VeriFog: A Generic Model-based Approach for Verifying Fog Systems at Design Time and Generating Deployment Configurations

Hiba Awad, Abdelghani Alidra, Hugo Bruneliere, Thomas Ledoux, Jonathan Rivalan

VeriFog: A Generic Model-based Approach for Verifying Fog Systems at Design Time and Generating Deployment Configurations

Hiba Awad
IMT Atlantique, LS2N (UMR CNRS 6004), Inria Rennes, Smile
Asnières-sur-Seine and Nantes
France
hiba.awad@smile.fr

Abdelghani Alidra
IMT Atlantique, LS2N (UMR CNRS 6004), Inria Rennes
Nantes, France
abdelghani.alidra@imt-atlantique.fr

Hugo Bruneliere
IMT Atlantique, LS2N (UMR CNRS 6004)
Nantes, France
hugo.bruneliere@imt-atlantique.fr

Thomas Ledoux
IMT Atlantique, LS2N (UMR CNRS 6004), Inria Rennes
Nantes, France
thomas.ledoux@imt-atlantique.fr

Jonathan Rivalan
Smile
Asnières-sur-Seine, France
jonathan.rivalan@smile.fr

ABSTRACT

Fog Computing is a paradigm decentralizing the Cloud by geographically distributing computation, storage, network resources and related services. It provides benefits such as reducing the number of bottlenecks, limiting unwanted data movements, etc. However, managing the size, complexity and heterogeneity of the Fog systems to be engineered is challenging and can quickly become costly. According to best practices in software engineering, verification tasks could be performed on a system design prior to its implementation and deployment. We propose a generic model-based approach for verifying Fog systems at design time, also enabling the automatic generation of corresponding deployment configuration files. Named VeriFog, this approach is notably based on a customizable Fog Modeling Language (FML). We experimented in practice by modeling three use cases, from three different application domains, and by considering three main types of non-functional properties to be verified. From this modeling and verification effort, we show that we are able to automatically generate usable deployment configuration files for different deployment tools. In direct collaboration with our industrial partner Smile, the approach and underlying language presented in this paper are necessary steps towards a more global model-based support for the complete life cycle of Fog systems.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; **Software verification and validation**; • **Computer systems organization** → **Cloud computing**;

KEYWORDS

Model-Based Engineering, Modeling Language, Fog Computing, Verification, Non-Functional Properties, Design Time, Generation, Deployment Configuration.

1 INTRODUCTION

Fog Computing [35, 51] is a recent paradigm aiming to decentralize the Cloud by geographically distributing away computation, storage and network resources as well as related services. Instead of a centralized Cloud system, data centers of various sizes in the core network, and smaller data-centers or devices at the edge of the network, can be used collaboratively to form a single large-scale geo-distributed system. Thus, Fog Computing is at the crossroads of complementary areas of distributed systems: Cloud Computing [6] of course, but also Edge Computing [41] and IoT [28].

Over the last years, Fog systems have been extensively studied in the research community, though their actual dissemination in industry remains relatively limited [18]. The industrial report, corroborated by our partner Smile, provides several reasons to explain this situation. A first one is the size, complexity and heterogeneity of the systems to be designed, developed, tested, deployed and then maintained in such a Fog Computing context. Indeed, a Fog system can combine large-scale Cloud resources with a possibly huge number of IoT devices at the Edge of the network. All these varied Fog resources have to be organized into several, sometimes many, inter-connected areas. As a result, such a Fog system can quickly become very challenging and costly to manage efficiently [3]. An important goal is notably to ensure an appropriate Quality of Service (QoS) [44]. This may concern security [32], energy [30], or resource usages [1] aspects (among others).

Best practices in software engineering already showed that a common way to limit development and management costs is to verify the system design prior to its actual implementation and deployment. However, up to our current knowledge, the verification approaches that have been applied so far in the context of Fog systems mostly concern already implemented and/or deployed systems [24]. Moreover, by looking at a detailed study of the state of the art [4], we observed that system modeling is currently rather limited in Fog Computing and mostly concerns the simulation of only parts of the Fog systems. Thus, to go a step further, we introduce the VeriFog model-based approach for the verification of

Fog systems at design time and the automated generation of corresponding deployment configuration files. To this end, we notably propose a generic and customizable Fog Modeling Language (FML) to specify Fog system models that can then be navigated, queried and transformed accordingly. To experiment with our approach and language in practice, we tested them for three main types of important non-functional properties (security, energy and performance) that we verified in the context of three different Fog systems we modeled. In addition, we automatically generated initial deployment configuration files, for different deployment tools, from the obtained Fog system models. These Fog systems come from three different use cases illustrating various application domains of Fog Computing. The objective is to demonstrate the practical relevance of VeriFog for verifying several non-functional properties of Fog systems during their design phase and then accelerating their deployment. We also want to show the actual usability of FML for modeling different kinds of Fog systems. On the longer term, both VeriFog and FML belong to a wider research initiative intending to capitalize on Fog system models in order to better support the Fog system's overall life cycle.

This paper is organized as follows. Section 2 presents the general background and motivation for our work, resulting in the four research questions we intend to address in this paper. Based on that, Section 3 introduces the overall VeriFog approach and its targeted verification support. Section 4 describes the FML language as a core component of VeriFog. Section 5 presents the additional generation support we provide for deployment configuration files. Then, Section 6 illustrates the practical experiments with our verification and generation approach, and associated language, in the context of three different use cases. Section 7 summarizes the current Eclipse/EMF-based implementation of VeriFog, FML, and the studied use cases. Section 8 discusses the current status, limitation, and lessons learned from the work. Finally, Section 9 explains the related work while Section 10 concludes the paper by opening on future research perspectives.

2 BACKGROUND AND MOTIVATION

As introduced earlier, managing the size, complexity and heterogeneity of Fog systems can rapidly become both challenging and expensive. Providing an answer towards the resolution of this kind of problem is a main objective of the ongoing SeMaFoR project [3], for example. In the context of this collaborative research effort involving both research and industrial partners, we aim at proposing an approach for supporting the self-management of Fog resources. More particularly, we are currently designing and developing a generic, end-to-end, decentralized and collaborative self-management solution to be able to operate different kinds of Fog systems. To this end, we are notably tackling both Fog modeling concerns and their potential impact on various aspects of the Fog system's life cycle. The work presented in this paper is positioned at the intersection of these two dimensions.

On the one hand, Fog Computing remains a relatively recent paradigm. Thus, many aspects related to Fog systems and the general support for their engineering and management are open challenges [18]. In the scientific literature, these aspects have been studied from a theoretical and/or research perspective over the past years. Still, for various reasons, it appears that they have yet to be experimented in practice within the industrial context. Among the different phases of the Fog system's life cycle which are worth exploring, the design phase is particularly important. Notably, any verification to be made onto a given system before its implementation and deployment can be highly beneficial in the long run (e.g. to improve the overall QoS of the system [25, 44]). This has already been observed in the context of other kinds of distributed systems [21, 31] and still needs to be adapted for modern Fog systems. However, existing verification approaches have been mostly applied on (Fog) systems already implemented and/or deployed [24, 37].

On the other hand, the approach resulting from a project such as SeMaFoR is meant to rely on a language allowing to model different Fog systems, coming from different application domains and for possibly covering different purposes. A deep study of the state of the art in terms of existing modeling languages and related capabilities in a Fog Computing context is already available [4]. From this extensive study, we indicated that the already existing modeling support dedicated to Fog systems, their building and their management, is rather limited at this stage. As a consequence, more efforts are needed in order to complement the available (mostly simulation-based) solutions with more elaborated model-based solutions intended to Fog systems in particular. Notably, generative solutions dedicated to Fog systems and targeting different kinds of related development artifacts (e.g. source code, configuration or build files) are globally missing.

Based on this analysis, we considered the following four Research Questions (RQs) in the next sections of the paper:

- RQ1. Can we build a (model-based) approach for verifying Fog systems before their implementation or deployment?
- RQ2. In such an approach, which language do we need to support the modeling of Fog systems?
- RQ3. Once verified, can the resulting Fog system models be also used for automatically generating an initial deployment configuration file?
- RQ4. Can such an approach, language, and models be used for verifying different non-functional properties on different types of Fog systems within different application domains, and for generating actual configuration files?

3 AN APPROACH FOR DESIGN-TIME VERIFICATION OF FOG SYSTEMS (RQ1)

Figure 1 presents an overview of the VeriFog iterative approach.

(1) The Fog System Architect (FSA) is the main actor. Ideally, she/he is an engineer having the required knowledge and experience in terms of both Fog infrastructures and the targeted application domain(s) (e.g. smart cities, industrial IoT, autonomous vehicles). In practice, it can rather be a collaboration between several Fog system's engineers (e.g. experts on distributed systems) and domain experts (e.g. persons in charge of the Fog system).

(2) Using an appropriate Fog Modeling Language, the FSA starts by modeling the topology of the Fog system(s) she wanted to design via one or several models/files. This way, the FSA identifies and represents the various types of Fog resources that will compose the target Fog system, as well as the various types of possible interconnections between these types of resources. In some cases,

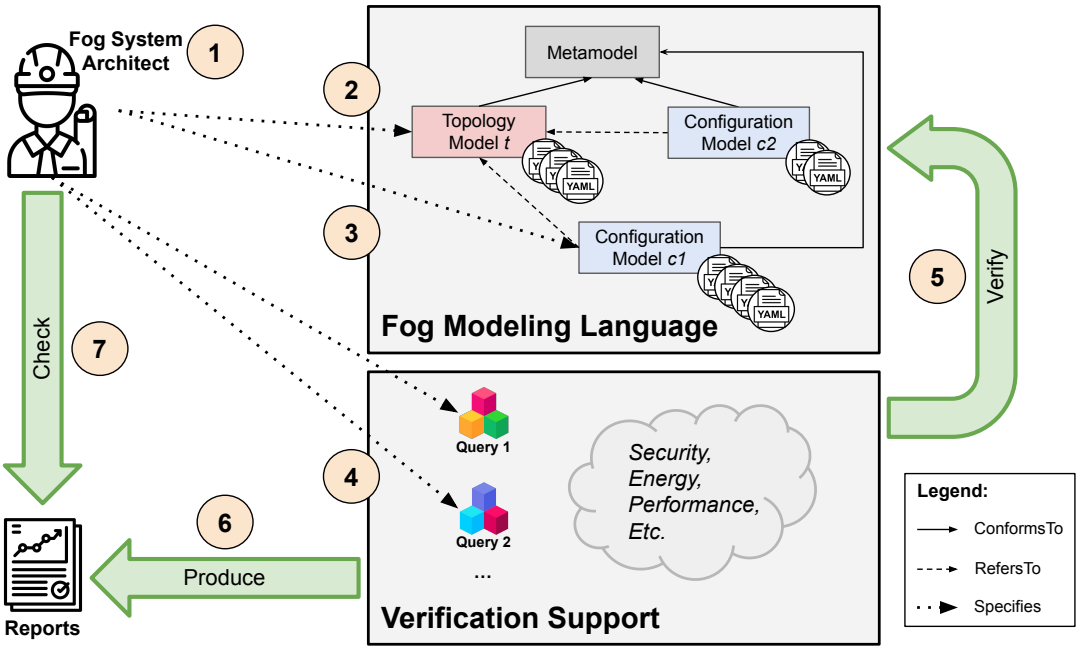

Figure 1: Overview of the VeriFog iterative approach (the circled numbers correspond to numbered paragraphs in the text).

she proposes one single topology that can then be instantiated in different configurations of the target Fog system. In other cases, several topologies can be proposed at this step to initially study the pros and cons of each one of them (before moving to the next step).

③ Based on the previously defined topology, or the selected one if several have been defined in the previous step, the FSA can specify different Fog system's configurations via several models/files. This way, different possible states of the target Fog systems can be represented at various given points in time. These configurations directly refer to the corresponding topology: they contain concrete instantiations of the various types of Fog resources described in the topology, as well as their actual interconnections.

④ Once the required topology and configuration models are properly specified, that all conform to the metamodel of the used Fog Modeling Language, the FSA can identify and express the non-functional properties she/he wants to verify on the target Fog system. As later displayed in Table 2, each property to be verified can be associated to different queries in order to be able to provide a global assessment over the target Fog system. In practice, these queries have to be expressed using a (model) query language (in our current experiments, we used OCL[1]) that is compatible with the Fog Modeling Language we propose (cf. Section 7).

⑤ Once the required queries are properly specified, they can be actually verified onto the set of Fog system's models. Depending on the kind of verification to be made on the Fog system, a given query can be executed over a topology model, a configuration model, or eventually a combination of them. Depending on the type of a given query, the result returned by the execution of this query can take different types: a numerical value, a boolean value or a string textually describing the result. Various examples of non-functional properties and associated queries are presented in Section 6.

⑥ Once all the required queries are executed over the corresponding Fog system's models, a report can be produced to aggregate the individual query execution results. Such a report can take different forms: a simple list of query execution results (as in our current implementation of the approach, cf. Section 7), a more structured and elaborated document automatically generated from the Fog system's models and the results, a tool displaying in a more dynamic way the content of both these models and results, etc.

⑦ Finally, the FSA can consult and check this report. It can be used as a valuable input for taking decisions on whether to 1) proceed with the current Fog system's design and eventually start to work on its implementation, or 2) perform another iteration of the VeriFog approach to improve the system's design. In the latter case, the FSA may decide to rework the initial topology model by completing the various types of possible Fog resources. She/he may also decide to first experiment with other configurations before actually modifying the topology, etc.

4 A GENERIC LANGUAGE FOR MODELING FOG SYSTEMS (RQ2)

In the context of the SeMaFoR project [3], we provided a study of the state of the art in terms of existing languages and related support for modeling Fog systems [4]. However, none of the available solutions appears to come with a generic reusable language that would allow engineers to easily specify, share, maintain and evolve Fog system's models for different kinds of Fog architectures. As a result, and also capitalizing on other existing work on Cloud modeling [2, 12], we designed and developed a generic and customizable Fog Modeling Language (FML). In what follows, we present both its abstract

[1]https://projects.eclipse.org/projects/modeling.mdt.ocl

syntax (i.e. its core metamodel) and current concrete syntax (i.e. a textual notation). Its semantics is associated to the language usage (cf. Section 6).

4.1 Abstract Syntax - Metamodel

The main objective of the FML is to allow specifying the various resources composing the Fog system as well as the relationships between them. To this end, the proposed language is able to represent both the different possible *topologies* of Fog systems (i.e. the types of Fog resources) and the corresponding *configurations* based on these topologies (i.e. the instances of these types). Figure 2 shows a partial version of the metamodel of the language.

The main elements of a Fog system are considered as *FogResource*s. As the language is meant to be generic and extensible, any *FogResource* can be customized by adding specific metadata via both *Tag* elements (i.e. key/value pairs) and custom *Attribute* elements. Each *Attribute* is characterized by a *AttributeType* targeting a particular type of Fog resource and having a corresponding *UnitOfMeasurement*. The root of the model is the *FogSystem* containing various *FogAreas* (i.e. Fog sub-systems), themselves containing the other kinds of elements. This way, as introduced earlier, both topologies and corresponding configurations can be jointly modeled.

On the topology side, at the *FogSystem*-level, different *NodeType*s can be specified. These are either *PhysicalNodeSpecification*s (i.e. for hardware devices) or *VirtualNodeSpecifications* (i.e. for software components). Naturally, a given *NodeType* can host other *NodeType*s. For example, a virtual node can be hosted on a physical node or eventually on another virtual node. The different *NodeType*s can be interconnected via various *NetworkLinkType*s that materialize either download or upload links depending on the case. In addition, each *NetworkLinkType* is related to a given *NetworkType* that can have different characteristics associated to it (e.g. these characteristics can be expressed via particular *Tag*s or custom *Attribute*s).

On the configuration side, at the *FogArea*-level, the overall structure is similar. However, several *Node*s can be interconnected via various *NetworkLink*s (download or upload) directly related to corresponding *Network*s. Each *Node* is representing a particular instance of a previously specified *NodeType*. In addition, such a *Node* carries different general runtime properties concerning its operating system, CPU, memory, disk space, health status, etc. Similarly, each *NetworkLink* and *Network* are particular instances of a previously specified *NetworkLinkType* and *NetworkType* (respectively).

Finally, it is important to notice that some aspects of the language are not described in the paper for the sake of readability and understandability. Indeed, some simple multiplicities (e.g. 0..1 / 1..1) and associated labels are hidden in Figure 2. For the same reason, the provided support for different kinds of constraints over *NodeType* and *NetworkType* elements is not detailed neither in the paper. Note that this part of the language is directly adapted from the constraint support proposed in previous work on Cloud modeling [2, 12]. More importantly, the language also supports the modeling of both the services and related applications that can be hosted in the Fog system nodes. These are also important elements in the context of highly distributed Fog systems. However, such elements are not described here as not used in the current work where we mostly focus on modeling Fog infrastructures.

4.2 Concrete Syntax - YAML Textual Notation

The technical background of the typical FSA usually makes her quite familiar with solutions for distributed system's infrastructures. In such solutions, textual notations like YAML[2] are frequently encountered. As a consequence, we have opted for a YAML textual syntax to accompany the current version of the proposed Fog Modeling Language. Figure 3 shows an excerpt of this textual notation in the context of one of the practical use cases from Section 6.

The different *FogResource*s composing the *FogSystem* can be modeled thanks to separate YAML files (.yml). These files can be organized, stored and shared as needed by the concerned Fog engineers. For example, the left file in Figure 3 shows a *VirtualNodeSpecification* named "Fognode2", as part of a given topology. Three *Tag*s are associated with this particular *NodeType* to add specific performance-related metadata in this particular case. Then, a "spec" section allows to specify the general value of the different properties (both the base and custom ones). This section also allows to declare the corresponding *NetworkLinkType*s (download and upload ones) and related metadata, etc. Based on this topology specification, the right file in Figure 3 shows a given configuration providing an instantiation of the previously defined "Fognode2" *NodeType*, as a *Node* named "fn2". In a configuration definition, the "spec" section (from the topology specification) is replaced by a "status" section that allows to indicate the runtime value of the different properties, to declare the corresponding *NetworkLink*s, etc.

Note that, to simplify the learning and usage of the proposed notation, we voluntarily decided to have a similar syntax for describing both topology and configuration elements. Only some dedicated sections of the files are different at the two levels, e.g. "spec" vs. "status" as mentioned earlier on the provided example.

5 AN AUTOMATED GENERATION OF DEPLOYMENT CONFIGURATIONS (RQ3)

In this section, we present how the Fog system models resulting from previous verification can also be used for automatically generating initial deployment configuration files. These usable deployment configuration files can possibly target different deployment tools according to the needs.

5.1 A Generative Approach

Figure 4 presents an overview of the *Pre-Deployment Generation* component extending VeriFog.

(1) After verifying one or several Fog system models and validating their design, the FSA can select the *Fog System model* (that conforms to the *Fog System metamodel*) which she/he wants to configure. This *Fog System model* contains the complete information about both the topology and finally selected configuration. Therefore, at this stage, it is a consolidated model that is considered as ready for pre-deployment generation.

(2) The key element of the *Pre-Deployment Generation* component is the *Deployment Abstractor*. Its goal is to extract the relevant deployment information from the *Fog System model* in order to create a corresponding *Abstract deployment model*. This *Abstract deployment model* is a deployment specification (of the concerned

[2] https://yaml.org/

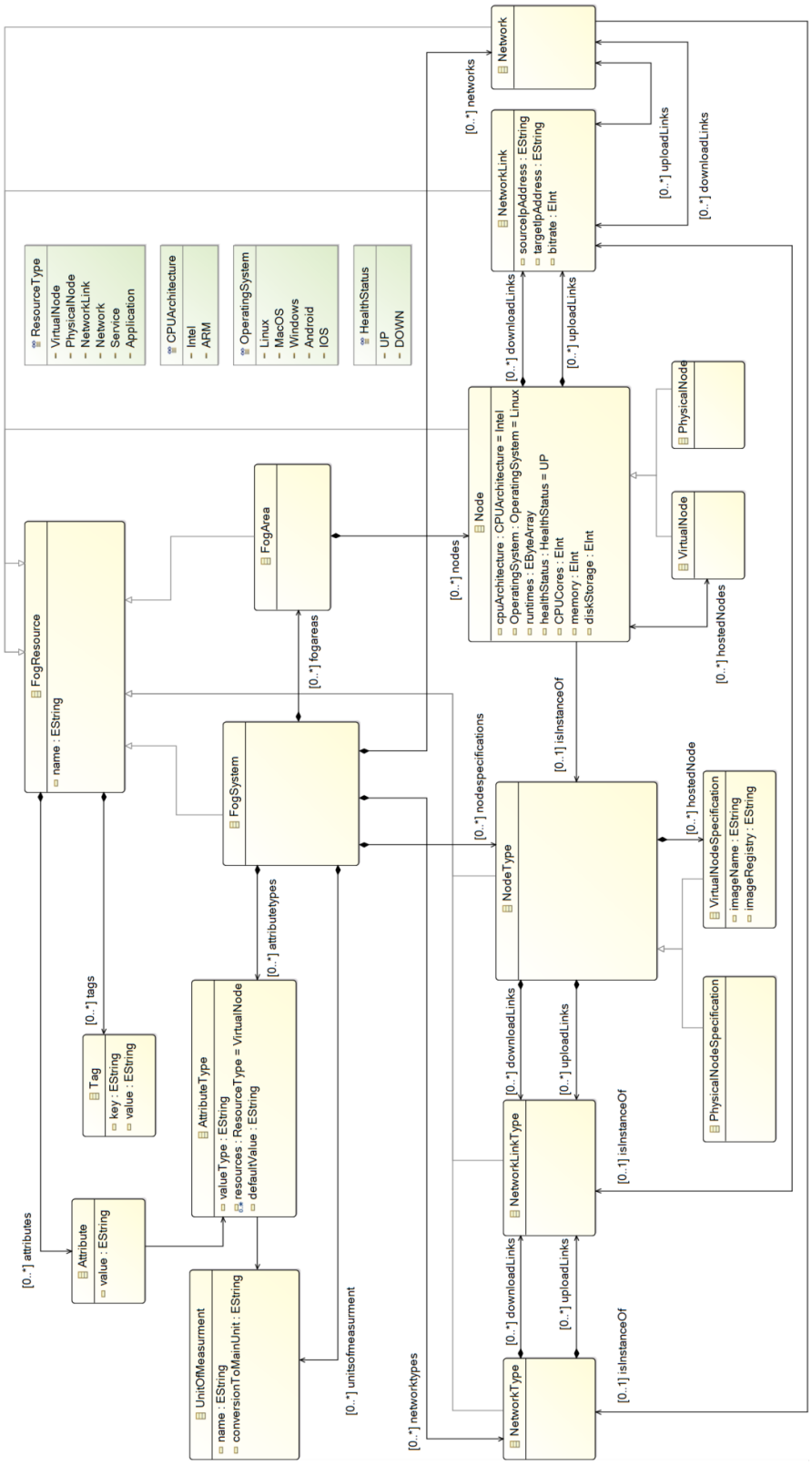

Figure 2: Partial metamodel (i.e. abstract syntax) of the proposed Fog Modeling Language.

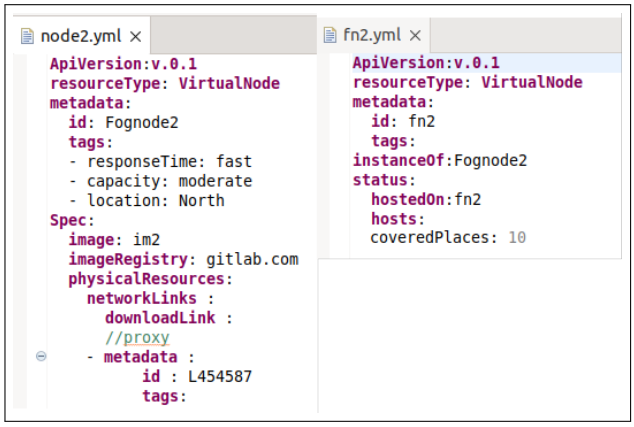

Figure 3: Excerpt of the YAML-like textual notation (i.e. concrete syntax) of the proposed Fog Modeling Language.

Fog System) that is independent from any particular *Deployment Tool*. To this end, it conforms to an *Abstract Deployment metamodel* that provides the required pivot deployment modelling language.

③ Once the *Abstract deployment model* obtained, the FSA has to select the *Deployment Tool* which she/he wants to use in order to actually deploy the specified Fog System (e.g., Terraform, Ansible, Chef). The selection can be notably influenced by technical constraints, e.g., the features provided by a given tool. It can also be guided by the QoS properties to be verified and the availability of a corresponding *Deployment Tool* in the target technological environment.

④ Finally, the *Abstract Deployment Framework* will automatically generate an initial *Deployment Configuration file*, for the selected *Deployment Tool*, from the previously obtained *Abstract deployment model*. This *Deployment Configuration file* can then be used for actually deploying the Fog System onto the selected *Deployment Tool*, once the FSA is fully satisfied with the Fog System specifications.

5.2 Support for Automation

We now focus more specifically on the additional support we provide to automate the generation of deployment configuration files within the context of the previously introduced *Pre-Deployment Generation* component.

Illustrative Example. We consider here a very simple Fog system s composed of one physical machine $pm1$ of type PM and one virtual machine $vm1$ of type VM. Both machines are interrelated as $vm1$ is hosted on $pm1$.

5.2.1 Deployment Abstractor

The main goal is to produce an abstract deployment specification from the original specification of the Fog system. To this end, a main element of the *Pre-Deployment Generation* component is the *Deployment Abstractor*. It takes as input a *Fog System model* and produces as output a corresponding *Abstract Deployment model*. Therefore, the *Deployment Abstractor* is a model-to-model transformation [34] that realizes a mapping between the concepts manipulated by the two concerned modeling languages.

As described in Section 3, we decided to rely on the Fog Modeling Language (FML) and corresponding metamodel (cf. Section 4) for specifying the *Fog System models*. For the abstract deployment language, we decided to use the Essential Deployment MetaModel (EDMM) [49] in order to specify the *Abstract Deployment models*. We believe EDMM is relevant because it directly results from a well-documented and relatively recent systematic review of existing deployment automation technologies. In EDMM, a deployment is modelled as a set of components of different types and having various properties. The deployment is composed of both physical and functional components. A component can also be a logical unit of an application. All these components can be interconnected by relations of different types (physical, functional or logical ones). The components and their associated operations (i.e., executable procedures) can be implemented by one or several artifacts (e.g., script files).

Table 1: Main rules of the proposed FML-to-EDMM mapping.

FML (*Fog System metamodel*)	EDMM (*Abstract Deployment metamodel*)
Fog system	Deployment model
Physical Node Specification (Node Type)	Component Type
Virtual Node Specification (Node Type)	Component Type
Virtual Node	Component
Physical Node	Component
Attribute	Property
Attribute Type	Property Type
Tag	Property Type
Network Link	Relation

Table 1 gives an overview of the FML-to-EDMM mapping we proposed in order to realize the *Deployment Abstractor*. The resulting *Abstract Deployment model* describes the deployment configuration independently from any specific *Deployment Tool*. The root of the FML model is a *Fog system* element that is transformed into the root of the EDMM model, i.e., a *Deployment model* element. This *Deployment model* root element will contain, directly or indirectly, all the other model elements. In FML, the main types of nodes are *Physical Node Specification* (i.e., for hardware devices) and *Virtual Node Specification* (i.e., for software components). They are both transformed into *Component Type* elements in EDMM. All the related *Attribute Type* and *Tag* elements in FML are converted to corresponding *Property Type* elements in EDMM. Similarly, the main types of nodes in FML are *Physical Node* and *Virtual Node*. They are transformed into *Component* elements in EDMM. All the related attributes are converted to corresponding *Property* elements in EDMM. The lower level details of this mapping can be seen in the implemented model-to-model transformation (cf. Section 7).

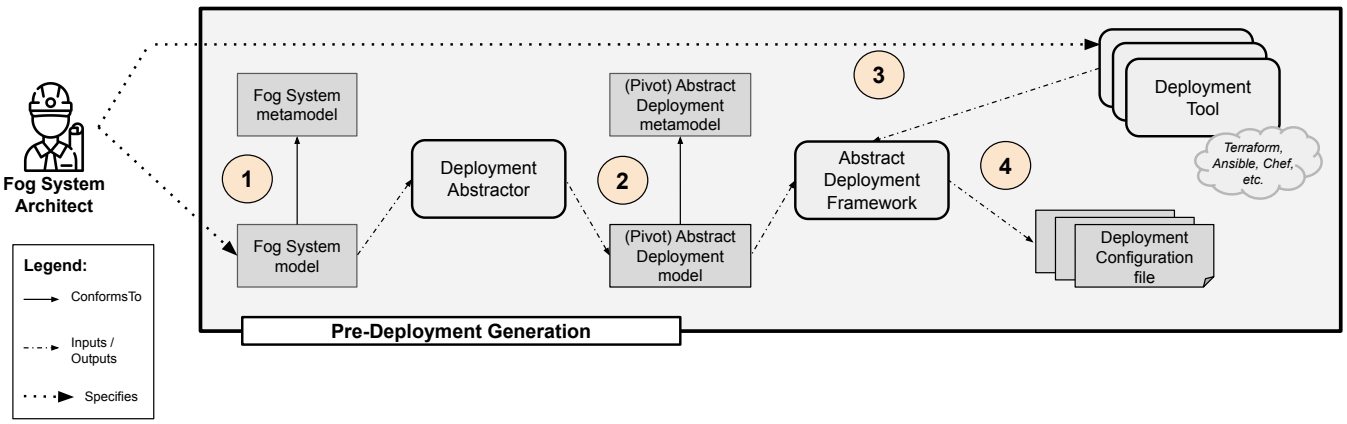

Figure 4: Overview of the Pre-Deployment Generation component extending VeriFog (the circled numbers correspond to numbered paragraphs in the text).

Illustrative Example. The *Fog System model* of s in FML is composed of a *Physical Node* $pm1$ having a *Physical Node Specification* PM and a *Virtual Node* $vm1$ having a *Virtual Node Specification* VM. The hosting relation is materialized by a *Network Link* $nl1$ between the *Physical Node* $vm1$ and the *Virtual Node* $pm1$. After transformation by the *Deployment Abstractor*, the resulting *Abstract Deployment model* of s in EDMM will be composed of a *Component* $pm1$ of *Component Type* PM and a *Component* $vm1$ of *Component Type* VM. The hosting relation is now materialized by a *Relation* $r1$ between the *Component* $vm1$ and the *Component* $pm1$.

5.2.2 Abstract Deployment Framework

Another key element of the *Pre-Deployment Generation* component is the *Abstract Deployment framework*. It takes as input an *Abstract Deployment model*, along with the selection of a specific *Deployment Tool*, and produces as output a corresponding *Deployment Configuration file*. If needed (e.g., for technical reasons), the input *Abstract Deployment model* can be first transformed into a corresponding textual file thanks to a dedicated model-to-text transformation or code generator [39].

In VeriFog, we decided to use the Deployment Model Abstraction Framework (DeMAF) [45] as our *Abstract Deployment framework*. We opted for DeMAF because it enables the generation of *Deployment Configuration files* from technology-agnostic deployment models that conforms to EDMM. Moreover, DEMAF comes with a command-line interface (CLI) that can be used directly by the FSA or integrated into an automation workflow. Importantly, the generated *Deployment Configuration files* are actually executable and suitable for different *Deployment Tools*. For example, if the FSA selects Terraform as the target *Deployment Tool*, the generated *Deployment Configuration file* will be in the '.tf' format containing the information required for a deployment in Terraform.

Illustrative Example. The *Abstract Deployment model* of s in EDMM, resulting from the previous step, can be taken as input of the *Abstract Deployment framework* DeMAF to automatically generate a corresponding *Deployment Configuration file* for the *Deployment Tool* Terraform. This file $s.tf$ contains all the information needed to actually deploy s, on Amazon Web Services (AWS) [5] for instance.

6 PRACTICAL APPLICATIONS ON DIFFERENT USE CASES (RQ4)

To evaluate the genericity, and more generally the usability, of our VeriFog approach and underlying Fog Modeling Language, we considered various Fog system's examples coming from different application domains. From the literature, we selected three distinct use cases we found relevant and representative of the heterogeneity of Fog systems. They are summarized in Table 2.

In what follows, we provide for each use case 1) a textual description of the overall context and concrete application, 2) a model of the corresponding Fog system in our Fog Modeling Language, 3) examples of queries used to verify non-functional properties which are particularly relevant in highly-distributed Fog systems and that can be verified at design-time, and 4) examples of generated abstract deployment models as well as corresponding configuration files targeting different deployment tools. For all use cases, the complete implementation resources can be accessed via the provided open source repository (cf. Section 7).

Note that, in the following sections, the presented models are voluntarily partial. Indeed, for the sake of readability, only the *NodeType* (in red) and *Node* (in blue) elements are displayed. Thus, the root elements (*FogSystem*, *FogArea*) and the network elements (*NetworkType*, *NetworkLinkType*, *Network*, *NetworkLink*) are not shown. Moreover, to keep the diagrams light, the latter are simply materialized by generic dependency relationships. Finally, the custom *Attributes* are displayed the same way than regular attributes within the corresponding elements.

6.1 Smart Campus: Energy Property

6.1.1 Scenario

The first use case is taken from a published work presenting a recent survey on existing Fog Modeling Languages [4]. In an university campus context, the survey proposes a motivating example of a Fog system that consists in two distributed applications for

Table 2: Several applications of the VeriFog approach.

Use Case Name	Verified Property	Query	Level	Return Type	Objective
Smart Campus	Energy	IsGreen	Topology	Boolean	Check if the system is green or not, i.e. if the majority of the nodes of this system have a renewable energy resource or not.
		Remaining Energy	Configuration	Double	Study the system energy status by calculating the difference between a consumption threshold and the energy consumption at time T.
		Green Energy Consumption	Configuration	Double	Calculate the percentage of green energy consumption by knowing the green level and consumed energy of each node.
Smart Parking	Performance	Efficiency	Topology	Double	Check if the system is efficient by calculating the parking lot ratio, i.e. the balance between the number of cameras and the covered slots.
		Latency	Configuration	Float	Calculate the system latency, i.e. the delay of analyzing each area of the parking lot and the delay of noise filtering in defected images.
		IsReliable	Configuration	Boolean	Study the system reliability by calculating the number of instances of each camera type with a given detection quality level.
Smart Hospital	Security	IsSecured	Topology	Boolean	Check if the system is "Secured" or not so we can know if most of the nodes in our system are located in a secure environment.
		RiskScore	Topology	Integer	Study the vulnerability of each node, based on the node location and the importance of its protected/stored data.
		Authorized Token Level	Configuration	Sequence	Group the different nodes by their respective authorization level.

smart surveillance and smart bell notification, respectively. They are made available as a set of loosely coupled micro-services that can be mutually shared in order to provide the expected capabilities.

6.1.2 Model

The Fog system of the monitored university campus is composed of several *FogAreas* corresponding to the different sections of the campus. We show in Figure 5 the content of two main *FogAreas* for the two sections of our example campus: the Main Department and the Dormitory.

The central elements are "center1" and "center2" data center *VirtualNodes*. They can be connected to other *FogAreas* of the system indirectly via the "gate" gateway *VirtualNode*, itself connected to the "publicCloudProvider" cloud *VirtualNode*. The different nodes composing the system are also connected to these two central data center nodes. In the dormitory, this is the case for the "watch" and "comp" *PhysicalNodes* (of type smart watch and computer, respectively). In the main department, this is the case for the "alarm", "mobile" and "cam" *PhysicalNodes* (of type alarm device, mobile phone and wifi camera, respectively). In this main department, the "cluster" Rapsberry Pi cluster *PhysicalNode* is indirectly connected via the "gate" gateway *VirtualNode*.

At the topology level, we added a "SourceEnergy" custom *Attribute* to model the energy source type associated to each *NodeType* instantiated in the system. Depending on the origin of the energy, it can be renewable (i.e. green), or fossile (e.g. from oil or petroleum). We also added another "greenEnergy" custom *Attribute* to be able to represent the green energy level associated to each *NodeType*. In case of multiple energy sources, this corresponds to the percentage of green energy over the total energy. For example, the level can be considered as high if the obtained percentage is higher than 50.

At the configuration level, we added a "energy_consumption" custom *Attribute* to be able to describe the amount of the total energy already consumed by each node at a given point in time. We also added another "green_consumption" custom *Attribute* to be able to specify the percentage of the total green energy already consumed by each node at a given point in time.

6.1.3 Queries

In the university campus, energy consumption is an important issue for both ecological and financial reasons. Thus, we need to assess the energy origin and consumption for the different kinds of Fog resources composing the system, as well as for the system as a whole. In what follows, we consider a college campus consisting of a main department and a dormitory.

Query "IsGreen" - To start with, we defined a global query at the topology level in order to assess the "greenness" of the Fog system based on the different types of instantiated nodes. Listing 6 shows the OCL code of this query that returns a boolean value. To summarize, we get the green energy level for each node in the system and we check whether the majority of the nodes have a high green energy level or not.

Query "Remaining Energy" - Then, we defined a query at the configuration level to compute the remaining available energy to be consumed (i.e. a double value) associated to each node type in the system. The objective is to calculate the amount of available energy with respect to the threshold imposed by the campus administrators. The calculation is based on 1) the total energy consumption of each

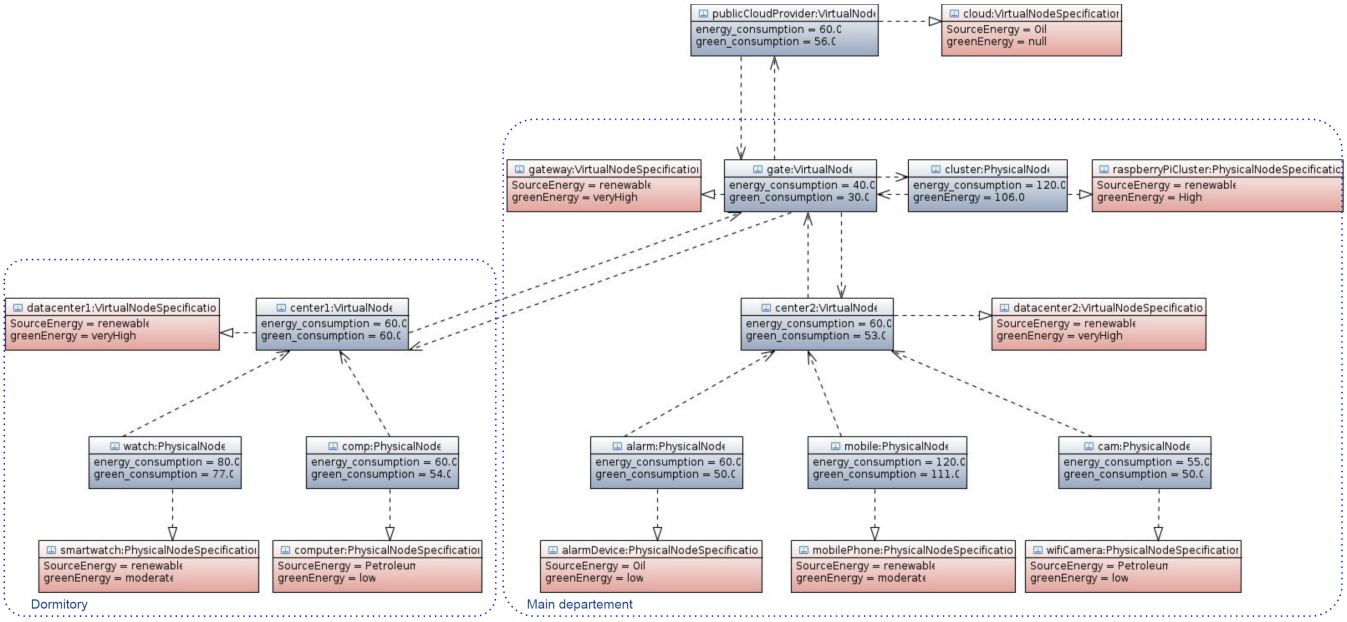

Figure 5: Partial model of the Fog system in the Smart Campus use case.

```
if NodeType.allInstances()->select(f | f.tags->exists(t | t.key =
↪  'greenEnergy' and (t.value = 'low' or t.value = 'null' or
↪  t.value='moderate')))->size() > NodeType.allInstances()->size()/2
↪  then
    'The system is not green'
else
    'The system is Green'
endif
```

Figure 6: OCL code of the IsGreen query in the Smart Campus use case (boolean return value printed in plain readable text).

node and 2) the threshold of 2000 J. The total energy consumption is the sum of the "energy_consumption" of each node in the system. For example, if this sum is 3000 J, a specific message indicate an over-consumption of 1000 J.

Query "Green Energy Consumption" - Finally, we defined another query at the configuration level to compute the amount of consumed green energy compared to the total consumed energy. The objective is to obtain the percentage of green energy consumed by the system. The calculation is based on 1) the "green_consumption" and 2) the "energy_consumption" of each node.

6.1.4 Generation

From the Fog system model of Smart Campus expressed in FML, the *Deployment Abstractor* allows to automatically obtain a corresponding abstract deployment model in EDMM. Figure 7 shows an excerpt of such a model in its YAML syntax. In this model, the *VirtualNodeSpecification* "datacenter1" and corresponding *VirtualNode* "center1" from the Fog system model became the *ComponentType* "datacenter1" and corresponding *Component* "center1", respectively.

For this specific use case, we selected Terraform as our *Deployment Tool*. Then, thanks to the *Abstract Deployment Framework*, a

EDMM-smartcampus-model.yml

```yaml
---
version: edm_1_0

components:

  center1:
    type: compute
    properties:
      energy_consumption : 60
      green_consumption : 60
    operations:

component_types:

  datacenter1:
    extends:
    properties:
     greenEnergy:
       type:
       default_value: veryHigh
     SourceEnergy:
       type:
       default_value: renewable
    description:
    metadata:
    operations:
```

Figure 7: Excerpt of a pivot abstract deployment model in the Smart Campus use case.

corresponding deployment configuration file and associated environment file are generated from the previous abstract deployment model. The environment file contains all the configuration properties mentioned in the EDMM smart campus model components, while the deployment configuration file describe the configuration

of each node in the system. Figure 8 shows an excerpt of such files in their respective syntax, i.e. JSON for the Terraform file and bash for the environment file. In the first file, the deployment-specific information related to the *Component* "center1" is notably described. The second file makes available configuration properties mentioned in the Smart Campus model in EDMM (and previously in the original FML model), namely "energy consumption" and "green consumption". Note that this file can be edited by the engineer, if necessary, to better fit with her actual environment constraints.

aws.tf

```
resource "aws_instance" "center1" {
  ami = "ami-0701e7be9b2a77600"
  instance_type = "t2.micro"
  key_name = aws_key_pair.auth.id
  vpc_security_group_ids = [aws_security_group.center1_security_group.id]
  subnet_id = aws_subnet.edmm.id
  connection {
    type  = "ssh"
    user  = var.ssh_user
    agent = true
    private_key = file(var.private_key_path)
    host = self.public_ip
  }
  provisioner "file" {
    source      = "./env.sh"
    destination = "~/env.sh"
  }
}
```

env.sh

```
#!/bin/bash
set -e
export CENTER1_GREEN_CONSUMPTION=60
export CENTER1_PUBLIC_ADDRESS=null
export CENTER1_ENERGY_CONSUMPTION=60
```

Figure 8: Excerpt of a deployment configuration file (and corresponding environment file) generated for Terraform in the Smart Campus use case.

6.2 Smart Parking: Performance Property

6.2.1 Scenario

The second use case is taken from a published work presenting a solution for drivers to be more efficient when searching for a slot for their cars in a parking lot [7]. Thus, in a parking lot context, the solution describes an example of a Fog system that detect the parked cars in each zone, and indicates available spaces to the drivers via a smart led screen at the entrance.

6.2.2 Model

The Fog system of the monitored parking lot is composed of several *FogAreas*, corresponding to its different sections, that deal with various kinds of nodes (e.g. micro-controllers, cameras). We show in Figure 9 the content of two *FogAreas* but the parking system can be composed of much more *FogAreas*.

The central elements of the shown model are "fogn1" and "fogn2" Fog *VirtualNodes*. These central nodes can be possibly connected to other *FogAreas* of the system, via the "proxyserver" proxy *VirtualNode* and the external "cloudserver" cloud *VirtualNode*. The different devices composing the system (i.e. different types of cameras) are also indireclty connected to these central Fog nodes via "micro1" and "micro2" micro-controller *VirtualNodes*.

At the topology level, we added a "detection_quality" custom *Attribute* to model the detection quality level associated to each camera *NodeType* instantiated in the system. This corresponds to the capability of each camera to always detect any vehicle in the covered zone. For example, if the camera can identify only cars and cannot systematically detect motorcycles, the detection quality will be indicated as low.

At the configuration level, we added a "coveredPlaces" custom *Attribute* to describe the number of slots covered by each Fog *Node* in the system. This corresponds to the number of slots where cameras can possibly detect the cars. We also added another "image_quality" custom *Attribute* to specify the percentage of clarity for the images detected by a given camera.

6.2.3 Queries

In a city's parking lot context, the overall performance of the system is an important issue in order to be able to provide an efficient service to the drivers. Thus, we need the data to be treated by each camera and Fog node to have a sufficiently high level of quality. For the following queries, we considered a smart parking having a reasonable size and thus using 16 cameras nodes and 2 corresponding Fog nodes.

Query "Is Efficient" - To start with, we defined a global query at the topology level to assess the overall efficiency of the parking lot. To this end, we computed the ratio between the number of covered slots and the total number of available cameras. If the ratio is closer to 1, there are possibly too many cameras. If the ratio is closer to 0, there are probably not enough cameras to cover all the slots.

Query "Latency" - Then, we defined a query at the configuration level to compute the time needed for the system to display the available slots on the smart led screen at the entrance of the parking lot. The objective is to evaluate the latency of the overall system. Listing 10 shows the OCL code implementation of this query which returns a real value. To summarize, the calculation is based on 1) the time consumed by the Fog nodes to analyze the images collected from the corresponding 16 cameras (cf. the value from Table 3 in the source paper [7], then encoded in the query) and 2) the time of image enhancement. The time of image enhancement is computed from the "image_quality" of each camera node. If the clarity of the image took by a given camera node is low, an image enhancement is required prior to its analysis.

Query "Reliability" - Finally, we defined another query at the configuration level in order to assess the reliability of the Fog system based on the different types of instantiated nodes. To summarize, we analyze the image quality of each camera in the system by ensuring that the majority of these cameras have an image quality higher than 50%.

6.2.4 Generation

From the Fog system model of Smart Parking expressed in FML, the *Deployment Abstractor* allows to automatically obtain a corresponding abstract deployment model in EDMM. Figure 11 shows an excerpt of such a model in its YAML syntax. In this model, the *PhysicalNodeSpecification* "cameraHighResolution" and corresponding *PhysicalNode* "camera1" from the Fog system model became the *ComponentType* "cameraHighResolution" and corresponding *Component* "camera1", respectively.

For this specific use case, we selected Docker Compose[3] as our *Deployment Tool*. Then, thanks to the *Abstract Deployment Framework*, a corresponding deployment configuration file and associated descriptor file (in this particular case) are generated from the previous abstract deployment model. Figure 12 shows an excerpt of such files in their respective syntax. In the deployment configuration

[3] https://docs.docker.com/compose/

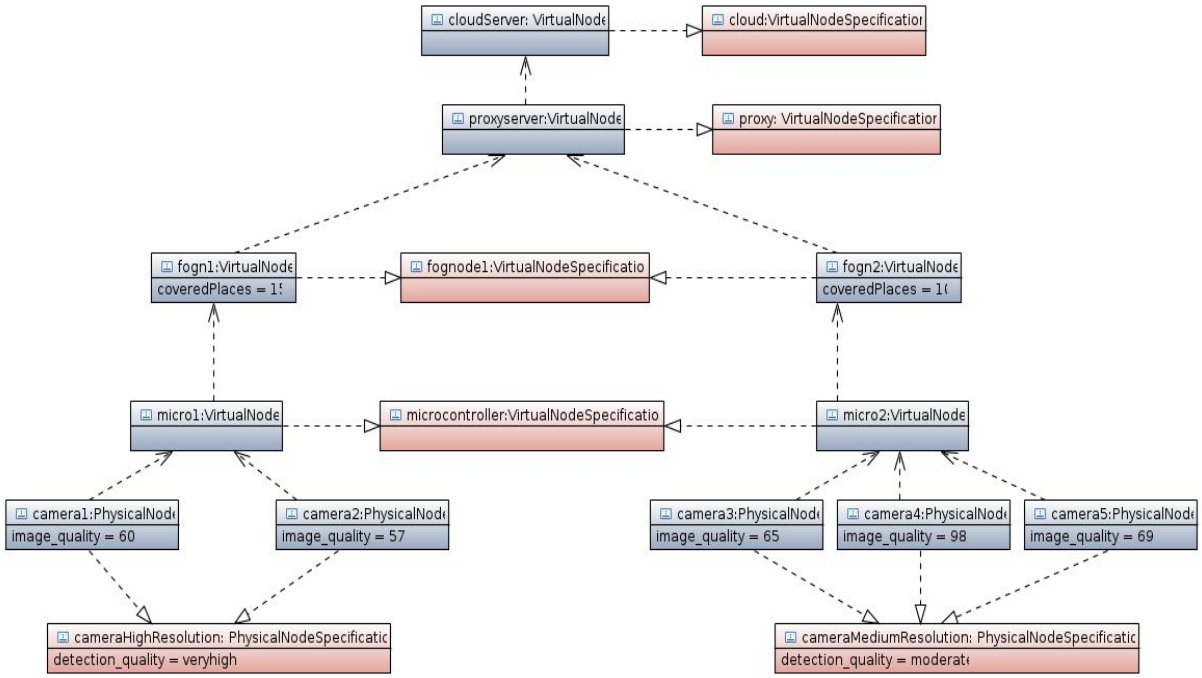

Figure 9: Partial model of the Fog system in the Smart Parking use case.

```
let totalEnhancement: Real =
  NodeType.allInstances()
    ->select(f | f.tags->exists(t | t.key = 'detection_quality' and
    ↪  (t.value = 'low' or t.value = 'verylow')))
    ->collect(f | if f.tags->exists(t | t.key = 'detection_quality' and
    ↪  t.value = 'low') or f.tags->exists(t | t.key = 'detection_quality'
    ↪  and t.value = 'verylow') then 0.702 else 0 endif)
    ->sum() + 0.00787
in
'the latency of the system is ' + totalEnhancement.toString() +  's'
```

Figure 10: OCL code of the Latency query in the Smart Parking use case (real return value printed in plain readable text).

file, the deployment-specific information related to the *Component* "camera1" is notably described. In the associated Docker file, the configuration property "image quality" from the Smart Parking model in EDMM (and previously from the original FML model) is directly visible.

6.3 Smart Hospital: Security Property

6.3.1 Scenario

The third use case is taken from a published work presenting a Model Driven Bandwidth Allocation framework based on a priority algorithm [36]. In an hospital context, the solution proposes a bandwidth allocator device that analyzes incoming events arising from blood pressure, body temperature, and glucose sensors, to finally provide the required bandwidth. The underlying algorithm is based on a priority algorithm: when two critical events occurs simultaneously, the bandwidth allocator proceeds with the event

EDMM-smartparking-model.yml ×

```yaml
---
version: edm_1_0

components:

  camera1:
    type: compute
    properties:
      image_quality : 60
    operations:


component_types:

 cameraHighResolution:
    extends:
    properties:
     detection_quality:
       type:
       default_value: veryhigh
    description:
    metadata:
    operations:
```

Figure 11: Excerpt of a pivot abstract deployment model in the Smart Parking use case.

having the shortest execution time and then provide a maximum bandwidth allocation to this particular event.

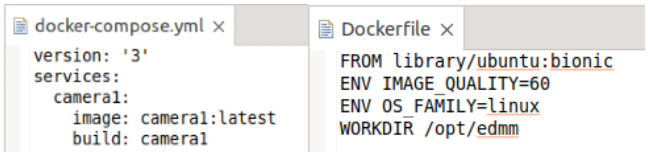

Figure 12: Excerpt of a deployment configuration file (and corresponding Dockerfile) generated for Docker Compose in the Smart Parking use case.

6.3.2 Model

The Fog system of the monitored hospital is composed of several *FogArea*s corresponding to the different sections of the hospital that deal with the various kinds of patients (e.g. adults, babies). We show in Figure 13 only the content of one *FogArea*, the other *FogArea*s have a similar structure and content.

The central element of the shown *FogArea* is a *VirtualNode* identified as "fognode1". This central node can be connected to other *FogArea*s of the system (for other sections of the hospital) via an external cloud *VirtualNode* identified as "cln1". The different nodes composing the shown *FogArea* are also connected to this central "fognode1" Fog node, either directly (for the "iotdevice1" and "iotdevice2" IoT *PhysicalNode*s) or indirectly via the "bandn1" bandwith allocator *VirtualNode* (for the "bloodPressureAlertEvent", "temperatureAlertEvent" and "glucoseAlertEvent" *VirtualNode*s).

At the topology level, we added a "securityLocation" custom *Attribute* to be able to model the security level associated to each *NodeType* instantiated in the system. This security level is based on the location of the nodes of a given type. If the node is in a zone completely protected by a firewall, the security level will be high. If it can be accessed by a third party which is not protected, it will be moderated. In the case where the node is located in a non-protected area, the security level will be low. We also added another "impact" custom *Attribute* to be able to represent the security impact associated to each *NodeType*. We grouped the impact level by categories: A means that the data and information stored in this node are very important, B moderately important, and C not so important.

At the configuration level, we added a "authorityLevelChecking" custom *Attribute* to be able to specify the level of the authority associated to each node. We estimate the access protection by considering numerical values ranging from 1 to 5. If there is no particular protection (no passwords, token, fingerprint, VPN, etc.), the level will be 1. If the access needs multiple verification steps, the authority level will be 5.

6.3.3 Queries

In an hospital context, preserving the privacy of the patients data is fundamental. Thus, we need the different information of each patient to be resilient and well secured. To be realistic, and to have a sufficiently rich Fog system model for the queries, we considered a large hospital with multiple sections and many different patients and devices.

Query "IsSecured" - To start with, we defined a global query at the topology level in order to assess the security of the Fog system based on the different types of instantiated nodes. Listing 14 shows the OCL code of this query that returns a boolean value. To summarize, we analyse the security level of each type of node in the system by ensuring that the majority of them are actually located in a high-security location.

Query "RiskScore" - Then, we defined another query at the topology level to compute the risk score (i.e. an integer value) associated to each node type in the system. The objective is to estimate the potential impact of a cyber-attack on each type of node. The calculation is based on 1) the impact of the data possibly leaking from this node type and 2) the probability of being attacked by a hacker [20]. The probability of the attack is computed from the "securityLocation" of the zone where the nodes of a given type are located: the probability of having an attack on a node is higher when the security level of the zone in which this node is located is low.

Query "AuthorityLevelChecking" - Finally, we defined a query at the configuration level to obtain sequences of nodes grouping them according to their respective level of authorization. This way, we can provide lists of nodes having easy, medium and difficult access. This can have a direct impact on where some data can actually be stored or not.

6.3.4 Generation

From the Fog system model of Smart Hospital expressed in FML, the *Deployment Abstractor* allows to automatically obtain a corresponding abstract deployment model in EDMM. Figure 15 shows an excerpt of such a model in its YAML syntax. In this model, the *VirtualNodeSpecification* "bandwidthAllocator1" and corresponding *VirtualNode* "bandn1" from the Fog system model became the *ComponentType* "bandwidthAllocator1" and corresponding *Component* "bandn1", respectively.

For this specific use case, we selected Azure[4] as our *Deployment Tool*. Then, thanks to the *Abstract Deployment Framework*, a corresponding deployment configuration file and associated environment file (in this particular case) are generated from the previous abstract deployment model. Figure 16 shows an excerpt of such files in their respective syntax. In the configuration file, the deployment-specific information related to the *Component* "bandn1" is notably described. The environment file contains the configuration property "authorityLevelChecking" from the Smart Hospital model in EDMM (and previously in the original FML model).

7 IMPLEMENTATION

Based on our own experience and technical expertise, we decided to implement in the Eclipse ecosystem an initial version of the tooling support associated with the proposed Fog Modeling Language, verification and generation Support, as well as current use cases. However, both the VeriFog conceptual approach and the FML language specification are technology-independent. As a consequence, they may be redeveloped in other technical environments if required in the future (e.g. by our partner company Smile for industrial purposes).

FML has been implemented by using the Eclipse Modeling Framework (EMF)[5] and related tools. For the abstract syntax, we specified a dedicated metamodel in Ecore. For the concrete syntax, we developed a grammar in Xtext[6] that we connected to this metamodel. This allowed us to produce a corresponding textual editor with

[4]https://azure.microsoft.com/
[5]https://www.eclipse.org/modeling/emf/
[6]https://www.eclipse.org/Xtext/

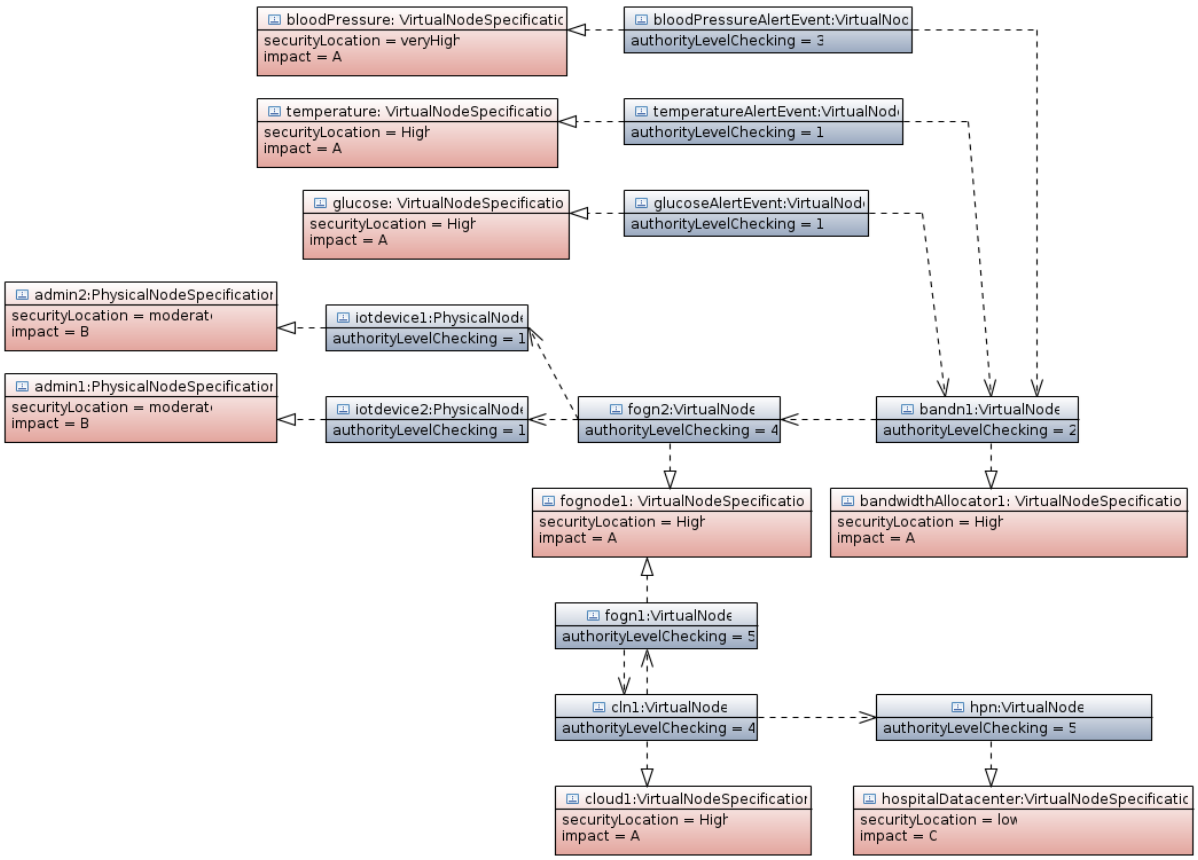

Figure 13: Partial model of the Fog system in the Smart Hospital use case.

```
if VirtualNodeSpecification.allInstances()->isEmpty() then
    -- Handle the case where there are no node types in the system
    'Cannot determine if the system is secured because there are no node
    ↪  types.'
else
    if VirtualNodeSpecification.allInstances()->size() = 0 then
        -- Handle the case where there are no node types in the system
        'Cannot determine if the system is secured because there are no
        ↪  node types.'
    else
       if VirtualNodeSpecification.allInstances()->select(f |
       ↪  f.tags->exists(t | t.key = 'securityLocation' and (t.value =
       ↪  'High' or t.value = 'veryHigh' or t.value='moderate')))->size()
       ↪  > VirtualNodeSpecification.allInstances()->size()/2 then
        'The system is secured.'
    else
            'The system is not secured.'
        endif
    endif
endif
```

Figure 14: OCL code of the IsSecured query in the Smart Hospital use case (real return value printed in readable text).

basic features (syntax highlighting, code completion, syntax checks, etc.). Thus, we provide an editing environment for Fog System Architects to write their Fog system models using several YAML files. As in the VeriFog approach, they can also get their models as full EMF-compatible models, and eventually serialize them as XMI files for further sharing or usage with other tools.

For the Verification Support, we implemented all the presented queries in two different languages: 1) Java to show the possible use of FML in combination with a general-purpose programming language, and 2) the Object Constraint Language (OCL)[7] to show the compatibility of our language with a standard navigation model and query language. The final reporting on the query execution results is currently implemented as strings to be displayed onto the Eclipse workbench's console, or to be printed into textual files to be then shared with the Fog System Architects.

The generation component is also partially implemented by relying on EMF, the EMF-compatible implementation of FML described before, and other related tools. As the *Abstract Deployment metamodel*, we decided to use the Essential Deployment MetaModel (EDMM) [48] and our own Ecore implementation of this metamodel. Then, we implemented the *Deployment Abstractor* and its FML-to-EDMM mapping by specifying a dedicated model-to-model transformation in the ATL Transformation Language (ATL) [26]. We also implemented a complementary model-to-text transformation, in Acceleo [38], for transforming EDMM *Abstract Deployment models* into corresponding YAML textual files.

[7] https://projects.eclipse.org/projects/modeling.mdt.ocl

EDMM-smarthospital-model.yml

```
---
version: edm_1_0

components:

  bandn1:
    type: compute
    properties:
      authorityLevelChecking : 2
    operations:


component_types:

  bandwidthAllocator1_vm:
    extends:
    properties:
     securityLocation:
       type:
       default_value: High
     impact:
       type:
       default_value: A
    description:
    metadata:
    operations:
```

Figure 15: Excerpt of a pivot abstract deployment model in the Smart Hospital use case.

deploy.json

```
{
  "contentVersion" : "1.0.0.0",
  "parameters" : {
    "DnsName" : {
      "type" : "string"
    },
    "vm_bandn1_adminUserName" : {
      "type" : "string",
      "defaultValue" : "vm_bandn1_Admin"
    },
```

env.sh

```
#!/bin/bash
set -e
export BANDN1_AUTHORITYLEVELCHECKING=2
export BANDN1_MACHINE_IMAGE=null
export BANDN1_INSTANCE_TYPE=null
export BANDN1_PUBLIC_ADDRESS=null
export BANDN1_OS_FAMILY=linux
```

Figure 16: Excerpt of a deployment configuration file (and corresponding environment file) generated for Azure in the Smart Hospital use case.

Moreover, we decided to use the Deployment Model Abstraction Framework (DeMAF) [43] as the *Abstract Deployment framework*. DEMAF mostly relies on JetBrains MPS [13] and supports Terraform among several other *Deployment Tools*. Thanks to the DEMAF *Abstract Deployment framework*, we can generate The Terraform *Deployment Configuration files* in the context of our three use cases.

All the resources described in the paper, including the complete source code of the proposed Fog Modeling Language, the models of the different use cases, the implementation of the related queries and the generation component are available in an open repository[8].

8 DISCUSSION

8.1 Current Scope and Limitation

The presented work capitalizes on previously existing work in the context of Cloud systems. However, as seen in the literature, there are significant differences between Cloud and Fog systems. In fact, the Edge and IoT layers of Fog systems have particular types of elements and properties which are not present in traditional Cloud systems (IoT devices or material at the edge with their hardware and software properties, heterogeneous communication means, etc.). Notably, the concept of Fog Area is key to model several subsystems supporting the decentralized characteristic of Fog systems. Moreover, explicitly characterizing network types and links is fundamental for Fog systems in order to accurately model the communications between the different Fog areas and their elements. Finally, the modeling of applications and offered services is also important in the context of highly distributed Fog systems. Up to our current knowledge, the already existing approaches and languages do not allow to properly model the large variety of Fog systems, specify their characteristics, and express related non-functional properties to be verified. Our generic and customizable approach is a direct answer to this challenge.

VeriFog and FML are generic because they allow to model different, and potentially any, kinds of heterogeneous Fog systems within different applications domains while also considering different types of non-functional properties to be verified. This notably includes properties that are usually more critical in Fog systems than in Cloud systems (e.g. security or performance/latency issues). Still, it is important to note that our intent is not to support by default in our approach all the possible Fog-specific issues. For example, the resilience to faults is deliberately not a 'first-class' concept in FML. This is the reason why we designed our approach as customizable so that 1) FML can be refined to add the needed attributes/metadata thus enriching the Fog system models as required and 2) additional queries can be defined accordingly to verify any issues (e.g. fault tolerance) thanks to these complementary attributes/metadata. More generally, genericity and extensibility are key relevant characteristics we can directly rely on when addressing other phases of the Fog system's life cycle.

Automation is also a particularly important complementary aspect of VeriFog. Indeed, we recently added to our approach the capability to support the automated generation of actual deployment configuration files. The work presented in this paper showed that providing such an automation support, from high-level specifications of Fog systems (i.e. FML models) to low-level configuration files, is actually feasible. This is notably a pre-requisite for the following phases of the Fog system's life cycle. For example, we could then verify deployment-related QoS properties by relying on these configuration files and combining a compatible deployment tool (e.g., Terraform) with associated QoS solutions (e.g., Infracost[9], tfsec[10]). To go one step further in terms of automation, we could also capitalize on the different artifacts we can already generate (i.e. models, files) in other contexts. For instance, we could automate the chaining of the various tools, frameworks and solutions involved within several different phases of the Fog system's life cycle. We could also provide a specific automation support in the context of other phases, for instance at deployment time (for resource management) and execution time (for self-adaptation). This way, we will further explore the support for other different Fog-specific issues that cannot be easily addressed at design time.

[8] https://zenodo.org/records/13132862

[9] https://www.infracost.io/

[10] https://aquasecurity.github.io/tfsec

8.2 Lessons Learned

We sustained a regular interactions pace with our partner company Smile to cope with the industrial relevance and applicability of our work. This notably allowed us to collect practical experience and feedback when elaborating on and evaluating the approach presented in this paper.

The ability to model different kinds of Fog systems in different application domains while considering different types of non-functional properties to be verified (e.g. security, performance, energy) was demonstrated and discussed. In addition, it also appeared that Fog system models are relevant to generate actual configuration files and then deploy initial versions of working Fog systems. An adaptation can then be carried out on these deployment configurations, for example by a developer or a system administrator. In doing so, a *reconciliation* can occur between the original architect (i.e. FSA) and the system's operator. The objectives of this reconciliation can vary: increasing volumes and adapting costs, updating components and configurations, complying with APIs or requirements that may have changed over time, etc. Dealing with such aspects naturally enriches the previous work on Fog system models and their verification.

Currently, it could be challenging to identify in the industry sufficient FSA workforce having both a solid system architectural/modeling expertise (design part) and sufficient technical/programming skills (verification part) for Fog systems. Considering the rarity of architects, means to extend the lifetime of their proposed models prevail over other considerations, such as direct compatibility with the common administration tools. To reduce this possible gap, we prototyped FML with a YAML textual concrete syntax that is quite close to the kind of configuration files the DevOps engineer usually handles. FML models and their reuse could also be realized via dedicated graphical interfaces, possibly integrated into existing DevOps solutions, for example to replace OCL (as mostly known in the MDE community) and Java (sometimes too general-purpose and verbose) in VeriFog.

Moreover, as DevOps tools and methodologies are gaining attraction in both scientific and industrial communities, the involved actors have to support a larger and larger operational perimeter. As a consequence, architects in general (FSA included) will be expected not only to design but also to verify, deploy and manage production-ready Fog systems. This consideration gives momentum to the work presented in this paper. Indeed, while high-level descriptions are used for generating large scale deployment configurations, readability and control are possible all along the model-based solution. This enables both automated and human evaluation at the different steps of the Fog system's life cycle. To support such an evaluation in practice, FML is a good facilitator as a commonly shared language for modeling Fog systems.

9 RELATED WORK

9.1 Simulation and Emulation

Designing, deploying, and evaluating Fog-based applications is a complex and costly endeavor. To this end, several works proposed to rely on simulation and emulation tools associated to dedicated languages. For example, Fogify [42] provides a modeling language for defining Fog topologies (encapsulating QoS constraints) which extends the Docker-compose specification. However, contrary to our approach, it does not directly cover the modeling of configurations and mostly focuses on Docker-based infrastructures. iFogSim [23], based on CloudSim [14], provides a modeling language for IoT devices and associated Fog resources, in order to enable the simulation of scheduling policies based on multiple QoS criteria. In this case, contrary to us, they have very specific runtime objectives. Another example is EmuFog [33] that enables the from-scratch specification of Fog Computing infrastructures and the emulation of real applications and workloads. Once again, they are mostly operating at runtime and with a specific target in mind (i.e. workload management). Even if all these work intend to somehow verify Fog systems before their deployments, they are closely related to particular industrial tools (e.g. Docker) or simulators (e.g. CloudSim). Moreover, they are not promoting a more generic modeling language to be exploited at each phase of the system's life cycle (cf. Section 10).

9.2 Quality of Service (QoS)

There are also a few model-based solutions targeting the QoS of Fog systems. For example, FogTorch [10] proposes a semi-formal language that considers various relevant Fog aspects in order to determine QoS-aware deployments of IoT application. Its successor FogTorchΠ [11] exploits Monte Carlo simulation models to take into account possible variations of the QoS and eligible deployments of Fog applications. SMADA-Fog [40] provides a semantic model-driven approach to support the deployment and adaptation of container-based applications in Fog Computing. The used language relies on two metamodels implemented within the Node-RED[11] deployment tool. However, in all cases, these solutions are strongly deployment phase-oriented, and do not target an end-to-end holistic approach as we do (cf. Section 10).

9.3 Life Cycle Management and Orchestration

As introduced before, the design and implementation of a complex Fog system can quickly become really challenging. Some works partially address the life cycle of Fog systems in order to overcome this complexity [22, 50]. However, they mainly deal with the execution phase and are not necessarily meant to be generalized to the support for the whole life cycle, notably since they do not come with reusable modeling languages. Indeed, a recent survey [17] found out that orchestration is an over-used word that sometimes refers to life cycle management. This semantic shortcut unfortunately leads the community to a too strong focus on runtime concerns. In fact, this survey shows that Fog orchestrated entities, when properly modeled as services, tasks, pipe-lines, workflows, etc. could be used more intensively in the context of Fog life cycle management.

From an industrial perspective, there are also relatively recent initiatives. For example, Fogernetes [46], based on Kubernetes[12], compares and maps the requirements of application components to available Fog nodes in order to ensure an optimal deployment according to some non-functional properties. However, this solution specifically relies on the Kubernetes, and focuses on non-functional

[11] https://nodered.org/
[12] https://kubernetes.io

properties while we could also possibly support functional properties as well (even if this is not the focus of the work presented in this paper). Going further, GitOps [8] is one of the main trends in the DevOps [27] ecosystem for Continuous Deployment that promotes infrastructure automation in highly distributed applications. A recent work [29] describes an initial implementation of GitOps at the Edge/IoT-level based on KubeEdge[13]. Such an approach could be interesting to generalize in the context of other similar technical frameworks, by relying on FML as a basis for instance.

9.4 Model-based Approaches for the Deployment of Distributed Systems

As introduced earlier in the paper, a systematic review of the existing declarative deployment tools has already been performed [49]. It notably resulted in the Essential Deployment Metamodel (EDMM) that generalizes the main elements supported by these deployment tools. Also, it allowed to combine multiple deployment tools in a more automated way [47]. Based on EDMM, DeMAF [45] is an abstract deployment framework that allows abstract deployment models in EDMM to be transformed into technology-specific deployment files. In the extended VeriFog, we integrate both EDMM and DeMAF into our our *Pre-Deployment Generation* component. We go a step further by now offering a comprehensive workflow from design time verification to pre-deployment time generation.

DOML [15, 16] is another model-based solution that provides a DevOps Modelling Language. It aims at describing Cloud applications in this language, and then generating the corresponding executable Infrastructure-as-code (IaC) code for deployment tools such as Ansible, Terraform, and Cloudify[19]. The underlying idea is to consider a single modelling paradigm addressing infrastructure provisioning, application deployment, and configuration all at once. However, DOML does not allow to model a complete system and is currently limited in terms of support for the verification of different non-functional properties. In addition, it does not specifically target the support for Fog systems such as in VeriFog.

CloudCAMP [9] proposes a model-driven engineering and generative programming approach integrated into an automated deployment and management platform for Cloud applications. As a result, CloudCAMP transforms partial specifications (i.e. models) into deployable IaC code possibly targeting different Cloud providers. Although this solution supports the modeling of application attributes and constraints, it does not intend to support verification activities. Moreover, contrary to VeriFog, it focuses on the modeling of the associated workflow and does not support the complete modeling of Fog systems.

To summarize, most of the currently existing approaches rely on a Domain-Specific Language (DSL) they propose to then generate actual deployment configuration files. Overall, these approaches are not intended to support the modeling of Fog systems (at least not natively). In addition, they do not allow neither the verification of different kinds of non-functional properties at design time before the generation of deployment configuration files for various deployment tools. The main objective of VeriFog is to make a concrete step in this direction.

[13]https://kubeedge.io

10 CONCLUSION AND FUTURE WORK

In this paper, we proposed a generic approach for Fog system architects and engineers to model their Fog systems, verify different kinds of relevant non-functional properties over them, and finally generate actual deployment configuration files potentially targeting different deployment tools. To this end, we notably introduced a customizable Fog Modeling Language (FML) that allows specifying Fog system models that can then be navigated, queried and transformed as needed. We experimented with our verification and generation approach, and underlying modeling language, by using them in the context of different use cases covering various application domains of Fog Computing.

The work described in this paper belongs to a more global research effort intending to support the complete life cycle of Fog systems. In fact, the presented modeling support for the design-time verification of Fog systems and the pre-deployment generation of configuration files is only a first step towards a wider usage of the proposed FML. The longer term objective is to generalize the (re)use of FML models in order to improve the support for other major activities within the system's life cycle. For example, at development time, we would like to consider the FML models as inputs for semi-automatically generating different wrappers or configuration files for various technical platforms (e.g. frameworks, schedulers) and kinds of Fog resources (e.g. Cloud servers, IoT devices). At deployment time, we also plan to rely on FML models in order to partially automate the allocation and/or provisioning of different Fog resources, the chaining of hosted services, the loading/unloading of related tasks, etc. At execution time, we already envision the use of FML models as a way to allow the self-adaptation, in some relevant cases, of the modeled systems (or at least parts of them). For example, one important goal is to make the systems more resilient to various types of faults or errors, or more respectful of high level properties defined in the models.

Another important objective, in direct collaboration with our industrial partner Smile, is to work on the integration of the proposed VeriFog approach, Fog Modeling Language, and other related contributions into real DevOps CI/CD pipelines. Such pipelines can be used in practice to support the production, maintenance and evolution of real Fog systems in industrial contexts, in a managed, offline, way. From an industrial perspective, VeriFog, FML and the corresponding global research effort appear to be a promising way to improve the overall QoS of the target Fog systems while possibly reducing the associated development and management costs.

ACKNOWLEDGMENT

This work was funded by the French Agence Nationale de la Recherche Technologique (ANRT Cifre PhD grant) and by the French Agence Nationale de la Recherche (ANR-20-CE25-0017 grant – SeMaFoR project).

REFERENCES

[1] M. Aazam, M. St-Hilaire, C.-H. Lung, and I. Lambadaris. Mefore: Qoe based resource estimation at fog to enhance qos in iot. In *23rd International Conference on Telecommunications (ICT 2016)*, pages 1–5, New York, U.S.A., 2016. IEEE.

[2] Z. Al-Shara, F. Alvares, H. Bruneliere, J. Lejeune, C. Prud'Homme, and T. Ledoux. CoMe4ACloud: An end-to-end framework for autonomic Cloud systems. *Future Generation Computer Systems*, 86:339–354, 2018.

[3] A. Alidra, H. Bruneliere, H. Coullon, T. Ledoux, C. Prud'Homme, J. Lejeune, P. Sens, J. Sopena, and J. Rivalan. SeMaFoR - Self-Management of Fog Resources with Collaborative Decentralized Controllers. In *18th Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2023)*, Melbourne, Australia, May 2023. IEEE.

[4] A. Alidra, H. Bruneliere, and T. Ledoux. A Feature-based Survey of Fog Modeling Languages. *Future Generation Computer Systems*, 138:104–119, 2023.

[5] Amazon. Amazon Web Services (AWS). https://aws.amazon.com/, 2024. Accessed: 2024-07-11.

[6] 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. *Communications of the ACM*, 53(4):50–58, 2010.

[7] K. S. Awaisi, A. Abbas, M. Zareei, H. A. Khattak, M. U. S. Khan, M. Ali, I. U. Din, and S. Shah. Towards a fog enabled efficient car parking architecture. *IEEE Access*, 7:159100–159111, 2019.

[8] F. Beetz and S. Harrer. Gitops: The evolution of devops? *IEEE Software*, 39(4):70–75, 2021.

[9] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda. Cloudcamp: A model-driven generative approach for automating cloud application deployment and management. Technical Report ISIS-17-105, Vanderbilt University, Nashville, TN, USA, 2017.

[10] A. Brogi and S. Forti. Qos-aware deployment of iot applications through the fog. *IEEE internet of Things Journal*, 4(5):1185–1192, 2017.

[11] A. Brogi, S. Forti, and A. Ibrahim. How to best deploy your fog applications, probably. In *IEEE 1st International Conference on Fog and Edge Computing (ICFEC 2017)*, pages 105–114, New York, U.S.A., 2017. IEEE.

[12] H. Bruneliere, Z. Al-Shara, F. Alvares, J. Lejeune, and T. Ledoux. A Model-based Architecture for Autonomic and Heterogeneous Cloud Systems. In *8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 201–212, Setubal, Portugal, 2018. SciTePress.

[13] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio. *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer Nature, Berlin, Germany, 2021.

[14] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[15] M. Chiari, E. D. Nitto, A. N. Mucientes, and B. Xiang. Developing a new devops modelling language to support the creation of infrastructure as code. In *European Conference on Service-Oriented and Cloud Computing (ESOCC 2022)*, pages 88–93, Berlin, Germany, 2022. Springer.

[16] M. Chiari, B. Xiang, S. Canzoneri, G. N. Nedeltcheva, E. Di Nitto, L. Blasi, D. Benedetto, L. Niculut, and I. Škof. Doml: a new modelling approach to infrastructure-as-code. *Information Systems*, online:102422, 2024.

[17] B. Costa, J. Bachiega, L. R. de Carvalho, and A. P. F. Araujo. Orchestration in fog computing: A comprehensive survey. *ACM Computing Surveys*, 55(2), jan 2022.

[18] R. Das and M. M. Inuwa. A Review on Fog Computing: Issues, Characteristics, Challenges, and Potential Applications. *Telematics and Informatics Reports*, 10:100049, 2023.

[19] DELL. Cloudify – an open-source cloud orchestration framework. https://docs.cloudify.co/, 2024. Accessed: 2024-04-19.

[20] V. Dumbravă and S. V. Iacob. Using probability – impact matrix in analysis and risk assessment projects. *Journal of Knowledge Management, Economics, and Information Technology*, 3:1–7, 2013.

[21] A. El-Hokayem, M. Bozga, and J. Sifakis. A framework for the specification and validation of dynamic reconfigurable systems. *SIGAPP Applied Computing Review*, 21(2):18–32, jul 2021.

[22] F. Guim, T. Metsch, H. Moustafa, T. Verrall, D. Carrera, N. Cadenelli, J. Chen, D. Doria, C. Ghadie, and R. G. Prats. Autonomous lifecycle management for resource-efficient workload orchestration for green edge computing. *IEEE Transactions on Green Communications and Networking*, 6(1):571–582, 2022.

[23] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya. ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[24] M. Haghi Kashani, A. M. Rahmani, and N. Jafari Navimipour. Quality of service-aware approaches in fog computing. *International Journal of Communication Systems*, 33(8):e4340, 2020.

[25] J. Huang, Q. Duan, Q. Chen, Y. Sun, Y. Tanaka, and W. Wang. Guaranteeing end-to-end quality-of-service with a generic routing approach. *SIGAPP Applied Computing Review*, 14(2):8–22, jun 2014.

[26] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Science of Computer Programming*, 72:31–39, 2008.

[27] L. Leite, C. Rocha, F. Kon, D. Milojicic, and P. Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[28] S. Li, L. D. Xu, and S. Zhao. The Internet of Things: a Survey. *Information systems frontiers*, 17:243–259, 2015.

[29] R. López-Viana, J. Díaz, and J. E. Pérez. Continuous deployment in iot edge computing: A gitops implementation. In *17th Iberian Conference on Information Systems and Technologies (CISTI 2022)*, pages 1–6, New York, U.S.A., 2022. IEEE.

[30] M. M. Mahmoud, J. J. Rodrigues, K. Saleem, J. Al-Muhtadi, N. Kumar, and V. Korotaev. Towards energy-aware fog-enabled cloud of things for healthcare. *Computers & Electrical Engineering*, 67:58–69, 2018.

[31] F. Marconi, M. M. Bersani, and M. Rossi. A model-driven approach for the formal verification of storm-based streaming applications. *SIGAPP Applied Computing Review*, 17(3):6–15, nov 2017.

[32] K. Massey, N. Moazen, and T. Halabi. Optimizing the allocation of secure fog resources based on qos requirements. In *2021 8th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2021 7th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 143–148, New York, U.S.A., 2021. IEEE.

[33] R. Mayer, L. Graser, H. Gupta, E. Saurez, and U. Ramachandran. Emufog: Extensible and scalable emulation of large-scale fog computing infrastructures. In *IEEE Fog World Congress (FWC 2017)*, pages 1–6, New York, U.S.A., 2017. IEEE.

[34] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[35] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 20(1):416–464, 2018.

[36] M. Naz, M. Rashid, F. Azam, M. Abbas, Y. Rasheed, M. H. Sinky, and M. W. Anwar. A model driven framework for efficient bandwidth allocation in fog computing using priority algorithm. In *2022 2nd International Conference on Computing and Information Technology (ICCIT)*, pages 39–44, New York, U.S.A., 2022. IEEE.

[37] F. Neves, R. Vilaça, and J. Pereira. Detailed black-box monitoring of distributed systems. *SIGAPP Applied Computing Review*, 21(1):24–36, jul 2021.

[38] Obeo. Acceleo – Generate anything from any EMF model. https://eclipse.dev/acceleo/, 2024. Accessed: 2024-07-11.

[39] J. Oldevik, T. Neple, R. Grønmo, J. Aagedal, and A.-J. Berre. Toward Standardised Model to Text Transformations. In *European Conference on Model Driven Architecture-Foundations and Applications (ECMFA 2005)*, pages 239–253, Berlin, Germany, 2005. Springer.

[40] N. Petrovic and M. Tosic. Smada-fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory*, 101:102033, 2020.

[41] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge Computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[42] M. Symeonides, Z. Georgiou, D. Trihinas, G. Pallis, and M. D. Dikaiakos. Fogify: A fog computing emulation framework. In *5th ACM/IEEE Symposium on Edge Computing (SEC 2020)*, pages 42–54, New York, U.S.A., 2020. IEEE.

[43] University-Stuttgart. Deployment Model Abstraction Framework (DeMAF). https://github.com/UST-DeMAF, 2024. Accessed: 2024-07-11.

[44] W. T. Vambe, C. Chang, and K. Sibanda. A review of quality of service in fog computing for the internet of things. *International Journal of Fog Computing (IJFC)*, 3(1):22–40, 2020.

[45] M. Weller, U. Breitenbücher, S. Speth, and S. Becker. The deployment model abstraction framework. In *26th International Conference on Enterprise Design, Operations, and Computing (EDOC 2022)*, pages 319–325, Berlin, Germany, 2022. Springer.

[46] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge. Fogernetes: Deployment and management of fog computing applications. In *IEEE/IFIP Network Operations and Management Symposium (NOMS 2018)*, pages 1–7, New York, U.S.A., 2018. IEEE.

[47] M. Wurster, U. Breitenbücher, A. Brogi, F. Diez, F. Leymann, J. Soldani, and K. Wild. Automating the deployment of distributed applications by combining multiple deployment technologies. In *11th International Conference on Cloud Computing and Services Science (CLOSER 2021*, pages 178–189, Lisbon, Portugal, 2021. INSTICC.

[48] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov. The edmm modeling and transformation system. In *Service-Oriented Computing–ICSOC 2019 Workshops: WESOACS, ASOCA, ISYCC, TBCE, and STRAPS, Toulouse, France, October 28–31, 2019, Revised Selected Papers 17*, pages 294–298, Berlin, Germany, 2020. Springer.

[49] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani. The essential deployment metamodel: a systematic review of deployment automation technologies. *SICS Software-Intensive Cyber-Physical Systems*, 35:63–75, 2020.

[50] R. Xie, Q. Tang, S. Qiao, H. Zhu, F. R. Yu, and T. Huang. When serverless computing meets edge computing: Architecture, challenges, and open issues. *IEEE Wireless Communications*, 28(5):126–133, 2021.

[51] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289 – 330, 2019.