Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Transformation of TOSCA to Natural Language Texts

Marco Radic

Course of Study: Informatik

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Jasmin Guth, M.Sc.

Commenced: April 4, 2017

Completed: October 4, 2017

CR-Classification: I.2.7, K.6.2, D.2.7

Abstract

Cloud computing changes the way businesses plan, use and manage their IT systems and resources. Different cloud providers offer distinctive interfaces for the deployment and management of applications in their respective cloud environments.

The organization OASIS addresses these circumstances with the *Topology and Orchestration Specification for Cloud Applications* (TOSCA). This standard offers a language to express applications as directed graphs and their management behavior in a standardized and vendor-independent manner.

In numerous roles in the development, a textual description of the application, its entities and their relationships, for instance to serve as textual documentation, is of use. The TOSCA standard places no restriction on the complexity of a topology graph. Therefore, a textual representation of the graph can also get arbitrarily large and complex. Additionally, every change has to be reflected in the documentation accordingly. Consequently, an automated approach to the generation of such textual representations is preferable.

This work describes a concept for the automated generation of textual descriptions of TOSCA topology graphs. This is accomplished by combining typical tasks from natural language generation with domain-specific information in order to generate appropriate textual descriptions. The concept is implemented in a prototype and validated in a use-case scenario.

Kurzfassung

Cloud Computing verändert die Planung, den Einsatz und das Management von informationstechnologischen Systemen in Unternehmen. Verschiedene Anbieter von Cloudservices bieten unterschiedliche Schnittstellen, um Deployment und Management von Applikationen in ihrer angebotenen Cloudumgebung zu ermöglichen.

Die Organisation OASIS adressiert diesen Sachverhalt mit der *Topology and Orchestration Specification for Cloud Applications* (TOSCA). Dieser Standard bietet eine Sprache, um Applikationen als gerichteten Topologiegraphen und ihr Managementverhalten standardisiert und anbieterunabhängig zu beschreiben.

In den unterschiedlichen Rollen der Entwicklung ist oftmals eine textuelle Beschreibung der Applikation, ihrer Komponenten und deren Beziehungen untereinander, beispielsweise zu Dokumentationszwecken, wünschenswert. Da der TOSCA Standard keine Restriktionen bezüglich der Komplexität eines Topologiegraphen setzt, kann auch eine textuelle Repräsentation eines solchen Graphen beliebig komplex werden. Zudem muss jede Änderung entsprechend in der textuellen Dokumentation angepasst werden. Daher ist ein automatisiertes Verfahren zu Generierung solcher textueller Beschreibungen erstrebenswert.

Diese Arbeit beschreibt ein Konzept zur automatisierten Generierung textueller Repräsentationen von TOSCA Topologiegraphen. Dazu werden Aufgaben und typische Merkmale aus dem Bereich der natürlichsprachlichen Generierung mit domänenspezifischen Informationen angereichert, um natürlichsprachliche Beschreibungen zu generieren. Das Konzept wird prototypisch implementiert und in einem Beispielszenario validiert.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

1 Introduction

Cloud Computing is a paradigm that fundamentally changes how computing resources are treated. Previously, businesses had to invest time and money into the purchase and maintenance of their own hardware. With cloud computing, instead of maintaining the whole infrastructure stack themselves, businesses can treat these resources abstractly as utilities [AFG+10]. This abstraction provides more flexibility [MG+11] and enables operations to focus on their business competences [LF09].

However, different cloud providers maintain distinctive ways to deploy and manage applications in cloud environments. This problem is further accompanied by the fact that dependence on the interfaces of a particular vendor is to be avoided to prevent lock-in situations.

With the TOSCA (Topology and Orchestration Specification for Cloud Applications) Standard, OASIS introduced a language that aims to standardize the way cloud applications are deployed and managed in cloud environments [BBH+13]. Using the standard, cloud applications are characterized using topology graphs to describe their components, properties and express relationships between them. In addition, the management is described using workflow plans. TOSCA aims to provide portability and independence of cloud providers.

In different stages of the development and management of a cloud applications, numerous roles are introduced to the application in different situations. Depending on the context, a textual documentation of the topology might be desired. As there are no size or complexity restrictions imposed on topology graphs, textual representations can get arbitrarily big and complex. Generating these textual descriptions as documentation of work is oftentimes tedious, as every change has to be reflected accordingly in the documentation. Therefore, a systematic approach for the automated generation of such texts is desirable.

In this work, we propose an approach that enables automated generation of textual descriptions of TOSCA topology graphs by addressing common tasks in Natural Language generation with domain-specific solutions. To validate the approach, a prototype was implemented, which takes a topology graph as input and outputs a textual description of it in English language. Furthermore, the approach is described in detail by running it on an exemplary topology graph.

Outline

This thesis is sectioned into the following Chapters:

Chapter 2 – Foundations: This Chapter introduces important foundations and concepts for TOSCA and Natural Language Generation that are important in this work.

Chapter 3 – Related Work: In this Chapter, related work in the context of this thesis is presented.

Chapter 4 – Approach: This Chapter describes the approach and conceptual work of this thesis, providing the theoretical knowledge for the prototypical implementation.

Chapter 5 – Implementation: This Chapter presents the implementation of a prototype following the conceptual work of the previous chapters.

Chapter 6 – Validation: In this Chapter, we validate the feasibility of the approach with an exemplary topology graph.

Chapter 7 – Conclusion and Future Work In this Chapter, we provide a conclusion and outlook for further work on this topic.

2 Foundations

In this Chapter, we introduce foundational knowledge that is required to understand important terms and concepts used throughout this thesis.

An introduction to TOSCA is provided in Section 2.1. Section 2.2 provides an overview for the topic of Natural Language Generation.

2.1 TOSCA

The *Topology and Orchestration Specification for Cloud Applications* (TOSCA) Standard by the Organization for the Advancement of Structured Information Standards (OASIS) provides a language standard for the description of enterprise cloud applications and their management and orchestration behavior [OAS13]. TOSCA allows for the composition of such applications through *Service Templates*. A Service Template contains two main concepts. One is a structural model of the service in the form of a topology graph, as well as *Plans* for managing and orchestrating the application. Cloud applications are abstracted to graphs, consisting of components as nodes, and respective relationships between them as edges. With topology graphs, TOSCA provides an abstraction for cloud applications, which together with workflow-based management plans increases portability and encourages reusability [BBKL14]. In order to ship the Templates and their implementations in a portable manner, TOSCA defines the *Cloud Service Archive* (CSAR) as a self-contained package. The general scheme of a Service Template is depicted in Figure 2.1.

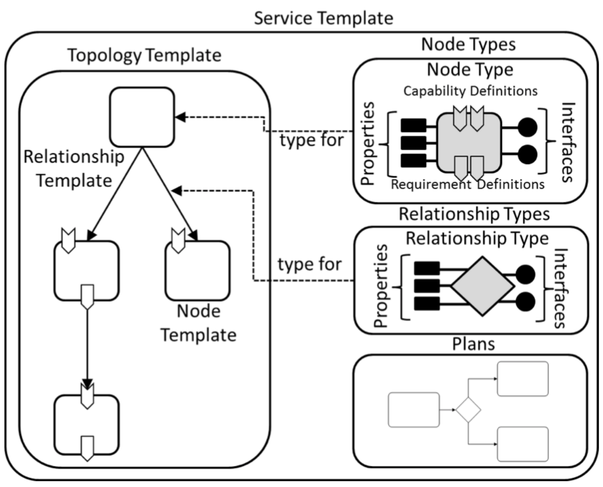

Figure 2.1: Generalized *ServiceTemplate* illustration [OAS13]

The topologies are described in *TopologyTemplates*, which represent directed graphs. Entities are represented as *NodeTemplates* and act as vertices. The edges are relationships between entities, which are represented as *RelationshipTemplates*. They are typed by *NodeTypes* and *RelationshipTypes*, respectively, which define the sets of properties and interfaces they offer. In addition, *NodeTypes* can explicitly express their capabilities they offer and requirements they need, for relationships to be established. The typing mechanism allows for inheritance, comparable to inheritance constructs in object-oriented programming languages.

Figure 2.2 depicts an example for a topology graph. Nodes are displayed with their ID, their respective *NodeTypes* are denoted using squared brackets. The *PredictionService* and *DeliveryService* are both applications that run on the Python Interpreter and therefore depend on a suitable Python installation. This semantic information is reflected in the topology using the *dependsOn* relationship between the two Services and the Python installation. All other relationships in the topology are *hostedOn* relationships. The Python interpreter and a Flink instance are hosted on a Ubuntu Virtual Machine installation. The Virtual Machine runs on a Hypervisor.

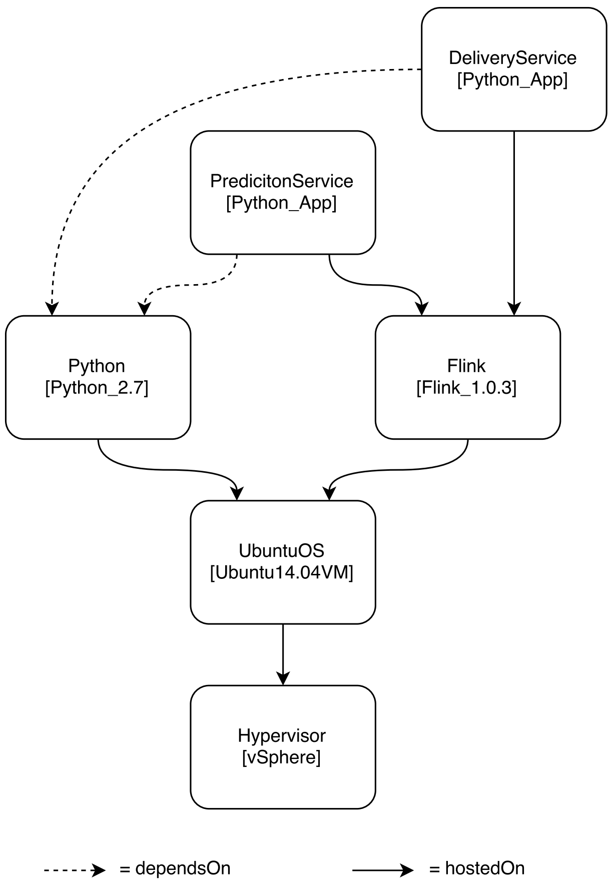

Figure 2.2: Example topology adapted from the OpenTOSCA Website [IAA17]

2.2 Natural Language Generation

Natural Language Generation (NLG) is regarded as a subfield of both computational linguistics and artificial intelligence and focuses on (semi-)automated generation of texts in natural language by computer systems [RD97]. In the following, a typical general architecture for Natural Language Generation systems is described.

Pipeline Architecture

For the implementation of NLG systems, Reiter et al. [RDF00] propose a module-based pipeline, where individual tasks are bundled in modules. The individual tasks each bundle a set of problems that the system might impose. While the system can implement every task individually, the authors emphasize that the pipeline architecture can also be seen as an outline with important topics to consider when implementing such a system. As a consequence, individual tasks might be modified, merged, swapped or removed entirely when appropriate. Nonetheless, when given the task of generating natural language for a given input specification, it is considered a useful measure to consider solutions for these imposed tasks when building a NLG system. The composition of the modules and their respective tasks are depicted in Figure 2.3. In the following, every task is described shortly.

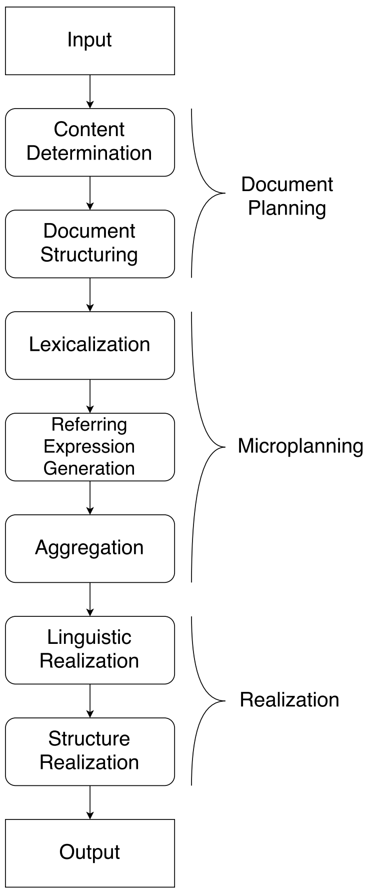

Figure 2.3: NLG Pipeline proposed by Reiter et al. [RDF00]

Document Planning

The *Document Planning* module generates an outline for the document to be generated. It determines the information to include and how that information is to be structured.

Content Determination The task of *Content Determination* is to identify the information that is to be communicated from the input and therefore precedes other tasks.

Depending on the application, this task has to incorporate different factors. The communicative goal has to be considered, which defines the topic that the output is required to cover. Furthermore the content itself is analyzed regarding its level of detail. Oftentimes, an initial level of knowledge in the specific domain is assumed or determined for readers of the text. This leads to decisions regarding the level of detail of the content to be generated. Moreover, constraints on the output or the system itself can change the requirements of content determination. Examples are space constraints on the length of the output or time constraints on the processing time of the NLG system. As these factors are decided, the information source, which acts as the input, is analyzed and processed into chunks of information. At this point, concrete input information is abstracted to an internal representation, which can be manipulated by the system.

Document Structuring While the output of the previous step provides the content to include, this content is unordered. As texts can be seen as ordered structures, such as sentences or paragraphs, ordering is established by the *Document Structuring* task. This structure is highly dependent on the application, therefore domain-specific information about the input is often necessary. A general objective of this task can be formulated as establishing an ordering over the content elements, such that the resulting text is intuitive and accessible to read. Common strategies for structuring include proper expression of hierarchical concepts, changing level of detail and grouping of similar content, e.g. in a paragraph.

Microplanning

Microplanning takes on the idea of the previous module, but takes it to a more granular level of detail. Instead of document-level, this module operates on a level of sections and sentences. The information that was previously determined is structured and modified in such a way that it can be expressed in natural language.

Lexicalization The task of *Lexicalization* is to find the correct wording for the content that was previously determined. This includes e.g. nouns, verbs and adjectives in order to describe the content in the target language. In addition, syntactic structures from the language are also incorporated.

Referring Expression Generation Every time an entity from the content is referred to in text, it is referenced by a referring expression. As an example, consider the following sentences “*John bought milk. He drove home right after.*” Both “*John*” and “*he*” are expressions referring to John. Oftentimes, an entity might be referred to in different ways, depending on the contextual information available. As the content is determined and domain-specific information is available, the limited context available is used to limit the range of possible referring expressions. A problem that might occur in this task is the avoidance of ambiguities in the references. A referring expression should be deterministically traceable to an entity [DR95]. Generally, in a NLG system, and *initial reference* is to be differentiated from a *subsequent reference*. An initial reference occurs when an entity is introduced, e.g. mentioned explicitly for the first time. Subsequent references to the same entity can account for the fact that an initial reference occurred in the context. The references used range from explicit descriptors of entities, pronouns that identify an entity in the context, to a set of attributes that identifies that entity without ambiguity [DR95].

Aggregation *Aggregation* is a structural task that determines the structures to use to express certain sets of information and removes redundancy [Dal99]. This includes grouping information into e.g. paragraphs or sentences. Several pieces of information can e.g. be combined in a single sentence. For example "*John was born in England. John was raised in Italy.*" could for instance be combined into the sentence "*John was born in England and raised in Italy.*" The degree of grouping messages into structures is content- and domain-specific.

Realization

At this point in the pipeline, the whole document is present in an abstract structure inside the system. For mapping these structures to a document instance, a *Realizer* is used. Depending on design decisions in previous tasks, the level of realization that has to be applied to the internal structures differs.

Linguistic Realization While previous tasks assembled internal structural representations of text, the *Linguistic Realizer* maps these representations to sentences. For this, morphological and orthographical rules of the target language are inflicted along with correct syntax and grammar rules. The amount of realization done in this step is dependent on the level of abstractness of the internal representations established by previous tasks. For instance, a system using templates (Section 2.2) might perform considerably less work on ensuring correct grammar as the templates are available preconfigured.

Structure Realization Previously, the Linguistic Realizer constructed a text out of internal representation. This text is still regarded as abstract, as it needs to medium or document to be placed in. The *Structure Realizer* is concerned with this task. Conceptually the Linguistic Realizer builds components of a text, such as sentences or paragraphs. The Structure Realization task puts these language constructs into documents, which is the output of the system. This corresponds to assembling a document following conventions of document formats, e.g. the source code of a LaTeX-document or correct HTML-tags.

Templating

Sometimes, the text in specific domains is of rather static nature. Patterns in texts can be identified and subsequently extracted for reusability. These extracted parts of sentences can be used as sentence templates. When incorporating NLG into a system, a fully featured linguistic systems is often not necessary, as argued in [Rei95] and [VTK05]. Therefore, if a fully linguistic system is not needed or too costly, templating presents a viable alternative [RM93].

They are also referred to as *templated sentence plans*, as they provide an organized way to plan out the sentence structure in a generalized manner. The use of templates does not necessarily exclude the possibility of making use of linguistic systems, as there exist *hybrid* systems that make use of templates, as well as linguistic features such as dynamic morphology to generate grammatically sound text [VTK05].

3 Related Work

In this Chapter, related work to the generation of natural language texts in the context of information technological systems is presented.

XtraGen

Stenzhorn [Ste02] introduces the framework *XtraGen*, which uses Java and XML messages to construct an easily integratable solution for applications generating text. The author argues that the majority of software systems that need to generate text do not need complete linguistic systems, and therefore uses a concept of expendable templates that can include predefined text as well as rules and constraints for dynamic generation of text. For this reason, they can be considered *hybrid* templates, as linguistic information can be incorporated into this structure. These templates are defined using XML and makes use of its expressive nature. Preset conditions define the cases for when it is appropriate to use the template. The system can be considered a general-purpose NLG system, as it makes no assumptions about domain-specific information of the input.

Generating Natural Language Texts from Business Process Models

Leopold et al. [LMP12] present an approach for the transformation of BPMN 2.0[1] process models into natural language texts. Processes in specific domains are often designed by domain experts. These experts may, although, lack knowledge of specific conventions and notations in the process modeling language, which motivates an automated generation of textual representations of these process models. This is accomplished by implementing the pipeline-architecture approach by Reiter et al. [RDF00]. In the Text Planning step,

[1]http://www.bpmn.org

relevant linguistic information is extracted from the process model using a part-of-speech tagger and the semantic net *WordNet* [Mil95]. The order is derived by splitting the processes into fragments in order to restructure them in tree form. This ordering provides a consistent approach for Document Structuring. Microplanning as well as Realization are accomplished by making use of closed-source, commercial tools. Created documents do not only include raw text, but are enriched by the inclusion of lists and paragraphs to provide flexibility in expressing complex subgraphs of models.

ModelExplainer

The MoDex system, developed by Lavoie et al. [LRR96], aims to generate a human-readable linguistic representation of models based on the Object Definition Language standard. It is argued that different requirements analysts as well as domain experts work on modeling using graphical notation. A textual representation can aid in complementing such a graphical representation in order to prevent semantic errors during the modeling and in later stages of development. The system generates HTML files that contains hyperlinks for interactive navigation throughout different text paragraphs. Other notable features of the generated documents are the inclusion of examples of instances of the model, as well as negative example to emphasize the semantic implications of the model to prevent errors in further development. The authors argue that, since the modeling language is domain-independent, MoDex is domain independent as well. In order to adapt to this, the authors propose to incorporate annotations and additional free text from user entry to improve the quality and semantic accuracy of the generated texts.

Generating Natural Language specifications from UML class diagrams

Meziane et al. [MAA08] present an approach to textual representations of UML class diagrams with the help of *GeNLangUML*. The authors argue that the need for natural language representations is often need in Software Engineering for documentation and maintenance purposes. The implementation of *GeNLangUML* is template based in regards to sentence plans and realization, because sophisticated grammar formalisms are not needed to express the class relations. This is further emphasized by the authors, as they claim that the language used in diagrams is a small subset of the English language.

NaturalOWL

Androutsopoulos et al. [GA07], [ALG13] propose their approach to verbalization of Semantic Web OWL ontologies, with their system NATURALOWL. The pipeline approach by Reiter et al. [RDF00] was implemented here again. What is remarkable about their approach, is the level of influence that the author of an ontology model has in the generation of the textual representation. They accommodate the domain-independence and generality of OWL ontologies by actively relying on additional annotations by the ontology's author in order to generate more coherent and semantically sound text. For example, the author has to provide inflected forms of every verb, noun or adjective used in an OWL message triple for the triple to be correctly realized in text later. In addition to this, custom sentence plans can be provided, which lead to more expressive sentences in the text.

Automatic Documentation Generation via Source Code Summarization of Method Context

McBurney et al. [MM14] address the problem of automated documentation generation for source code. They observe that manually created documentation of source code oftentimes has a lower priority as the development progresses. In addition, the documentation provided sometimes lacks useful insights for further development. An automated approach to documentation generation can counteract this, by providing standardized descriptions of methods for the documentation that automatically changes alongside with changes in the source code. A notable difference in their approach to previous work on this topic is the incorporation of the context of a method in the software system in the textual documentation for a method. For a given method, they analyzed the call graph of the system using a modification of the PageRank algorithm to determine the methods that call the original method, as well as the methods that get called by it. Then, the system extracts semantic information about the method by relying on meaningful and consistent naming, following the *Software Word Usage Model* naming conventions. This information is then propagated to a NLG system that follows the typical pipeline approach by Reiter and Dale [RDF00].

4 Approach

This Chapter describes an approach to generate textual descriptions from TOSCA topology graphs.

The typical NLG pipeline tasks are adjusted to suit TOSCA topology-specific characteristics. An overview of the approach is provided in Figure 4.1. First, the individual components from the topology are extracted from the source files of the input. Then, an internal graph representation is constructed from these components. The system then performs an algorithm on the graph in order to determine the structure of the text (Section 4.3). The result of the algorithm is a list of individual messages, which are then processed in the Microplanning task. After further processing and composition by the Realizer task, the output is a textual description of the topology.

4.1 TOSCA Topologies and Domain

4.1.1 Definitions

The TOSCA specification [OAS13] defines a topology graph as part of a *Service Template*. We define the topology graph as a directed graph $G = (V, E)$ by extending it with the following definitions:

$$\begin{aligned}
V :=& \{\textit{NodeTemplates} \text{ in the } \textit{TopologyTemplate}\} \\
E \subseteq& \{(u,v) \mid \quad u, v \in V\} \\
E_{inv} :=& \{(v,u) \mid \quad (u,v) \in E\} \\
\mathsf{Edgelabels} :=& \{\text{Types of } \textit{RelationshipTemplates}\} \\
\mathsf{Nodelabels} :=& \{\text{Names of } \textit{NodeTemplates}\} \\
\mathsf{relationship} :& E \to \mathsf{Edgelabels} \\
\mathsf{source} :& E \to V, (u,v) \mapsto u \\
\mathsf{target} :& E \to V, (u,v) \mapsto v
\end{aligned}$$

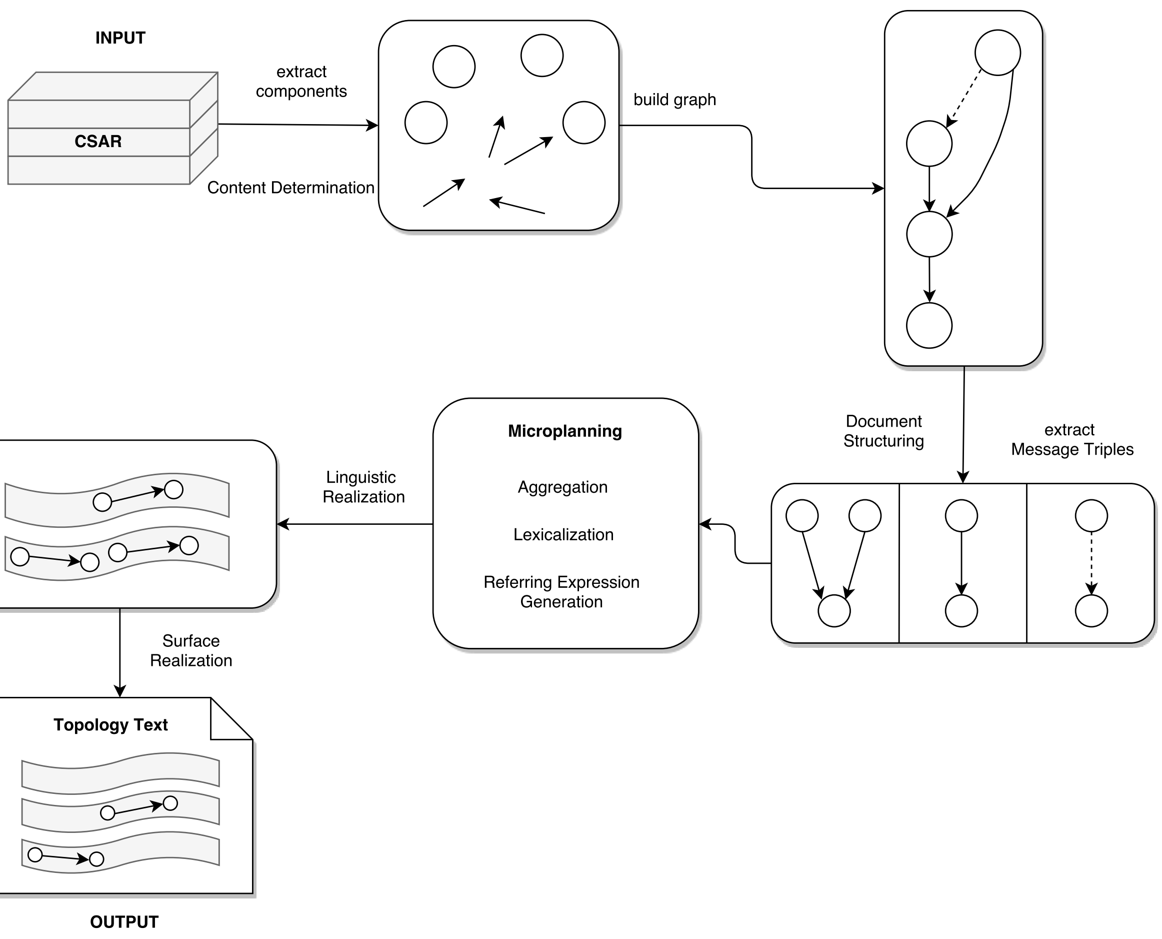

Figure 4.1: Overview of the Approach

4.1.2 Input: TOSCA Topology

The first task of the pipeline is to determine the content from the given input data. The systems input is a CSAR, that according to the TOSCA specification [OAS13], has the following structure:

```
.csar
 ├─ /TOSCA-Metadata
 │   └─ TOSCA.meta
 ├─ Definitions
 ├─ Plans
 └─ ...
```

In a CSAR file, a `TOSCA.meta` has to be present in the `TOSCA-Metadata/` directory. It describes metadata and organization about the CSAR and its components. The rest of the files are `.tosca` files that include all the elements of a *ServiceTemplate* organized in directories. These files are collected and transformed into an internal graph representation of NodeTemplates (Listing 4.1) and RelationshipTemplates (Listing 4.2). This transformation allows for the following steps to make use of characteristics of graphs, such as well known algorithms.

4.2 Formalizing Relationships as Message Triples

During the individual pipeline tasks, a system-specific internal representation of the document to be created is typically used. Reiter et al. [RDF00] propose the concept of a `Message` data structure, which contains all the accumulated information and transformations of previous pipeline tasks. In NATURALOWL [ALG13], edges of ontology graphs are transformed to `Message Triples`, which in short contain information about the source and target of an ontology relationship. For this work, we adapt this concept and introduce Message Triples to our system. Similarly to its use in NATURALOWL, Message Triples are used to represent edges in the topology graph. The data structure aims to facilitate bundling similar content by allowing to collect edges with the same relationship type into a single Message Triple.

A Message Triple is a three-tuple, that, given $e \in E$, is of the following form:

$$<\text{source}(e), \text{relationship}(e), \text{target}(e)>$$

With the introduction of Message Triples, a naive approach to building sentences could be to generate one Message Triple per edge in the topology graph. The system could then generate one sentence per Triple generated. While this is possible, it does not provide additional value over storing the individual edges themselves, as their source node, type of relationship and target node can easily be determined. In addition to storing individual relationships as Message Triples, an *and*-operator to bundle a set of nodes is allowed as the first argument. This allows for multiple, similar relationships that have the same target and are of the same type to be expressed together for more fluency in the output. In addition, this allows for Message Triples to be aggregated, which corresponds to sentence Aggregation and therefore partially covers the Aggregation pipeline task. Consider the example depicted in Figure 4.2 (r.).

The corresponding Message Triple would be of the following form:

$$<\text{and}(v_1, \ldots, v_{n-1}), \text{relationship}(e), v_n>$$

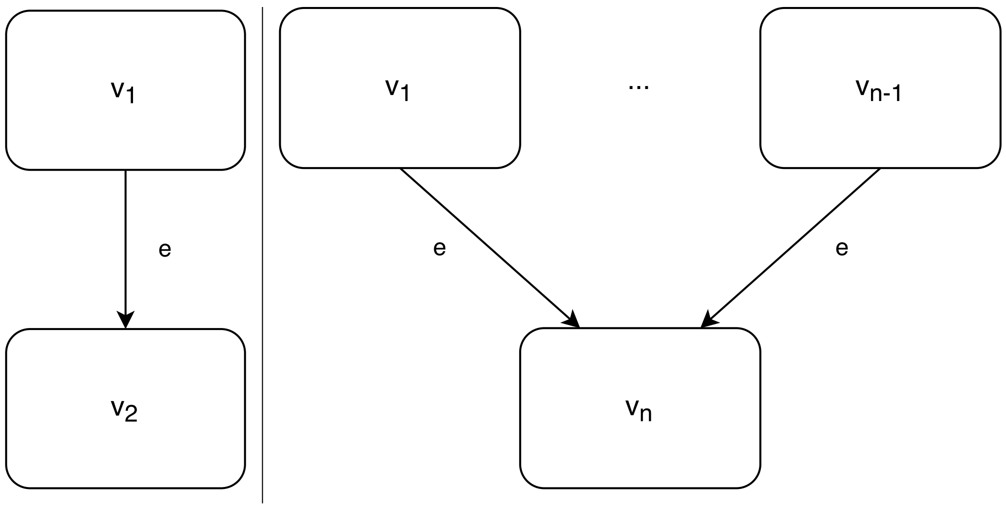

Figure 4.2: Simple Message Triple consisting of one relationship (l.) and multiple of the same relationship type (r.)

4.3 Determination of Order and Aggregation of Message Triples

Having defined an intermediate representation of information in the form of Message Triples, this informational content has to be brought into an order according to the Document Structuring task. For that, we propose an algorithm that is a modification of the Depth-First Search algorithm for graphs [Tar72]. The procedure aims to collect all edges in the topology graph in a specific order and merges multiple edges into Message triples whenever possible. This provides a linear order of Message Triples. The corresponding pipeline task is Document Structuring, as the algorithm provides a structured order to Message Triples that are eventually transformed into the output text. The algorithm is listed in Algorithm 4.1.

The input for the algorithm is assumed to be a topology graph $G(V, E)$ along with a root node, which is usually the most *low-level* element in the topology, for instance a Hypervisor or Virtual Machine. In the first step, the given topology graph is inverted, replacing E with E_{inv} for the rest of the procedure. This is done to incorporate the observation, that topology graphs tend to “spread” from the root-node, as branching occurs. After that, the list that holds the resulting Message Triples is initialized, alongside with a Stack. The root-Node is then pushed onto this newly generated Stack. The algorithm then enters a loop, where, similar to iterative Depth-First Search, the Stack’s top node is popped. If this node was not yet visited, the algorithm marks it as visited and collects all outgoing edges of that node in Message Triples, which are added to the list of Triples. All target nodes of these outgoing edges are then pushed to the Stack.

Algorithm 4.1 TripleCollection-Algorithm

```
procedure CollectTriples(topology, root)
    invert(topology)                                    // $E$ is replaced by $E_{inv}$
    Triples ← List
    S ← EmptyStack
    S.push(root)
    while S is not empty do
        $v$ ← S.pop()
        if $v$.visited is false then
            $v$.visited ← true
            relationshipGroups ← $v$.outgoingEdges.groupBy(relationship)
            for all group ∈ relationshipGroups do       // transform pairs of the same
relationship types
                Triples.add(MessageTriple(group.sourceNodes,group.relationship,v))
            end for
            for all $(v, u) \in E_{inv}$ do
                S.push($u$)
            end for
        end if
    end while
    return Triples
end procedure
```

A precondition of the algorithm is that there exists a path from every node $v \in V$ to the root node. For a TOSCA topology, this is usually only the case for a single-tier application topology with exactly one root node. In order to allow applications with multiple root nodes, which corresponds to multiple tiers in application topologies, the tiers have to be split beforehand.

This is accomplished by assuming that there exists a unique relationship in the topology graph, which is used to denote the connection-relationship of one tier to another, and removing this relationship would mean that the two tiers are no longer weakly connected. In practice, this is a *connectsTo*-relationship or a derivation of the *connectsTo* base type. By temporarily removing all *connectsTo*-relationships and storing them elsewhere, we can determine every single tier and its root as the input for the algorithm. The procedure will in consequence not capture the relationship between tiers as a Message Triple. The stored relationships can then be transformed to Message Triples and added to their respective Message Triple list that is the output for a single tier.

The conceptual illustration of a topology with two tiers is depicted in Figure 4.3. The two tiers operate separately, but maintain a connection at some point between two components through a *connectsTo* relationship.

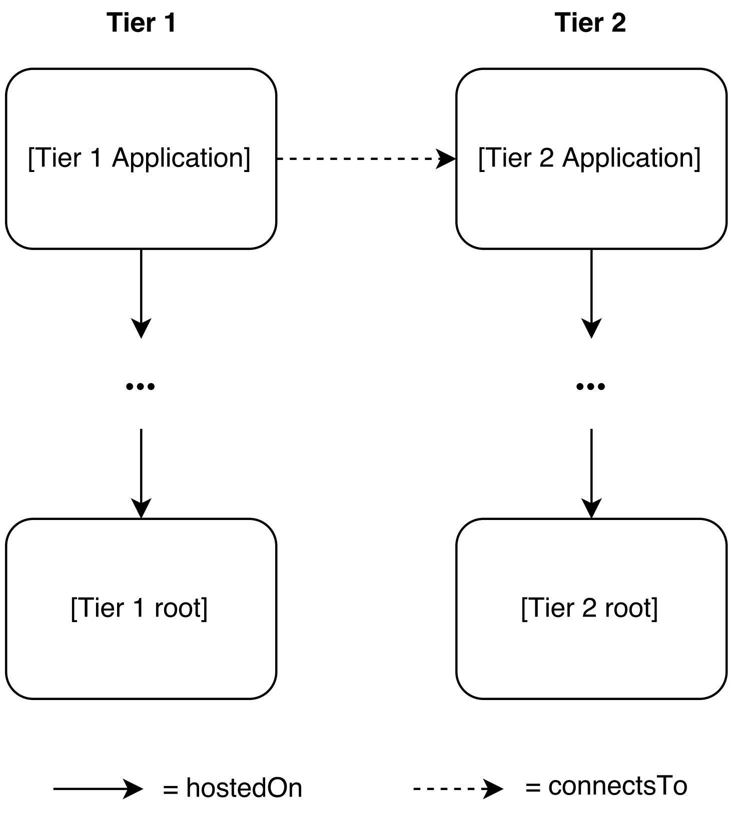

Figure 4.3: Topology with two tiers

In this example, the algorithm runs on every tier separately. Then, the Message Triples of **Tier 2** are extended with a *connectsTo* Message Triple, which connects the two tiers.

Sentence Aggregation

Aggregation it part of the Microplanning task. As information is expressed in text, redundancy can occur. The task of aggregation is to remove redundant information and to produce more coherent and fluent output [Dal99]. In Section 4.3, aggregation is already performed to a certain degree during the execution of the algorithm for Document Structuring. In this section, we propose an additional rule for sentence aggregation for the system.

The two sentences:

The Prediction Service and the Delivery Service depend on the Python Interpreter.

The Prediction Service and the Delivery Service are hosted on a Flink instance.

A possible aggregating transformation that attempts to remove duplicate information and merges the two sentences into one:

The Prediction Service and the Delivery Service depend on the Python Interpreter and are hosted on a Flink instance.

We can formalize this rule for a nonempty sets of entities `A`, `B` and `C`, and relationships rel_1 and rel_2:

$$\begin{aligned}&< A, \mathsf{rel}_1, B >\\ &< A, \mathsf{rel}_2, C >\\ &\rightarrow [< A, \mathsf{rel}_1, B >, < A, \mathsf{rel}_2, C >]\end{aligned}$$

Message Triples of this form are collected and included in the same template in the system.

Listing 4.1 Example of a *NodeTemplate* Definition with additional *Properties*

```
<tosca:NodeTemplate xmlns:ns119="http://opentosca.org/nodetypes" name="VSphere_5.5"
    id="VSphere_5_5" type="ns119:VSphere_5.5">
<tosca:Properties>
      <ns119:CloudProviderProperties xmlns="http://opentosca.org/nodetypes"
          xmlns:ty="http://opentosca.org/nodetypes">
            <ty:HypervisorEndpoint>endpointurl</ty:HypervisorEndpoint>
            <ty:HypervisorTenantID>smartservices</ty:HypervisorTenantID>
            <ty:HypervisorUserName>user</ty:HypervisorUserName>
            <ty:HypervisorUserPassword>pass</ty:HypervisorUserPassword>
      </ns119:CloudProviderProperties>
</tosca:Properties>
</tosca:NodeTemplate>
```

Listing 4.2 Example of a *RelationshipTemplate* Definition

```
<tosca:RelationshipTemplate
    xmlns:tbt="http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes" name="con_13"
    id="con_13" type="tbt:HostedOn">
<tosca:SourceElement ref="Ubuntu-14_04-VM"/>
<tosca:TargetElement ref="VSphere_5_5"/>
</tosca:RelationshipTemplate>
```

4.4 Lexicalization

The task is Lexicalization is to determine the required lexical and syntactical means to express the generated Message Triples. This includes the determination of descriptors that individual entities will later be referred to as in the output text. The subtasks of Lexicalization of this system are illustrated in Figure 4.4. Incoming Message Triples are temporarily split into their individual entities, for which a describing name is determined. This is elaborated in detail in Section 5.2.1. Then, appropriate templates for the Message Triples are determined. These processing steps are supported with the incorporation of domain-specific knowledge. This knowledge e.g. aids in the determination of an entity type from a taxonomy. Furthermore, it facilitates the initial process of creating templates. These templates are then prepared to form a sentence, but is still considered an internal representation, as following steps might further transform these sentence templates.

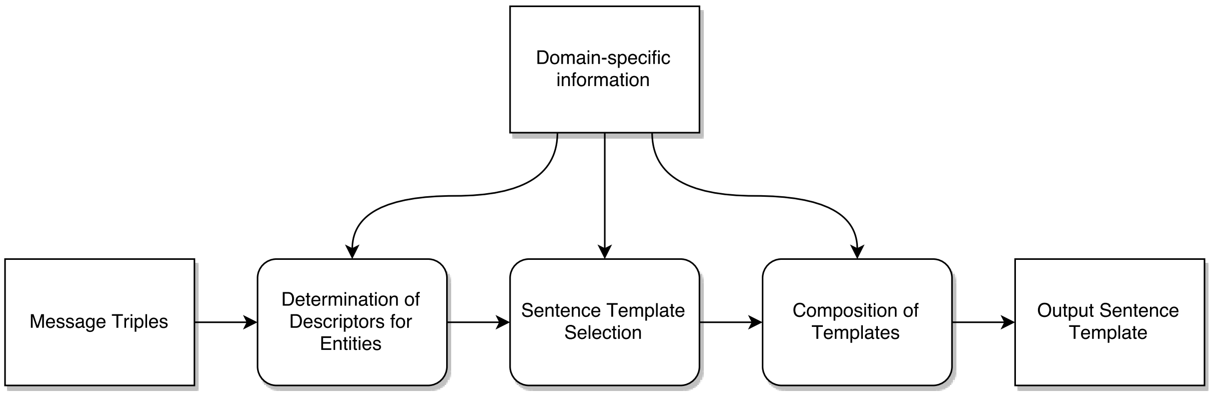
Domain-specific information
Message Triples
Determination of Descriptors for Entities
Sentence Template Selection
Composition of Templates
Output Sentence Template

Figure 4.4: Subtasks of the Lexicalization task

4.4.1 Sentence Templates

In language independent NLG systems, very refined language models are used, which can get highly complex. They are both difficult to create and maintain. Furthermore, it is argued that the sophistication of such models does not correlate with sophistication of language output in the majority of cases when domain-specific information is to be verbalized. Oftentimes, similarities and patterns are identified in the language used to describe domain-specific topics. These can be extracted and stored separately to reduce the complexity of the system. These fixed parts serve as templates, as they can be reused.

4.4.2 Descriptors for Entities

Naming of Nodes and Relationships is up to the architect designing the TOSCA ServiceTemplate. To further categorize nodes such as *Ubuntu VM*, a taxonomy is needed. Some systems use a semantic net such as WordNet [Mil95], usually for general domains. In the case of the specific domain of TOSCA, a small custom taxonomy might be created for the system that categorizes entities in a two-layer hierarchical categorization.

4.5 Generating Referring Expressions

The way an entity is referred to in a text is context-sensitive and determined by the referring expression task. The task determines if an entity is referred to using an explicit descriptor, or if a suitable pronoun is used. In the proposed system, the generation of referring expressions is greedy. The entity that was referred to last is saved in a variable. When an entity is about to be mentioned, it is checked for equality to that variable. Essentially, as long as no other entity was referred to, an entity can be referred to with an appropriate referring pronoun.

4.6 Realization

The Realization task is divided into two subtasks.

The first, Linguistic Realization, processes the information and structures that were generated during the previous tasks and generates sentences from the underlying representation in the system. In the proposed system this task is closely tied to sentence templates. As discussed in [RM93], truly linguistic systems are difficult and expensive to maintain. Especially when only a subset of expressions of a language is used in the problem domain, a rather static approach using templates sentences is argued to be more suitable. In this approach, sentence templates are used to provide the Realization module.

5 Implementation

This Chapter discusses a prototypical implementation that incorporates concepts and ideas of the presented approach in Chapter 4. The prototype uses the Java programming language[1] Specific use-cases are presented in Chapter 6.

5.0.1 TOSCA's `documentation`-Tag

Throughout this work, we have not assumed the presence of the TOSCA `documentation`-Tag in any XML element. This tag is an optional field that every element in TOSCA supports by specification [OAS13]. Using this field, the author of the respective topology has the ability to provide additional documentation and information for the element. This documentation can be provided in natural language, which acts as a valuable resource for any natural language generation system. The documentation provided by the author could be incorporated into the natural language description of the topology. Furthermore, the field could be filled with additional semantic information that could otherwise not be inferred with the given information about the topology.

Listing 5.1 documentation tag content example, adapted from [IAA17]

```
<documentation>
The corresponding TOSCA topology consists of three stacks: one for the first device,
one for the second device, and one for the Message Broker.
</documentation>
```

In this implementation, the presence of a `documentation`-Tag for any element is not assumed, although they can aid in producing more qualitative content. An example is provided in Listing 5.1.

[1]https://www.java.com

5.1 Document Planning

5.1.1 Handling Input and Content Determination

The input for the prototype is a collection of `.tosca` files from CSAR archives. Using the TOSCA XSD schema, which is available at [OAS13], these files are then deserialized into Java objects using the *Java Architecture for XML Binding* (JAXB) component of OpenTOSCA. With JAXB, the elements and their respective hierarchy are mapped to Java objects according to the provided schema. Then, an internal representation of the topology is created using a graph structure. Nodes and edges are created and connected according to the *TopologyTemplate* in the *ServiceTemplate*. *NodeTemplates* and *RelationshipTemplates* are encapsuled into the respective nodes and edges. This allows access to properties of the respective *Nodes* and *Relationships* in the topology, combined with graph algorithms that can be used on the structure.

5.1.2 Document Structuring

To obtain `MessageTriples`, the proposed algorithm is then performed on the graph structure. The result is an ordered list of `MessageTriples` that can be seen as an alternative representation of the graph.

5.2 Microplanning

5.2.1 Sanitization of Names and Descriptors

Naming in the topology files usually follows naming conventions. For example, sometimes very detailed information such as the version number of a piece of software is required to be included. In a description of the topology, this might not be the level of detail that is needed, instead of *Ubuntu_14_04-VM* for example, the entity in question could also be referred to as *Ubuntu*, *Ubuntu VM*, *Linux OS* or else. This is also the case with custom tags used for elements, such as in the `CloudProviderProperties` tag in Listing 4.1. Here, camel case is used to denote the properties which otherwise might not have been referred to by their natural language name (such as “Hypervisor-Endpoint”). Camel cased words can simply be split up by the rules they are constructed, as a whitespace can be inserted before every capital letter that is not the first, then optionally modifying the case of the first letter of every word when appropriate. As a more general solution, a rule-based String sanitizer is used in this work. Individual rules are added manually. They aim to remove special characters and simplify overly technical names for the sake of readability. These rules make use of regular expressions. Expressions are matched as patterns, and then appropriately replaced with sanitized versions for later use in textual descriptions. Table 5.1 lists some expressions in Java Syntax. Regular braces (a `()`-pair) enclose entities that can in the matched replacement referred to as `$n`, where `n` denotes the `n`-th entity.

Regular Expression	Replacement
`([A-Za-z])_([0-9])`	`$1 $2`
`([0-9])(_|-)([A-Za-z])`	`$1 $3`
`([0-9])(_|-)([0-9])`	`$1.$3`
`([A-Za-z])(_|-)([0-9])`	`$1 $3`

Table 5.1: Regular expressions used to sanitize entity names using Java syntax

As for descriptors for relationships, we can observe that a small number of base types are defined as *RelationshipTypes*. Based on this observation, we infer the descriptor for a Relationship by following the inheritance path to the base type and use an appropriate verbalization of it.

5.3 Realization

5.3.1 Sentence Templates

The system uses sentence templates to perform the Linguistic Realization task. A Template Engine is responsible for mapping Message Triples to sentences using appropriate templates. The templates are strings with special placeholders, similar to those found in other templating engines such as Jinja[2]. Since every Message Triple contains exactly one target-node, because of how the algorithm in Section 4.3 operates, it is referred to in the templates as

```
{{target}}
```

The source-nodes $(v_1, \ldots, v_n)$ are referred to with

```
{{v_1}}, ..., {{v_n}}.
```

[2]http://jinja.pocoo.org

Relationships $(\text{rel}_1, \ldots, \text{rel}_m)$ are represented as

```
{{rel_1}}, ..., {{rel_m}}.
```

Every Message Triple is realized using one sentence template. For every placeholder, the appropriate referring expression is filled.

5.3.2 Structure Realization

With linguistically realized sentences, the task of producing the output text remains. The prototype currently supports only basic export functionality as an unformatted text. Generally speaking, this component can be designed in a modular way, so that different export formats, such as HTML or LaTex source code for instance, are possible outputs. Depending on the use-case, the export could be directly integrated into external software through an interface, and further processed outside of the system.

6 Validation

In Chapter 4 we introduced an approach to generate textual descriptions of TOSCA topologies. Chapter 5 provides information on a prototypical implementation of the approach. In this Chapter, feasibility of the presented approach is validated using the prototype by running it on an exemplary topology. Individual tasks are revisited in the context of the use-case and described.

6.1 Use-Case: IoT Scenario

The following topology is adapted from the OpenTOSCA website[1]. The application topology is depicted in Figure 6.1. It uses different technologies to realize the following scenario:

Two Raspberry Pi devices control a temperature sensor and a ventilator respectively (both not part of the application topology). One device acts as a Publisher of information and obtains data from the temperature sensor, which is then sent to an Eclipse Mosquitto[2] Topic via MQTT[3]. The Mosquitto instance acts as a Broker in the MQTT network and can send a signal to the second Raspberry Pi device, which is subscribed to the Topic. If the data received by the second device indicates a temperature above a certain threshold, the device activates the ventilator connected to it. Both devices interface with their connected sensors and the MQTT Topic using Python[4] scripts.

6.1.1 Document Planning

As described in the approach, the content determination includes all nodes and relationships found in the topology. The topology graph consists of multiple tiers that

[1]http://www.opentosca.org/sites/examples.html
[2]https://mosquitto.org
[3]http://mqtt.org
[4]https://www.python.org

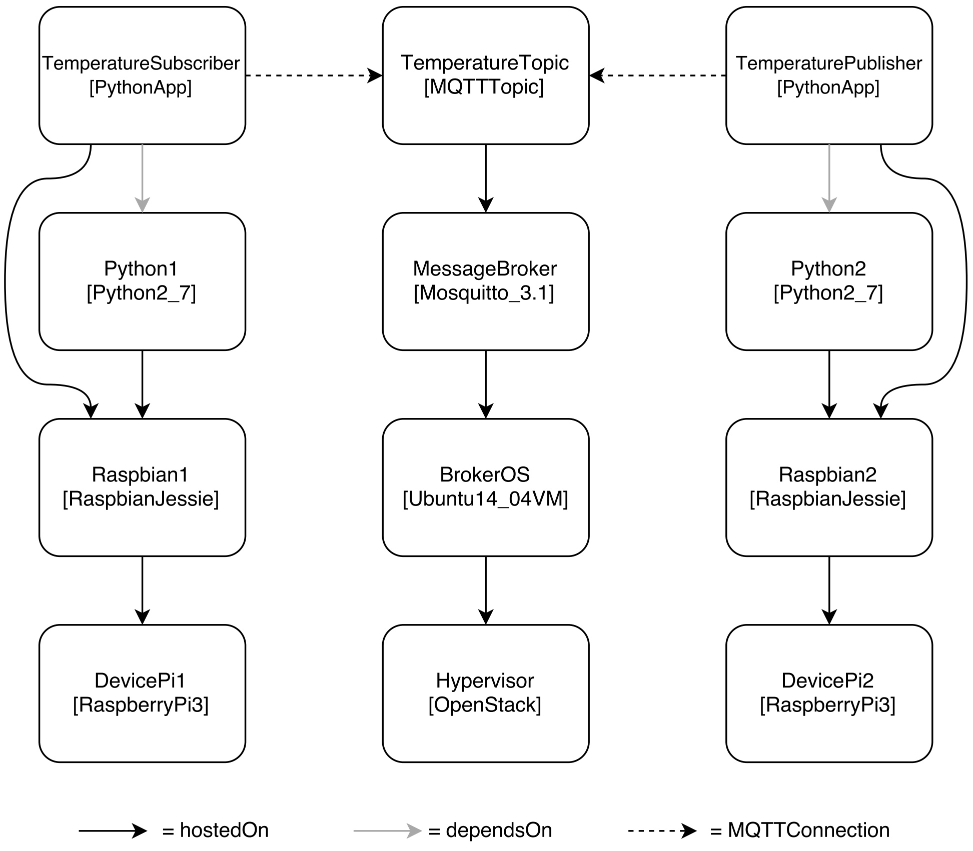
TemperatureSubscriber [PythonApp]
TemperatureTopic [MQTTTopic]
TemperaturePublisher [PythonApp]
Python1 [Python2_7]
MessageBroker [Mosquitto_3.1]
Python2 [Python2_7]
Raspbian1 [RaspbianJessie]
BrokerOS [Ubuntu14_04VM]
Raspbian2 [RaspbianJessie]
DevicePi1 [RaspberryPi3]
Hypervisor [OpenStack]
DevicePi2 [RaspberryPi3]
= hostedOn
= dependsOn
= MQTTConnection

Figure 6.1: TOSCA Topology representing an IoT scenario including two Raspberry Pi devices and a MQTT Broker instance

communicate with each other through a `MQTTConnection`-edge. As the document structure algorithm can only handle stacks individually, these connection relationships are temporarily not considered in the construction of Message Triples to obtain three distinct connected components in the topology. This information is recognized by the system as it detects three root-nodes. Every stack is then processed by the algorithm, yielding the lists of Message Triples in Listing 6.1:

Subsequently, the `MQTTConnection`-edges are transformed into additional Message Triples:

These two additional Message Triples are then appended to the list of Message Triples for the Broker-Tier, as it represents the target of both connection-relationships (Listing 6.2).

Listing 6.1 Message Triples created by the algorithm

```
Tier 1:
MessageTriple[source=[Raspbian1], target=[DevicePi1], type=hostedOn]
MessageTriple[source=[Python1, TemperatureSubscriber], target=[Raspbian1], type=hostedOn]
MessageTriple[source=[TemperatureSubscriber], target=[Python1], type=dependsOn]

Tier 2:
MessageTriple[source=[BrokerOS], target=[Hypervisor], type=hostedOn]
MessageTriple[source=[MessageBroker], target=[BrokerOS], type=hostedOn]
MessageTriple[source=[TemperatureTopic], target=[MessageBroker], type=dependsOn]

Tier 3:
MessageTriple[source=[Raspbian2], target=[DevicePi2], type=hostedOn]
MessageTriple[source=[TemperaturePublisher, Python2], target=[Raspbian2], type=hostedOn]
MessageTriple[source=[TemperaturePublisher], target=[Python2], type=dependsOn]
```

Listing 6.2 Additional Message Triples for connections

```
MessageTriple[source=[TemperatureSubscriber], target=[TemperatureTopic],
    type=MQTTConnection]
MessageTriple[source=[TemperaturePublisher], target=[TemperatureTopic],
    type=MQTTConnection]
```

6.1.2 Microplanning

In Microplanning, all nodes and relationships are processed to identify an appropriate and readable representation in the output document. The natural language representation of relationships are derived from their base type. Since *hostedOn* and *dependsOn* are considered base types, only the descriptor *MQTTConnection* has to be changed. One alternative is to express this relationship explicitly as a “*MQTTConnection relationship*”. The approach of the prototype is to use the descriptor of the relationships base type, in this case a *connectsTo* relationship.

6.1.3 Realized Output

After Document Planning and Microplanning, an internal representation of the topology has been assembled. The Realizer task takes this representation as input and produces the output. Appropriate sentence templates for Message Triples are selected and filled. In this case, sentence templates with the ability to express additional information for nodes has been selected. If a node is referred to as a single entity and is mentioned as such for the first time in the text, it is introduced to the reader by explicitly stating its type in a subordinate clause.

The generated output of the prototype is listed in Listing 6.3.

Listing 6.3 Output text for the example topology

```
The topology consists of a total of 12 entities and 13 relationships.
It is split into three stacks.

Raspbian1, which acts as an Operating System, is hosted on DevicePi1, which acts as an
    IoT device.
Python1 and TemperatureSubscriber are hosted on Raspbian1.
TemperatureSubscriber, which acts as a MQTT Subscriber instance, depends on Python1,
    which acts as an Interpreter.
BrokerOS, which acts as an Operating System, is hosted on a Hypervisor.
MessageBroker, which acts as a MQTT Broker instance, is hosted on BrokerOS.
TemperatureTopic, which acts as a MQTT Topic, depends on MessageBroker.
Raspbian2, which acts as an Operating System, is hosted on DevicePi2, which acts as an
    IoT device.
TemperaturePublisher and Python2 are hosted on Raspbian2.
TemperaturePublisher, which acts as a MQTT Publisher instance, depends on Python2, which
    acts as an Interpreter.
```

7 Conclusion and Future Work

This Chapter provides a short summary of this work. In addition, future work on the system is proposed and use-cases are discussed.

Conclusion

In this work, an approach for the automated generation of natural language descriptions of TOSCA topology graphs has been proposed. To accomplish this, every typical task in Natural Language Generation is addressed with a solution that is tailored to the problem domain of application topologies.

First, foundations of TOSCA and Natural Language generation were presented. This provided a base for examining related work.

With the aid of observations made in foundational as well as related work, an approach for transforming TOSCA topologies to natural language texts was presented. First, TOSCA topologies are input and its files are parsed and processed. To determine a linear order of entities to structure the text, an algorithm was presented. Through application of aggregation techniques, more fluent and readable sentences are formed. The sentences are then surfaced from underlying representations by a Realizer component. The output of the system is a textual representation of the topology graph.

This concept was implemented in a prototype and validated with an exemplary topology graph.

Future Work

This Section discusses Future Work based on this work.

The presented approach does currently not include processing techniques to express individual components of graphs in detail, such as Properties and Interfaces of Nodes. For this, the entire pipeline has to be extended, ranging from Content Determination, over additional sentence structures, to Structure Realization in the output document. More granular descriptions are therefore a task for future work.

In addition, the prototype only provides simple text export functionality of the output, which could be extended to support different document formats or an integration with visual modeling tools for topologies such as Winery [KBBL13] to combine visual and textual information in documents.

The generated textual representations can also be used as a starting point to assess the possibilities of reconstructing topology graphs from textual or other unstructured representations of applications. This can be motivated by the sheer amount of online software repositories on platforms such as GitHub[1]. Oftentimes, these repositories include descriptions and instructions in natural language. Recent work on the automated generation of TOSCA topologies from public repositories by Endres et al. [EBLW] has shown that applications can be automatically "ported" to topologies graphs. The quality of the results could possibly be improved by including information from the textual documentation available. For this purpose, named entity recognition and part-of-speech tagging [MS+99] could be inflicted on the textual descriptions.

[1]https://octoverse.github.com

Bibliography

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. “A view of cloud computing.” In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on p. 15).

[ALG13] I. Androutsopoulos, G. Lampouras, D. Galanis. “Generating natural language descriptions from OWL ontologies: the NaturalOWL system.” In: *Journal of Artificial Intelligence Research* 48 (2013), pp. 671–715 (cit. on pp. 27, 31).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. “OpenTOSCA–a runtime for TOSCA-based cloud applications.” In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 692–695 (cit. on p. 15).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “TOSCA: Portable Automated Deployment and Management of Cloud Applications.” Englisch. In: Advanced Web Services. New York: Springer, Jan. 2014, pp. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2014-01&engl=0 (cit. on p. 17).

[Dal99] H. Dalianis. “Aggregation in natural language generation.” In: *Computational Intelligence* 15.4 (1999), pp. 384–414 (cit. on pp. 22, 34).

[DR95] R. Dale, E. Reiter. “Computational interpretations of the Gricean maxims in the generation of referring expressions.” In: *Cognitive science* 19.2 (1995), pp. 233–263 (cit. on p. 22).

[EBLW] C. Endres, U. Breitenbücher, F. Leymann, J. Wettinger. “Anything to Topology - A Method and System Architecture to Topologize Technology-Specific Application Deployment Artifacts.” In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017), Porto, Portugal* (cit. on p. 48).

[GA07] D. Galanis, I. Androutsopoulos. “Generating multilingual descriptions from linguistically annotated OWL ontologies: the NaturalOWL system.” In: *Proceedings of the Eleventh European Workshop on Natural Language Generation*. Association for Computational Linguistics. 2007, pp. 143–146 (cit. on p. 27).

[IAA17] IAAS. *OpenTOSCA*. 2017. URL: http://www.opentosca.org/ (cit. on pp. 19, 39).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery–a modeling tool for TOSCA-based cloud applications.” In: *International Conference on Service-Oriented Computing*. Springer. 2013, pp. 700–704 (cit. on p. 48).

[LF09] F. Leymann, D. Fritsch. “Cloud Computing: The Next Revolution in IT.” In: *Proceedings of the 52th Photogrammetric Week* (2009), pp. 3–12 (cit. on p. 15).

[LMP12] H. Leopold, J. Mendling, A. Polyvyanyy. “Generating natural language texts from business process models.” In: *International Conference on Advanced Information Systems Engineering*. Springer. 2012, pp. 64–79 (cit. on p. 25).

[LRR96] B. Lavoie, O. Rambow, E. Reiter. “The Modelexplainer.” In: *Proceedings of the 8th international workshop on natural language generation*. 1996, pp. 9–12 (cit. on p. 26).

[MAA08] F. Meziane, N. Athanasakis, S. Ananiadou. “Generating Natural Language specifications from UML class diagrams.” In: *Requirements Engineering* 13.1 (2008), pp. 1–18 (cit. on p. 26).

[MG+11] P. Mell, T. Grance, et al. “The NIST Definition of Cloud Computing.” In: (2011) (cit. on p. 15).

[Mil95] G. A. Miller. “WordNet: a lexical database for English.” In: *Communications of the ACM* 38.11 (1995), pp. 39–41 (cit. on pp. 26, 37).

[MM14] P. W. McBurney, C. McMillan. “Automatic documentation generation via source code summarization of method context.” In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 279–290 (cit. on p. 27).

[MS+99] C. D. Manning, H. Schütze, et al. *Foundations of Statistical Natural Language Processing*. Vol. 999. MIT Press, 1999 (cit. on p. 48).

[OAS13] OASIS-Standard. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 17, 18, 29, 30, 39, 40).

[RD97] E. Reiter, R. Dale. “Building applied natural language generation systems.” In: *Natural Language Engineering* 3.1 (1997), pp. 57–87 (cit. on p. 19).

[RDF00] E. Reiter, R. Dale, Z. Feng. *Building Natural Language Generation Systems*. Vol. 33. MIT Press, 2000 (cit. on pp. 19, 20, 25, 27, 31).

[Rei95] E. Reiter. “NLG vs. templates.” In: *arXiv preprint cmp-lg/9504013* (1995) (cit. on p. 23).

[RM93] E. Reiter, C. Mellish. “Optimizing the costs and benefits of natural language generation.” In: *IJCAI*. 1993, pp. 1164–1171 (cit. on pp. 23, 37).

[Ste02] H. Stenzhorn. “XtraGen: a natural language generation system using XML- and Java-technologies.” In: *Proceedings of the 2nd workshop on NLP and XML-Volume 17*. Association for Computational Linguistics. 2002, pp. 1–8 (cit. on p. 25).

[Tar72] R. Tarjan. “Depth-first search and linear graph algorithms.” In: *SIAM journal on computing* 1.2 (1972), pp. 146–160 (cit. on p. 32).

[VTK05] K. Van Deemter, M. Theune, E. Krahmer. “Real versus template-based natural language generation: A false opposition?” In: *Computational Linguistics* 31.1 (2005), pp. 15–24 (cit. on p. 23).

All links were last followed on September 28, 2017.

Declaration