Figure 6.

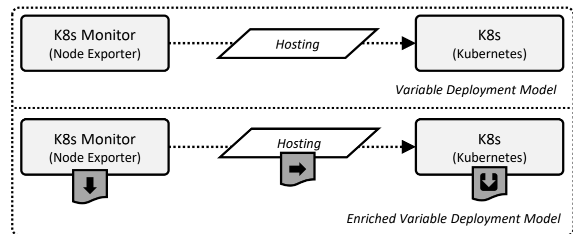


Figure 6: The monitoring agent is pruned when Kubernetes is absent (simplified).

The corresponding function *createHostingCondition* is given in Listing 5. On a given component, the Condition Enricher collects all hosting relations (Lines 3 to 5). If any hosting relation is present, the component should be present. Therefore, relations are joined by an *or* (Line 8).

```
1  function createHostingCondition(component:
       Component):
2    # Add hosting relations
3    hostings = []
4    for (relation of component.hostingRelations)
5      hostings.add(relation.id)
6
7    # A single hosting is sufficient
8    return {or: hostings}
```

Listing 5: The function for creating the pruning condition of a component with hosting relations.

3.2.3 Persistent Components

Simply combining the hosting pruning condition with the existing pruning conditions leads to circles within variability conditions: hosted components check for their hosting while hosting components check for hosted components. Such circles result in the unexpected removal of the majority of the deployment model (Stötzner et al., 2023c). When resolving variability, the Variability Resolver tries to remove elements while complying with conditions and constraints. Therefore, the Variability Resolver might decide to remove Kubernetes. As a result, the virtual machine and OpenStack are removed. With our envisioned hosting-aware pruning of components, the monitoring agent, the logging agent, and the shop component are removed. As a result, every element is pruned, and an empty unexpected variant is derived.

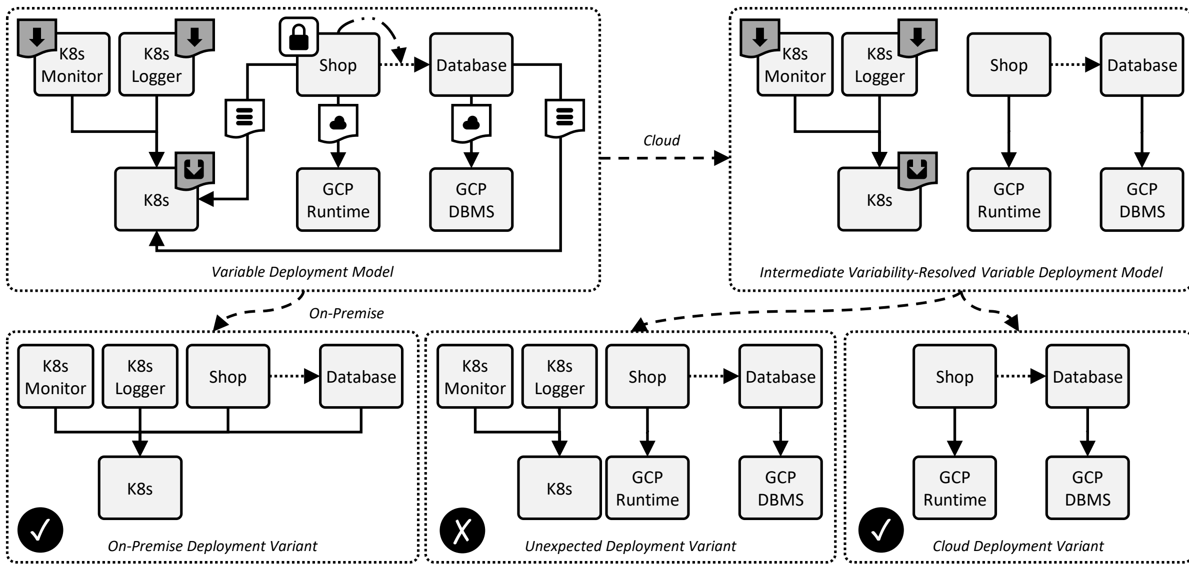


Figure 7: An isolated graph of coexistences leads to an unexpected deployment variant (simplified).

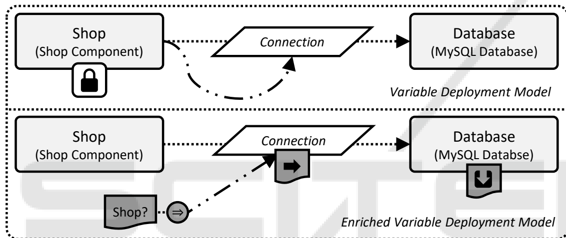


Figure 8: The shop component is modeled as a persistent component (simplified).

Therefore, we introduce *persistent components* into VDMM, which cannot be pruned. They essentially give a *presence impluse* and enforce, in combination with constraints, the presence of required elements. For example, the shop component is consumed by users and should never be removed from the deployment. Therefore, we annotate the component as *persistent*, as shown in Figure 8. As a result, the Condition Enricher ignores the component and does not generate any pruning conditions. With the presence of the shop component, the presence of the database connection is ensured. But also, the presence of the database is ensured since otherwise assigned constraints and conditions would be contradicted.

3.2.4 Optimization

With persistent components, we only prevent circles at annotated components. However, we introduced more circles and ambiguities. For example, a condition generated for the monitoring agent checks for the presence of its hosting while the pruning condition at Kubernetes checks for the presence of any incoming relations, as shown at the bottom of Figure 6.

We describe such scenarios as *coexistences* since these components only exist if their *co-component* exists while missing a presence impulse, e.g., by a persistent component or by an implied incoming relation. Therefore, these components are *not relevant* and should be automatically removed.

Considering our motivating scenario, if the on-premise deployment variant is required, then the shop component enforces the presence of Kubernetes. As a result, the monitoring and logging agents are present, as shown at the bottom left of Figure 7. If the cloud deployment variant is required, then the shop component enforces the presence of GCP. However, this results in an isolated graph of on-premise components consisting only of coexistences, as shown at the top right of Figure 7. Two possible deployment variants can be derived: the shop component hosted on GCP, as shown at the bottom right of Figure 7, and the shop component hosted on GCP along with Kubernetes, the monitoring agent, and the logging agent, as shown at the bottom middle of Figure 7. The last deployment variant is unexpected and occurs due to an underspecification of variability conditions and constraints.

One way to address this is to model additional conditions and constraints manually. However, we propose to minimize the number of components when resolving variability. Since circles are allowed to be removed, we remove them to reduce the number of components. This results in the Variability Resolver removing any coexistence and has the positive side-effect of reducing the overall deployment complexity, which typically reduces costs and execution time.

However, this does not completely address am-

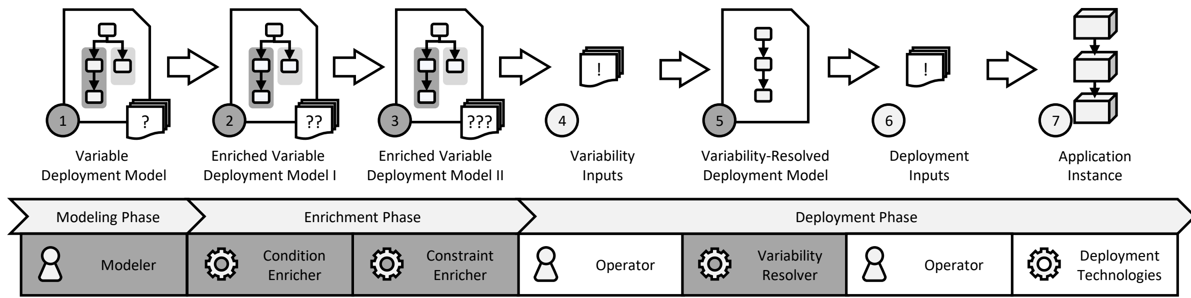


Figure 9: The Hosting-Aware Pruning Method. Differences from the original Pruning Method are presented in dark gray (figure based on Stötzner et al. (2023c)).

biguity issues. There may be multiple minimal Variability-Resolved Deployment Models. We require that the Variability Resolver aborts in such cases to ensure that variability is always resolved consistently. The approach could be extended to incorporate other elements, such as relations, to reduce this risk.

3.2.5 Circle-Based Incoming Relation Pruning

There are two different variants of the incoming relation pruning condition for components: one that leads to circles and one that mitigates them (Stötzner et al., 2023c). In the original Pruning Method, the circle-mitigating variant is used. However, since we have the concepts of optimization, persistent components, and constraints in place, we use the circle-based incoming relation pruning condition, which improves modeling in some edge cases. In these edge cases, the Variability Resolver can now decide, due to the hosting constraints and circles, how to prune. For example, an application is either hosted on a static host along with its database or on an elastic host, as shown in Figure 10. With the circle-mitigating variant, the Variability Resolver cannot resolve variability due to the hosting constraints and the fact that, by design, the elastic host cannot be pruned. However, with the circle-based incoming relation pruning, the Variability Resolver is allowed to choose between the two hostings of the app.

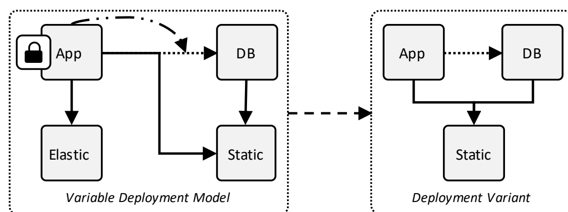


Figure 10: The Variability Resolver can derive a deployment variant (simplified).

3.3 Checks

The Variability Resolver conducts consistency checks to assess that the generated model conforms to EDMM. We extend this resolver to check that each component has exactly one deployment artifact if it had at least one deployment artifact assigned before resolving variability. In addition, the resolver checks that the same aspect considering incoming relations.

4 HOSTING-AWARE PRUNING

With our building blocks in place, we extend our original method and present the *Hosting-Aware Pruning Method*. This method has the following steps. An overview is given in Figure 9.

Step 1: Create Model. The modeler creates the Variable Deployment Model. In contrast to the original method, the modeler can use variability constraints and implied relations. Moreover, the modeler must not consider the removal of components without hostings and the modeling of hosting constraints. However, at least one persistent component is required.

Step 2: Enrich Conditions. The Condition Enricher enriches the Variable Deployment Model. In contrast to the original method, the Condition Enricher generates the hosting pruning condition for components and ignores persistent components.

Step 3: Enrich Constraints. The newly introduced Constraint Enricher enriches the Variable Deployment Model by generating relation constraints and hosting constraints.

Step 4: Assign Variability Inputs. The operator assigns the variability inputs as described in the original method. This step might also be automated.

Step 5: Resolve Variability. The Variability Resolver derives the Variability-Resolved Deployment Mode under given variability inputs. In contrast to

the original method, the Variability Resolver also considers constraints. Moreover, the Variability Resolver minimizes the number of components. To ensure that variability is not resolved differently in different runs under the same inputs, the Variability Resolver aborts if there are multiple minimums.

Step 6: Assign Deployment Inputs. The operator assigns the deployment inputs, e.g., access credentials. This step has not been explicitly mentioned before.

Step 7: Deploy Application. The Variability-Resolved Deployment Model, which is an Essential Deployment Model, is executed as described in the original method.

5 EVALUATION

We evaluate our concepts by implementing a prototype and by conducting a case study. The prototype and the model of the case study are open-source and publicly available on Zenodo[1] and GitHub[2].

5.1 Prototype

Our prototype is based on OpenTOSCA Vintner (Stötzner et al., 2022, 2023a,b,c). OpenTOSCA Vintner is a preprocessing and management layer for TOSCA (OASIS, 2020). TOSCA is an open standard for the orchestration of cloud applications. Thereby, EDMM is implemented by TOSCA Light (Wurster et al., 2020), an EDMM-compliant subset of TOSCA. Moreover, VDMM is implemented by Variability4TOSCA (Stötzner et al., 2022, 2023a,b,c), which extends TOSCA by conditional elements and the pruning of them. OpenTOSCA Vintner supports executing TOSCA models based on orchestrator plugins. Plugins for the open-source TOSCA orchestrators Unfurl[3] and xOpera[4] are implemented.

OpenTOSCA Vintner implements the Condition Enricher and Variability Resolver. We extend OpenTOSCA Vintner as follows. First, we extend the Condition Enricher implementation to generate the hosting pruning condition and to ignore persistent components. Then, we implement the Constraint Enricher. Also, we extend the Variability Resolver, which is based on the open-source satisfiability solver MiniSat[5], to respect variability constraints, to minimize the number of components and to abort if there are multiple minimums. Therefore, we assign each component a weight of one and minimize the sum of all weights using the underlying satisfiability solver.

[1]https://doi.org/10.5281/zenodo.10452506

[2]https://github.com/OpenTOSCA/opentosca-vintner

[3]https://github.com/onecommons/unfurl

[4]https://github.com/xlab-si/xopera-opera

[5]http://minisat.se

Figure 11: The hosting-aware Variable Deployment Model of our motivating scenario (simplified).

5.2 Case Study

We evaluate the technical feasibility of our concepts by conducting a case study using our prototype. Therefore, we model the variability of the webshop, resolve the variability, and deploy the webshop. We show that our concepts are functional and that our extensions to the method and prototype are working.

5.2.1 Modeling Phase

In the modeling phase, we take the modeler's role and model the Variability4TOSCA model of the webshop, as depicted in Figure 11. We must model at least one persistent component and choose the shop component. Also, we model the constraint that the shop component always requires a database connection. Last, we model conditions at the hosting relations of the shop component, checking for the corresponding variant. Due to the hosting-aware pruning, we do not need to model any other variabilities.

5.2.2 Execution Phase

In the execution phase, we take the operator's role and deploy the webshop on-premise. We use the CLI of OpenTOSCA Vintner to import the Variability4TOSCA model and to state that we require the on-premise variant. OpenTOSCA Vintner automatically enriches the Variability4TOSCA model, resolves the variability, minimizes the number of components, removes absent elements, conducts consistency checks, and generates a TOSCA model. Then, we provide additional inputs required for the deployment, e.g., access credentials to OpenStack, and OpenTOSCA Vintner instructs Unfurl to execute the generated TOSCA model. As a result, a virtual machine is started on OpenStack, and Minikube is installed as Kubernetes on this virtual machine. Finally, the shop, database, monitoring agent, and logging agent are installed on Kubernetes.

5.2.3 Discussion

In contrast to the original method, our concepts reduce the number of manual variability conditions from six to two. However, we must model one persistent component and one relation constraint. Hence, the number of variability concepts that must be modeled is reduced from six to four. These numbers depend on the explicit scenario. However, our approach is of value, whenever many coexistences occur.

5.3 Modeling Hints

We assume that, in most cases, the modeler requires relation constraints. Therefore, we recommend using relation constraints instead of relying on conditions.

We recommend modeling manual conditions at hosting relations if there are different hostings available. Otherwise, variability might not be resolved as expected. Considering the example from Figure 10, a variant in which the application is hosted on the elastic host is impossible. The database is always present and, therefore, the static host. Thus, the static hosting between the application and the static host must be present since, otherwise, there is a contradiction with the pruning conditions at this relation. A condition at the static hosting of the application must be modeled to ensure its absence when desired. A condition at the elastic hosting of the application is not sufficient.

6 RELATED WORK

Software product line engineering (Pohl et al., 2005; Pohl and Metzger, 2018) is a methodology for managing the variability of software. Typically, constraints between features are modeled as feature models (Kang et al., 1990) while reusable artifacts are implemented, whose components have conditions assigned linked to features. Based on a given feature configuration representing, e.g., a customer configuration, the product, i.e., the software, is generated. A general approach of such product lines for structural models has been proposed (Groher and Voelter, 2007; Voelter and Groher, 2007). Our method essentially implements such a product line for Essential Deployment Models while focusing on modeling reusable artifacts and the generation part.

Czarnecki and Antkiewicz (2005) use manual and default conditions to manage the variability of models. They propose post-processing the derived model to patch or simplify it. In contrast, pruning preprocesses models before variability is resolved. Wąsowski (2004) also discuss post-processing derived models, e.g., to remove unused state machines of a state chart. Post-processing methods for deployment models (Harzenetter et al., 2020; Soldani et al., 2022; Knape, 2015; Soldani et al., 2015) can be integrated into our method.

Over the last decades, there has been plenty of research in the area of product line engineering and UML (Ziadi et al., 2004; Clauß and Jena, 2001; Junior et al., 2010; Korherr and List, 2007; Dobrica and Niemelä, 2008, 2007; Sun et al., 2010). Typically, the variability of UML models is modeled using UML stereotypes and the UML Object Constraint Language. In comparison, we also use the concept of constraints to model dependencies between VDMM elements. However, we focus on simplifying modeling variability by pruning elements.

There is various research in the domain of deployment optimization (Kichkaylo and Karamcheti, 2004; Hens et al., 2007; Fehling et al., 2010; Glaser, 2016; Tsagkaropoulos et al., 2021; Zhu et al., 2021; Andrikopoulos et al., 2014; Leymann et al., 2011). Typically, these works optimize the deployment considering costs, resource consumption, energy consumption, service qualities, etc. Thereby, for example, similar to us, Hens et al. (2007) model a constraint to ensure that each component has exactly one host. In contrast to these works, we utilize optimization to address ambiguity when pruning elements.

Pruning elements simplifies modeling variability. Loesch and Ploedereder (2007) and Von Rhein et al. (2015) also propose methods to simplify variability. However, these methods are used for restructuring and debugging variability, whereas we simplify modeling variability in the first place.

Other research (Boucher et al., 2010; Dehlinger and Lutz, 2004; Krieter et al., 2023; Faust and Verhoef, 2003) uses the terminology *pruning* in the domain of product line engineering, however, with another meaning. For example, Faust and Verhoef (2003) propose a method to merge changes applied to derived variants back into the product line.

To conclude, we base on established concepts. However, we use, combine, and adapt them in our domain to manage the variability of deployment models while focusing on reducing manual modeling efforts.

7 CONCLUSION

To address the manual modeling effort of modeling hosting conditions at components, we introduce the hosting-aware pruning of components. Therefore, we annotate components to be persistent and model constraints between elements to prevent the unexpected

removal of elements of deployment models. Moreover, we minimize the number of components to address ambiguity. To evaluate the technical feasibility of our concepts, we implement a prototype and conduct a case study. Concluding, our concepts reduce the manual modeling effort and support the modeler.

However, our approach requires a variety of underlying concepts. Integrating additional concepts and ensuring that they are interoperable is challenging. To this end, we think we have reached the limits of pruning elements in deployment models.

In future work, we plan to evaluate the cognitive load when modeling. Moreover, we plan to optimize the deployment considering, e.g., deployment time.

ACKNOWLEDGEMENTS

This publication was partially funded by the German Federal Ministry for Economic Affairs and Climate Action (BMWK) as part of the Software-Defined Car (SofDCar) project (19S21002).

REFERENCES

Andrikopoulos, V., Gómez Sáez, S., Leymann, F., and Wettinger, J. (2014). Optimal Distribution of Applications in the Cloud. In *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE 2014)*, pages 75–90. Springer.

Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., and Demonceau, L. (2010). Tag and Prune: A Pragmatic Approach to Software Product Line Implementation. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 333–336. ACM.

Brogi, A., Canciani, A., and Soldani, J. (2018). Fault-aware management protocols for multi-component applications. *Journal of Systems and Software*, 139:189–210.

Clauß, M. and Jena, I. (2001). Modeling variability with UML. In *GCSE 2001 Young Researchers Workshop*. Springer.

Czarnecki, K. and Antkiewicz, M. (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Generative Programming and Component Engineering*, pages 422–437, Berlin, Heidelberg. Springer.

Dehlinger, J. and Lutz, R. R. (2004). Software fault tree analysis for product lines. In *Eighth IEEE International Symposium on High Assurance Systems Engineering, 2004. Proceedings.*, pages 12–21.

Dobrica, L. and Niemelä, E. (2007). Modeling Variability in the Software Product Line Architecture of Distributed Services. In *Proceedings of the 2007 International Conference on Software Engineering Research & Practice, SERP*, pages 269–275. CSREA Press.

Dobrica, L. and Niemelä, E. (2008). A UML-Based Variability Specification For Product Line Architecture Views. In *Proceedings of the Third International Conference on Software and Data Technologies*. SciTePress.

Endres, C., Breitenbücher, U., Falkenthal, M., Kopp, O., Leymann, F., and Wettinger, J. (2017). Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS 2017)*, pages 22–27. Xpert Publishing Services.

Faust, D. and Verhoef, C. (2003). Software product line migration and deployment. *Software: Practice and Experience*, 33(10):933–955.

Fehling, C., Leymann, F., and Mietzner, R. (2010). A Framework for Optimized Distribution of Tenants in Cloud Applications. In *Proceedings of the 2010 IEEE International Conference on Cloud Computing (CLOUD 2010)*, pages 1–8. IEEE.

Glaser, F. (2016). Domain Model Optimized Deployment and Execution of Cloud Applications with TOSCA. In *System Analysis and Modeling. Technology-Specific Aspects of Models*, pages 68–83, Cham. Springer International Publishing.

Groher, I. and Voelter, M. (2007). Expressing Feature-Based Variability in Structural Models. In *Workshop on Managing Variability for Software Product Lines*.

Guerriero, M., Garriga, M., Tamburri, D. A., and Palomba, F. (2019). Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 580–589.

Harzenetter, L., Breitenbücher, U., Falkenthal, M., Guth, J., and Leymann, F. (2020). Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration. In *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*, pages 40–49. Xpert Publishing Services.

Hens, R., Boone, B., de Turck, F., and Dhoedt, B. (2007). Runtime Deployment Adaptation for Resource Constrained Devices. In *IEEE International Conference on Pervasive Services*, pages 335–340. IEEE.

Junior, E. A. O., de Souza Gimenes, I. M., and Maldonado, J. C. (2010). Systematic Management of Variability in UML-based Software Product Lines. *J. Univers. Comput. Sci.*, 16(17):2374–2393.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.

Kichkaylo, T. and Karamcheti, V. (2004). Optimal resource-aware deployment planning for component-based distributed applications. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 150–159. IEEE.

Knape, S. (2015). Dynamic Automated Selection and Deployment of Software Components within a Heteroge-

neous Multi-Platform Environment. Master's thesis, Utrecht University.

Korherr, B. and List, B. (2007). A UML 2 Profile for Variability Models and their Dependency to Business Processes. In *18th International Workshop on Database and Expert Systems Applications (DEXA 2007)*, pages 829–834.

Krieter, S., Krüger, J., Leich, T., and Saake, G. (2023). VariantInc: Automatically Pruning and Integrating Versioned Software Variants. In *Proceedings of the 27th ACM International Systems and Software Product Line Conference - Volume A*, SPLC '23, page 129–140. ACM.

Leymann, F., Fehling, C., Mietzner, R., Nowak, A., and Dustdar, S. (2011). Moving Applications to the Cloud: An Approach based on Application Model Enrichment. *International Journal of Cooperative Information Systems*, 20(3):307–356.

Loesch, F. and Ploedereder, E. (2007). Optimization of Variability in Software Product Lines. In *11th International Software Product Line Conference (SPLC 2007)*, pages 151–162. IEEE.

OASIS (2020). *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS).

Oppenheimer, D. (2003). The importance of understanding distributed system configuration. In *Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop*.

Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do internet services fail, and what can be done about it? In *4th Usenix Symposium on Internet Technologies and Systems (USITS 03)*.

Pohl, K., Böckle, G., and van der Linden, F. (2005). *Software Product Line Engineering*. Springer Berlin Heidelberg.

Pohl, K. and Metzger, A. (2018). *Software Product Lines*, pages 185–201. Springer International Publishing, Cham.

Soldani, J., Binz, T., Breitenbücher, U., Leymann, F., and Brogi, A. (2015). ToscaMart: A method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406.

Soldani, J., Breitenbücher, U., Brogi, A., Frioli, L., Leymann, F., and Wurster, M. (2022). Tailoring Technology-Agnostic Deployment Models to Production-Ready Deployment Technologies. In *Cloud Computing and Services Science*. Springer.

Stötzner, M., Becker, S., Breitenbücher, U., Kálmán, K., and Leymann, F. (2022). Modeling Different Deployment Variants of a Composite Application in a Single Declarative Deployment Model. *Algorithms*, 15(10):1–25.

Stötzner, M., Breitenbücher, U., Pesl, R. D., and Becker, S. (2023a). Managing the Variability of Component Implementations and Their Deployment Configurations Across Heterogeneous Deployment Technologies. In *Cooperative Information Systems*, pages 61–78, Cham. Springer Nature Switzerland.

Stötzner, M., Breitenbücher, U., Pesl, R. D., and Becker, S. (2023b). Using Variability4TOSCA and OpenTOSCA Vintner for Holistically Managing Deployment Variability. In *Proceedings of the Demonstration Track at International Conference on Cooperative Information Systems 2023*, volume 3552 of *CEUR Workshop Proceedings*, pages 36–40. CEUR-WS.org.

Stötzner, M., Klinaku, F., Pesl, R. D., and Becker, S. (2023c). Enhancing Deployment Variability Management by Pruning Elements in Deployment Models. In *Proceedings of the 16th International Conference on Utility and Cloud Computing (UCC 2023)*. ACM.

Sun, C., Rossing, R., Sinnema, M., Bulanov, P., and Aiello, M. (2010). Modeling and managing the variability of Web service-based systems. *Journal of Systems and Software*, 83(3):502–516.

Tsagkaropoulos, A., Verginadis, Y., Compastié, M., Apostolou, D., and Mentzas, G. (2021). Extending TOSCA for Edge and Fog Deployment Support. *Electronics*, 10(6).

Voelter, M. and Groher, I. (2007). Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *11th International Software Product Line Conference (SPLC 2007)*, pages 233–242. IEEE.

Von Rhein, A., Grebhahn, A., Apel, S., Siegmund, N., Beyer, D., and Berger, T. (2015). Presence-Condition Simplification in Highly Configurable Systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 178–188. IEEE.

Węsowski, A. (2004). Automatic Generation of Program Families by Model Restrictions. In *Software Product Lines*, pages 73–89. Springer.

Wurster, M., Breitenbücher, U., Brogi, A., Diez, F., Leymann, F., Soldani, J., and Wild, K. (2021). Automating the Deployment of Distributed Applications by Combining Multiple Deployment Technologies. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science*. SciTePress.

Wurster, M., Breitenbücher, U., Falkenthal, M., Krieger, C., Leymann, F., Saatkamp, K., and Soldani, J. (2019). The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Software-Intensive Cyber-Physical Systems*, 35:63–75.

Wurster, M., Breitenbücher, U., Harzenetter, L., Leymann, F., Soldani, J., and Yussupov, V. (2020). TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies. In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 216–226. SciTePress.

Zhu, L., Giotis, G., Tountopoulos, V., and Casale, G. (2021). RDOF: Deployment Optimization for Function as a Service. In *IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 508–514. IEEE.

Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2004). Towards a UML Profile for Software Product Lines. In *Software Product-Family Engineering*. Springer.