A Maude-based rewriting approach to model and verify Cloud/Fog self-adaptation and orchestration⋆

Khaled Khebbeb[a,*], Nabil Hameurlain[a] and Faiza Belala[b]

[a]LIUPPA laboratory, University of Pau and countries of Adour, France
[b]LIRE laboratory, Constantine 2 University, Algeria

ARTICLE INFO

Keywords:
Self-adaptation
Orchestration
Fog computing
Cloud computing
Formal methods
Rewriting logic
Linear Temporal Logic
Maude

ABSTRACT

In the IoT-Fog-Cloud landscape, IoT devices are connected to numerous software applications in order to fully operate. Some applications are deployed on the Fog layer, providing low-latency access to resource, whilst others are deployed on the Cloud to provide important resource capabilities and process heavy computation. In this distributed landscape, the deployment infrastructure has to adapt to the highly dynamic requirements of the IoT layer. However, due to their intrinsic properties, the Fog layer may lack of providing sufficient amount of resource while the Cloud layer fails ensuring low-latency requirements. In this paper, we present a rewriting-based approach to design and verify the Cloud-Fog self-adaption and orchestration behaviors in order to manage infrastructure reconfiguration towards achieving low-latency and resources quantity trade-offs. We rely of the formal specification language Maude to provide an executable solution of these behaviors basing on the rewriting logic and we express properties with linear temporal logic (LTL) to qualitatively verify the adaptations correctness.

1. Introduction

Fog computing [32] is an emerging paradigm that extends the Cloud [20] to be closer to the things that produce and act on IoT [3] data. The Fog employs resources, called Fog nodes, that can be deployed anywhere with a network connection: on a facility indoors, on top of a power pole, in/on a vehicle, alongside a railway track, etc. Any device with computing power, storage capacity, and network connectivity can be a Fog node. Examples include local dedicted servers, industrial controllers, switches, routers, embedded servers and so on. The idea of the Fog is analyzing IoT data close to where it is collected (i.e., at the edge of the IoT-Cloud continuum network) mainly to minimize latency. It offloads important amount of network traffic from the core network (i.e., the Cloud) and it keeps sensitive data inside the network (i.e. close to the IoT devices) [4, 26]. The Fog is an important and clever solution for latency sensitive computation and storage. It is however not ready to fully replace the Cloud. The latter is still highly used for computation-intensive and latency-insensitive activity. The Cloud provides massive quantity of computing resource gathered in distant facilities called data-centers (hence the latency). It is a strong reliable source (and backup solution) of computing and storage power, yet it fails answering low-latency requirements, whereas Fog nodes are globally categorized with limited computing resource capabilities, possess relatively low storage and energy abilities but ensure low-latency computation. Figure 1 gives a conceptual vision of the Cloud-Fog resource offering in the IoT context.

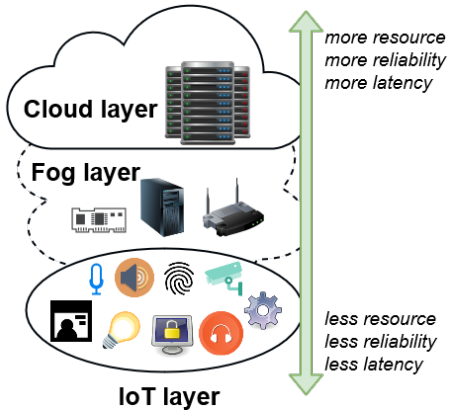

Figure 1: Resource offering in the IoT-Fog-Cloud landscape

In the IoT-Fog-Cloud landscape, devices in the IoT layer interract with software applications (ex. cloudlets, micro-services, etc.) that can be deployed on the Cloud layer servers (ex. Virtual machines, containers) and/or Fog layer nodes. On the one hand, Fog applications are as diverse as the Internet of Things itself [4]. They perform several tasks as monitoring or analyzing data from network-connected things and devices, then initiate an action. Actions can involve machine-to-machine communications or human-machine interaction. Examples include locking/opening a window/door, reconfiguring an equipment (i.e. by changing settings), applying the brakes on a vehicle, opening a valve in response to a pressure threshold or sending an alert to a human operator to make a preventive/reactive intervention. The possibilities are numerous but have in common to be relatively low resource consuming yet low latency-sensitive. On the other hand, Cloud-based applications can be used by both IoT devices and Fog-based applications. Cloud applications

⋆This paper has been invited for submission to Journal of Systems Architecture (JSA) as an extended version from DETECT 2019 [15] @ MEDI 2019 workshops. https://www.irit.fr/MEDI2019/

*Corresponding author
khaled.khebbeb@univ-pau.fr (Khaled Khebbeb)
ORCID(s): 0000-0001-6675-0646 (Khaled Khebbeb)

perform intensive latency-insensitive calculation (ex. strategic planning with complex data processing, complex decision making, high workload peak absorbtion, etc.) and massive storage requirements (ex. huge database hosting and big data processing) including data sustainability solutions. Thus, Cloud and Fog are defacto complementary and have to coexist in the IoT landscape [18].

The IoT environment is of a highly dynamic nature [8], it requires the Fog layer to be ubiquitous, dynamic and smoothly scalable and reconfigurable (i.e., adaptable) to support services mobility across nodes and demand fluctuations [22]. Similarly, the Cloud layer is wanted to be highly scalable - and therefore elastic [10, 11]- to absorb the workload peaks (i.e., by providing more resource) as well as contractible when the workload drops (i.e., by freeing unneeded resource). Given the importantly dynamic nature of the IoT layer, which incarnate the environement pushing needs and requirments, both Fog and Cloud adaptation has to be of autonomic management (i.e., of minimal human intervention) and are therefore qualified as self-adaptable [12]. If Cloud elasticity (which implies self-adaptation) has been studied and explored for years [1] and considered as relatively mature, Fog self-adaptation in terms of node availability control (switching on/off), services distribution across nodes and the Fog layer temporal evolution are important research concerns that still need to be investigated, which makes it challenging to master Fog systems design at the present time. The main questions we tend to address are: (I) how to accurately design self-adaptation aspects in the Fog layer and (II) how to thoroughly express and ensure properties regarding Fog applications requirements and characteristics, while continuously evolving over time. Furthermore, we tend to study (III) how self-adaptive behaviors of both Cloud and Fog layers articulate (i.e., orchestrate) in the IoT landscape to answer the heterogeneous demands in terms of computing resource capability, service availability and low-latency sensitivity. Formal methods present the appropriate mechanisms to address these open issues. Based on mathematical concepts, they provide the required accuracy and rigor to express and ensure high-level qualitative specification of both Cloud and Fog self-adaptation and present a reliable solution for temporal properties study of their dynamic behaviors.

In this paper, we propose a solution to manage Cloud Fog self-adaptation and orchestration basing on centralized control pattern as described in [30]. Orchestrating Cloud and Fog is a key concept aiming at optimizing the use of resource pools available at both layers in order to accurately meet the underlying IoT requirements. The idea is first to specify self-adaptation at Cloud and Fog layers in terms of structure and behavior. This step's output is to identify a set of monitoring predicates and atomic adaptation actions. The predicates are to diagnose both layers' states in terms of resource provisioning (over/under provisioning). The actions are to identify adaptation mechanisms to apply such as replicating a service instance, migrating a service to a different Cloud server (virtual machine) or Fog node, and resizing Cloud VMs in terms of resource (processing, memory and

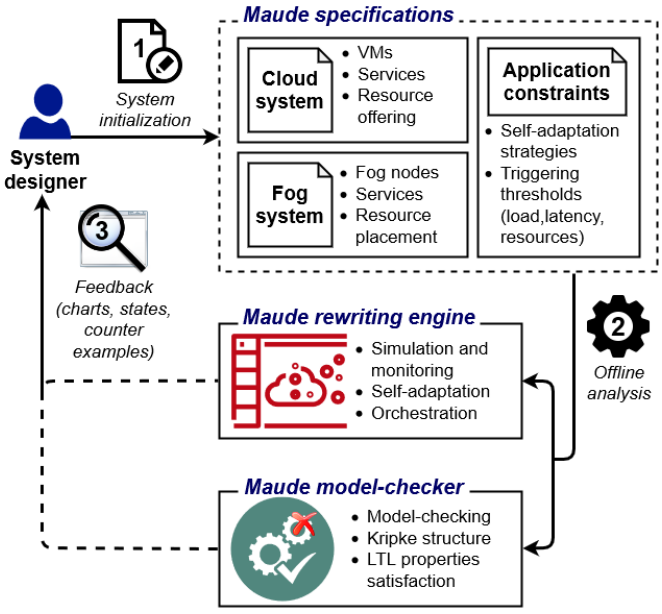

Figure 2: Our formal solution for the specification and analysis of Cloud-Fog self-adaptation and orchestration

bandwidth) offering, etc.. The second step is specifying a Cloud-Fog orchestrator which decides of the actions to be triggered in order to adapt at Cloud and/or Fog layers. The orchestrator considers the observed states (monitoring predicates) of both layers and then applies the proper sequence of actions (i.e., strategy) to achieve an adaptation at one or both layers, if the specified conditions are satisfied. Finally, the third step is identifying and designing a set of temporal properties to be satisfied to ensure the adaptations' qualitative correctness.

To achieve all these goals, we propose a formal modeling approach of self-adaptive Cloud and Fog orchestration based on rewriting logic [19] through the formal specification language called Maude and its associated tools including a model-checker for formal qualitative verification [6, 5]. We choose Maude for several reasons: (1) the language itself is expressive enough to model both Cloud and Fog layers in terms of structure which include sets of servers, nodes, services and resource allocation for each. (2) The Maude's underlying rewriting logic semantics are executable and allow designing structural reconfiguration (i.e., adaptation actions) with *correctness-by-definition* insurance. (3) The language and semantics support boolean expressions and first order logic which are relevant to design the monitoring predicates. Finally (4), Maude provides a model-checker which supports symbolic state-based verification of properties (by implementing a *Kripke* structure). The properties can be expressed with Linear Temporal Logic (LTL) which is relevant to study the managed Cloud-Fog environment temporal evolution in a qualitative point of view [2]. The main goal of our approach is finally to provide a formal design and implementation of the Cloud-Fog self-adaptive behaviors and express qualitative properties over these behaviors that can be formally verified.

Figure 2 summarizes the principal of our formal approach for the specification and analysis of Cloud-Fog self-adaptation and orchestration. The Maude-based specifications enable a system designer to model initial Cloud/Fog configurations and to express application constraints, respectively in terms of resource deployment (servers, resources and services for each layer) and self-adaptation strategies set-up (triggering conditions' threshold values for loads, resources and latency). Ultimately, the Maude system allows analyzing the specified behaviors under two aspects: simulation/monitoring and formal verification, respectively using the Maude rewriting engine and the Maude-built-in model-checker. Precisely, the designed system's state evolution can be simulated and monitored to witness the execution of the specified self-adaptation and orchestration strategies. In addition, the states evolution can be formally verified using a symbolic state-based model-checking technique relying on LTL as a temporal logic, and implementing a *Kripke* structure to consider symbolic high-level system states, thus overcoming the state explosion problem. Precisely, a *Kripke* structure allows expressing classes of equivalence to gather different structural states (i.e., configurations) under the same symbolic state with respect to logical predicates to be developed later.

The remainder of the paper is organized as follows. Section 2, introduces our model for Cloud-For orchestration and discusses its encoding into Maude's rewriting logic principles. Section 3 presents our Maude-based specification for Cloud and Fog layers in terms of structure and self-adaptive behavior. Section 4 gives the Cloud-Fog orchestrator's behaviors and adaptation triggering logic. It describes temporal qualitative properties using LTL and discusses formal verification using Maude's tools. Section 5 illustrates our solution of Cloud-Fog self-adaptation and orchestration through a smart city scenario case study. Section 6 discusses related wor, and finally, Section 7 concludes the paper and discusses future directions. In addition, we provide the Maude specification modules in Appendix A and we explain the defined LTL formulas in Appendix B.

2. A model for self-adaptive Cloud-Fog orchestration

The Fog extends the Cloud to be closer to the things that produce and act on IoT data. The Fog employs resources, called Fog nodes, that can be deployed anywhere with a network connection. Any device with computing power, storage capacity and network connectivity can be a Fog node. The main idea of the Fog is analyzing IoT data closer to where it is collected (i.e., at the edge of the IoT-Cloud continuum network) mainly to minimize the latency. The Fog offloads important amount of network traffic from the core network (i.e., the Cloud) and it keeps sensitive data inside the network (i.e. close to the IoT devices). On the other hand, the Cloud is highly used for computation-intensive and latency-insensitive activity. It provides massive quantity of computing resource gathered in distant facilities (hence the latency) called data-centers. The Cloud layer is a strong

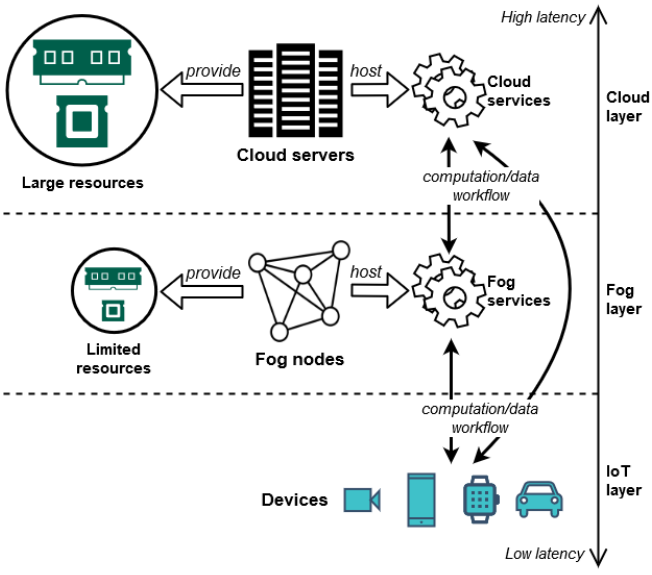

Figure 3: Services, resources and interactions in Cloud, Fog and IoT layers

reliable source of computing and storage power yet it fails answering low-latency requirements, whereas the Fog layer nodes are globally categorized with limited computing resource capabilities, possess relatively low storage and energy abilities but ensure low-latency computation. Both Cloud and Fog provide resource capabilities in terms of processing (CPU) and memory (RAM). Resources are available as pools hosted on both infrastructures (i.e., Cloud servers and Fog nodes) and are provided via (Cloud and Fog) services of different networking capabilities (i.e., bandwidth). Figure 3 shows how resources and services are provided on both Cloud and Fog layers, and how devices in the IoT layer interact with Cloud and/or Fog service instances to operate (i.e., by sending/receiving requests and data).

2.1. Cloud and Fog self-adaptation

The IoT layer is of a highly dynamic nature. It incarnates the environment pushing needs and requirements that continuously evolve over time. This requires the Fog layer to be ubiquitous, dynamic, smoothly scalable and reconfigurable (i.e., adaptable) to support services mobility across nodes, and demand fluctuations. Similarly, the Cloud layer is wanted to be highly scalable -and therefore adaptable- to absorb the workload peaks (i.e., by providing more resources) as well as contractible when the workload drops (i.e., by freeing unneeded resources). However, given how fast the demands may evolve, and how complex it might get to rapidly react to some situations by providing proper action plans, both Cloud and Fog adaptation need to be ensured in an automated way, thus qualified as self-adaptation. Generally, self-adaptation is ensured by autonomic management [12]. It consists of a controller (or more) monitoring a managed system to gather data describing its state (e.g., in terms of resource provisioning regarding the demand). This informa-

tion is then analyzed in search for anomalies in any kind. Finally, plans are provided to describe the required actions to be triggered in order to fix the detected anomalies, according to predefined preferences (i.e., strategies), and the process is reiterated from the monitoring task.

In the context of the Cloud-Fog ecosystem, we consider self-adaptation as the ability to cope with the evolving demands of the underlying IoT layer. The allocated resource to Cloud servers (VMs, Containers), Fog nodes and/or both layers' services must be scalable and dynamic to address the highly dynamic IoT requirements in terms of computing resources. This ability implies adaptation at service (software) and infrastructure (virualization and hosting) levels. Adaptation could be about providing more resource to maximize performance (when the demand rises), freeing unnecessary resource to minimize operating costs (whe the demand drops) as well as optimizing service resource placement for latency and performance requirements. In the Cloud, self-adaptation is ensured by elasticity: a property ensuring autonomic resource adjustemnt by (de)provisioning computing resource [10, 11]. In the Fog however, it is focused on service placement and mobility on Fog nodes mainly to ensure low-latency access and resource sufficiency [26] to process the IoT layer's demands. To control self-adaptation capabilities of both the Cloud and the Fog layers, we provide self-adaptation strategies to address various regulation problems within both layers respectively. The strategies are triggered when specified conditions are satisfied by executing the corresponding adaptation actions. The designed high-level strategies to be presented in this paper for both Cloud and Fog self-adaptation behaviors are described as follows:

Cloud self-adaptation strategies

- *Scale-out:* deploying additional resources (VMs and services) to cope with the growing demand.
- *Scale-in:* freeing unnecessarily provisioned VMs and services when the demand drops.
- *Scale-up:* adjusting VMs offering by allocating more computing resources.
- *Scale-down:* adjusting VMs offering by freeing unused computing resources.
- *Load-balancing:* redirecting requests across the deployed services to balance the system's load.
- *Service migration:* migrating services across the deployed VMs to optimize the resource utilization.

Fog self-adaptation strategies

- *Provisioning:* deploying additional resources (Fog nodes, service instances) to deal with growing demand.
- *Deprovisioning:* freeing unnecessarily deployed nodes and services when the demand drops.
- *Load-balancing:* redirecting requests across the deployed services to balance the system's load.
- *Service mobility:* (re)placing services across the available Fog nodes to optimize the resource utilization.

2.2. Orchestrating the Cloud-Fog self-adaptation

Orchestration [9, 21] is the automated configuration, coordination and management of computer systems and software. It consists of aligning the infrastructure, applications and data with the business requirements. The main purpose of orchestrating systems is to enable their directed actions towards common goals and objectives. As Cloud and the Fog coexist in order to serve common applications, they both need to self-adapt in a way to achieve their common goals and objectives. In this sense, orchestrating Cloud and Fog self-adaptation is about directing their respective self-adaptation behaviors towards ensuring decision making, based on shared monitoring data, in order to achieve common high-level goals and purposes. Precisely, meet application performance goals using minimized cost, maximize application performance within resource constraints, optimize application performance using low-latency and resource allocation trade-offs, and so on. To describe the orchestration of Cloud-Fog self-adaptation behaviors, we propose two orchestration strategies as follows:

Orchestration strategies

- *Offload:* relocating a Cloud service from a VM deployed on the Cloud to a Node deployed on the Fog to answer low-latency requirements.
- *Backup:* relocating a Fog service from a Fog node to a VM in the Cloud to free resource capability on the Fog while ensuring service continuity.

Cloud-Fog orchestration extends Cloud and Fog self-adaptation capabilities slightly further. Without orchestration, respective Cloud/Fog self-adaptation restricts reconfiguration on the Cloud/Fog to remain local: the Cloud(Fog) only adapts on the bounds of its visible/known environment, i.e., the Cloud(Fog) itself. On the other hand, orchestration provides both Cloud and Fog knowledge of each other, enabling richer possibilities for adaptation. Precisely, orchestration allows Cloud service migration to be made towards the Fog (*Offload* Cloud computation to the Fog) and vice versa (*Backup Fog computation to the Cloud*.

To enable the orchestration of Cloud-Fog self-adaption, we propose model based on autonomic control with centralized pattern, as described in [7, 31]. This model requires orchestrating both Cloud and Fog self-adaptation by monitoring data at both layers, thus gathering global knowledge on both layers, to trigger accurate decisions at each layer individually. Figure 4 shows our model of Cloud-Fog orchestration for autonomic self-adaptation behaviors. The proposed Cloud-Fog orchestrator is designed to operate as a self-adaptation controller for both Cloud and Fog layers at the same time. It is deployed on the Fog layer as a Fog node master to ensure low-latency requirements. The Cloud-Fog orchestrator decides of when, how and where to adapt by triggering the right actions at Cloud and/or Fog layers

if needed. Note that the orchestrator, as a software entity, might be subject to failures of any kind, due to hardware or software malfunctions, leading to a SPOF (single point of failure) scenario where Cloud-Fog orchestration is shut-off. To deal with issue, one can imagine monitoring the orchestrator's health state (response-time thresholds, data/packets reception acknowledgement, etc.) using other software entities to restore its activity if needed. Solutions for the orchestrator's fault-tolerance and resilience are to be developed in future work. The Cloud-Fog orchestrator behavior is summarized as follows:

Cloud-Fog orchestrator behavior

- Monitoring the Cloud and the Fog layers.
- Merging monitoring data of both layers.
- Controlling and orchestrating Cloud and Fog self-adaptation, basing on shared knowledge, by applying Cloud and/or Fog self-adaptation and orchestration strategies.

The main contributions we present in this work are: (1) a proposition to design Cloud/Fog layers including resources and services description, (2) a specification of self-adaptation logic (i.e., strategies) in the Cloud/Fog layers including monitoring predicates and adaptation actions, (3) a thorough expression of properties regarding Cloud/Fog applications requirements and characteristics, while continuously evolving over time and finally, a methodology to study (4) how self-adaptive behaviors of both Cloud and Fog layers articulate (i.e., orchestrate) in the IoT landscape to answer the heterogeneous demands in terms of computing resource capability, service availability and low-latency sensitivity, basing on the specified strategies. The main expected outcome is to describe and implement self-adaptation and orchestration strategies (i.e., how actions are triggered basing on logical expressions composing monitoring predicates), and ultimately to verify whether the system manifests or not the designed behaviors.

3. Maude-based modeling of the Cloud-Fog self-adaptation and orchestration

Providing executable formal specification and enabling the automated reasoning about their inherent properties and temporal evolution is not a trivial task. To design Cloud and Fog self-adaptation and orchestration, the first step is to (1) specify the Cloud-Fog environment structurally. Such specification requires identifying and defining all the elements that categorize the entire system. Precisely, we need to define resources and services, Cloud VMs and Fog nodes and all the relationships than link these elements (e.g., resources quantity offered by a VM/node, set of services hosted in a VM/node, etc.) in order to build the entire Cloud-Fog environment. Once the system structurally defined, the second step is to (2) enable reasoning over it by analyzing its attributes to diagnose its states. The idea is enable answering questions such as "*are there enough provisioned resources within a VM/node ?*", "*do Cloud/Fog services ensure a certain response time ?*", etc. in order to diagnose global states including *underprovisioning* (not enough resources), *overprovisioning* (too many resources) and so on [10, 16, 15]. As we intend to provide adaptation capabilities, the third step is to (3) define the adaptation mechanisms. It consists of describing in a generic way how a given system configuration (i.e., structure) may evolve to express a new configuration incarnating the desired adaptation (such as adding/removing resources/nodes and service/VM instances), while ensuring structural coherence and consistency. The fourth step is to (4) define high-level strategies to describe, implement and simulate the system's behavior in managing the Cloud/-Fog self-adaptations and their orchestration. In consists of defining the orchestrator's behavior by providing a logic to trigger the suitable adaptation actions in response of the observed states. Finally, the fifth step is to (5) provide a formal methodology to study and verify the defined behaviors' correctness while evolving over time. It consists of diagnosing whether the orchestrator manifests, or not, the designed self-adaptation and orchestration strategies, while detecting and reporting cases where the implemented behaviors deviate from the desired ones.

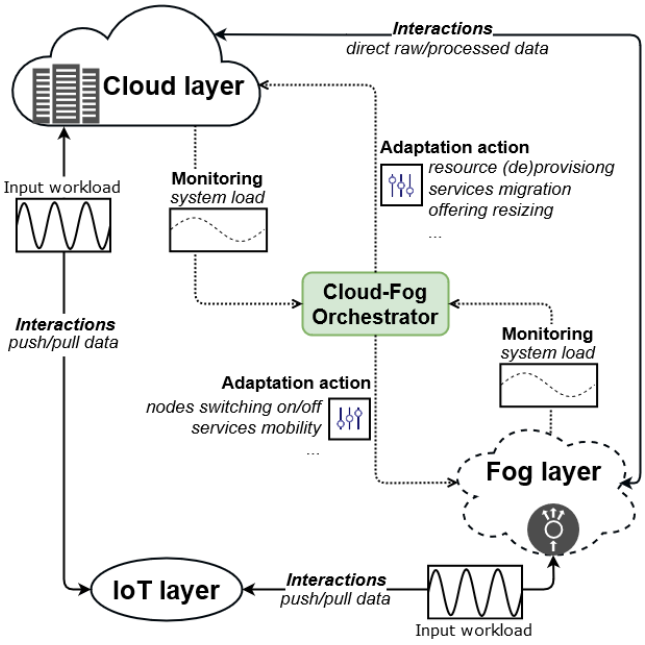

Figure 4: A model to orchestrate Cloud-Fog self-adaptation

To proceed with all the presented steps, we need to rely on a language which has the suitable expressiveness to capture the desired accuracy and complexity of specifications, together with enabling their executability. We also need reliable tools to perform formal verification and analysis of the designed behaviors' correctness. These considerations have lead us to rely on the Maude system, as an implementation of the rewriting logic, which satisfies perfectly all of our requirements. We choose Maude for several reasons. It answers each step requirements as follows : (1) the language

itself is expressive enough to model both Cloud and Fog layers in terms of structure which include sets of servers, nodes, services and resource allocation for each. (2) The language and semantics support boolean expressions and first order logic which are relevant to design monitoring predicates and therefore diagnose the system states. (3) The Maude's underlying rewriting logic semantics are executable and allow designing structural reconfiguration (i.e., adaptation actions) with *correctness-by-definition* insurance. (4) The language and rewriting engine enables designing rewrite rules of a conditional triggering nature. Finally (4), Maude provides a model-checker which supports symbolic state-based verification of properties (by implementing a *Kripke* structure). The properties can be expressed with Linear Temporal Logic (LTL) which is relevant to study the managed Cloud-Fog environment temporal evolution in a qualitative point of view. Maude [6, 5] is a high-level formal specification language based on rewriting and equational logic. A Maude program defines a logical theory and a Maude computation implements a logical deduction using axioms specified in the theory. A Maude specification is structured in two parts:

- A *functional module* which specifies a theory in *membership equational logic*: a pair $(\Sigma,\ E \cup A)$, where signature Σ specifies the type structure (sorts, subsorts, operators etc.). E is the collection of possibly conditional equations, and A is a collection of equational attributes for the operators (i.e., associative, commutative, etc.).

- A *system module* which specifies a rewrite theory as a triple $(\Sigma,\ E \cup A, R)$, where $(\Sigma,\ E \cup A)$ is the module's equational theory part, and R is a collection of possibly conditional rewrite rules.

To specify Cloud self-adaptation, we define a Maude functional module *CloudSpec* (in sub-section 3.2) to specify the Cloud structure (i.e., configuration) together with monitoring predicates and actions (setters and getters) to be applied over it (to diagnose states and apply reconfiguration actions). Similarly, to specify Fog self-adaptation, we define a functional module *FogSpec* (in sub-section 3.3) to describe the Fog layer structure, monitoring predicates and operations encoding different actions to reconfigure its structure. To specify the Cloud-Fog orchestrator's behaviors, we define a Maude system module *OrchControl* (in Section 4) to specify conditional rewrite rules expressing reconfiguration actions to be applied in order to orchestrate Cloud-Fog self-adaptations. We also define a Maude system module *Properties* which defines a *Kripke* structure to enable LTL-based model-checking. A *Kripke* structure is a model of temporal logic to represent the behavior of a system. It enables symbolic reasoning over system states which allows tackling the state explosion problem. Formal verification principles are to be developed in Section 4.2.

In previous work [15], we proposed a Maude-based specification for Cloud structures and self-adaptive (elastic) behaviors. We defined several strategies enabling horizontal scale elasticity, migration and load balancing at infrastructure and service levels of the Cloud layer. In this paper, we extend the previous specifications by considering vertical scale elasticity: we include computing resource (processing, memory, bandwidth) representation at infrastructure (VM instances) and application (service instances) levels. Furthermore, we provide monitoring predicates as well as operations (setters, getters) linked to resource consideration. As a part of the proposed extension, we model the Fog layer (cf. sub-section 3.3) in terms of structure and behavior. In addition, adaptation decisions are no longer decided at the Cloud layer. As explained before, the newly introduced Cloud-Fog orchestrator will encode the extended behaviors (to be developed in Section 4) to control both Cloud and Fog layers' adaptation. Also as part of the extension, we define a functional module *ServiceSpec* in sub-section 3.1 describing the application level of both Cloud and Fog layers.

Explaining the Maude specification modules: Constructing a rewriting logic based specification via the Maude language brings a considerable flexibility, extendability and reusability. Following a modular approach, a system designer can design, specify and implement their system in terms of structure (using functional modules), desired behaviors and properties (using system modules) with a highly structured methodology. Precisely, different part of the designed Cloud-Fog environment namely services, resources, VMs, Fog nodes, the Cloud and Fog layers including expression of their particular mechanisms, state predicates and properties can be separated across different modules. A module can be included in an other to reuse its contained specification. This allows easily extend and/or edit any module independently.

The presented extensions if this paper (comparing to the specifications presented in [15]) were possible simply by extending and editing the whole existing specification of Cloud structure including services, self-adaptation behaviors and properties. Service specifications were isolated in a separated module which is extended to define resources description. The Cloud specification module was extended to consider VMs' resources offering and their linked mechanisms and predicates. An additional functional module were provided to describe the Fog layer and its characteristics. Finally, system modules implementing the Cloud behaviors and properties were extended to consider both Cloud and Fog self-adaptation, orchestration and the temporal properties expressing the correctness of the designed behaviors.

Figure 5 gives a top view of our proposed modeling, execution and verification approach. We show the defined functional and system modules, their contents and how they lead to (I) enabling the designed behaviors' execution and to (II) verifying their correctness by model-checking. Direct links stand for the Maude-based encoding of the different specifications required to build our model. The dashed links stand for the *include* relationship (an arrow directed from A to B means that B includes A).

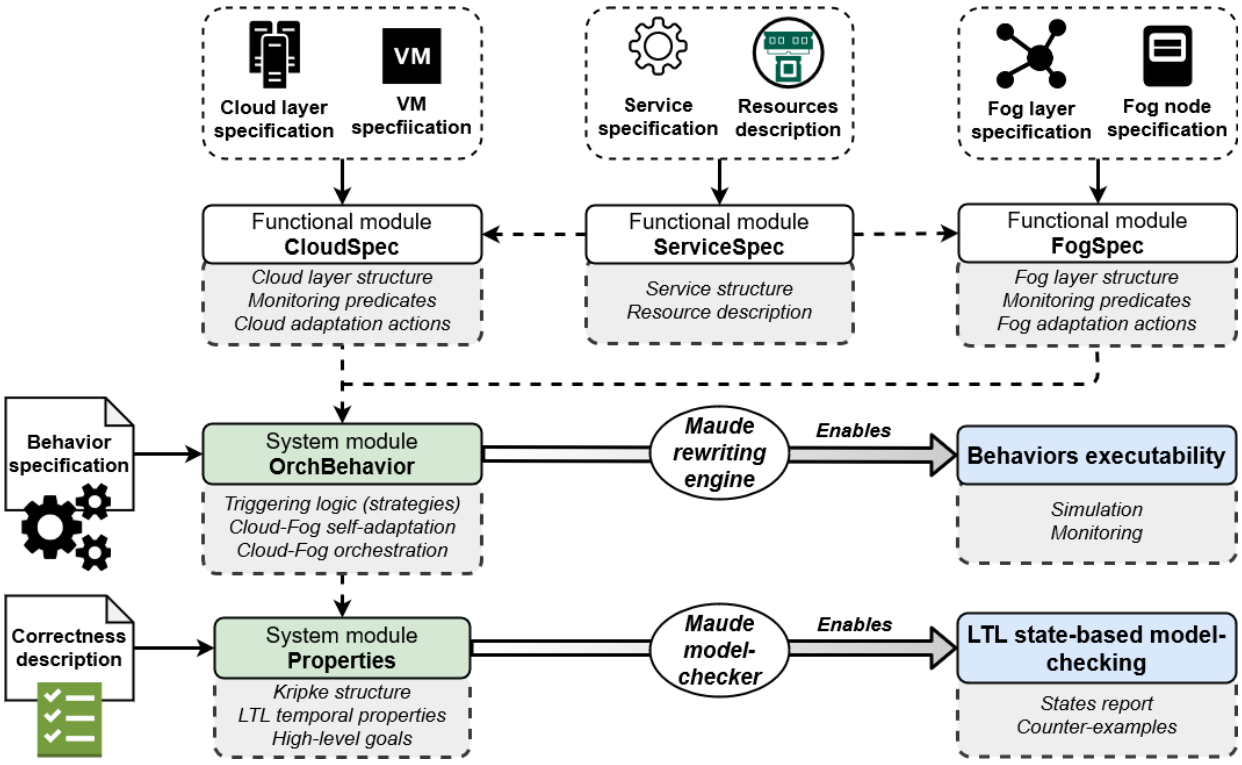

Figure 5: Top-view of our Maude-based modeling, implementation and verification approach for Cloud-Fog self-adaptation and orchestration

3.1. Modeling services and resources

Services represent the software applications deployed on the Fog and/or the Cloud layers. We consider services and resources as the central entities in our modeling. Services incarnate the entry point of the system: they receive the environments solicitations (i.e., requests) and their purpose is to provide access to the resources available at the Cloud/Fog layers. As services and resources are defined identically for both layers, we provide a single modeling specification, i.e., a Maude functional module *ServiceSpec* for services and resources in Listing 1 (shown in Appendix A). In our Maude-based modeling, a service is given as a sort `S` represented by a constructor (line 6):

```
S[max,rt,load:RES:state]
```

A resources description is given as a sort `RES` represented by a constructor (line 7):

```
-cpu,ram,bandwidth-
```

Note that constructors are given with operations that have the keyword `[ctor]` as parameter. For a service, the constructor highlights its current amount of handled requests (`load`), its current recorded response time (`rt`) an upper threshold in terms of latency (`max`), a state (sort `state`) to express its load state out of constructors `stable`, `overloaded`, `unused`, etc. (line 8) and a resource description (sort `RES`) expressing its requirements in terms of computing (CPU), memory (RAM) and networking (bandwidth). Service lists (sort `SL`) are construct recursively via the operation + which is associative and commutative (lines 9 – 11). Such representation allow considering sets of services to be hosted by a Cloud VM or a Fog node. In this functional module, we also define operations (lines 15 – 19) that return a service load, a service response time, resource requirements and that can compare two given resource specifications (i.e., >, < or =). Finally, we define monitoring predicates that return a given service state (line 13). A service is *overloaded* when its recorded response time exceeds its defined latency upper threshold, it is *unused* when its load (i.e. amount of requests) reaches zero and it is said to be *stable* when its load is above 0 and its response time is below its latency threshold.

3.2. Modeling the Cloud layer

The Cloud layer is an environment providing resources through a set of virtual machines (VMs) that host sets of services. The Cloud layer specifications in terms of structure constructors, monitoring predicates and reconfiguration adaptation actions are specified in the functional module *CloudSpec* given in Listing 2 (in Appendix A). Note that the previous specification including services, states and resource description are imported by inclusion (line 3). This inclusion allows reusing the previously defined specifications directly.

3.2.1. Cloud structures and monitoring predicates

A Cloud physical server and a virtual machine are defined as sorts CS and VM. A set of virtual machines is given by sort VML. Each sort is built according to its associated constructor (lines 6 – 11). A Cloud server is built by the constructor (line 7):

Table 1
Cloud monitoring predicates encoded into Maude

	Cloud monitoring predicate	Maude encoding
$\varphi 1$	a VM is overloaded	*EoverV(cs)*
$\varphi 2$	all VMs are overlaoded	*AoverV(cs)*
$\varphi 3$	a VM is unused	*EunV(cs)*
$\varphi 4$	a VM is underused	*EunderV(cs)*
$\varphi 5$	service migration is possible	*MIGpredC(cs)*
$\varphi 6$	a service is overloaded	*EoverCS(cs)*
$\varphi 7$	all services are overloaded	*AoverCS(cs)*
$\varphi 8$	a service in unused	*EunCS(cs)*
$\varphi 9$	request redirection is possible	*LBSpredC(cs)*
cs: the managed Cloud layer of sort *CS*		

Table 2
Cloud adaptation actions encoded into Maude

	Cloud adaptation actions	Maude encoding
$c1$	deploy new VM	*newV(cs)*
$c2$	destroy VM	*Vin(cs)*
$c3$	migrate service	*MIGc(cs)*
$c4$	deploy new service	*newCS(cs)*
$c5$	destroy service	*SinC(cs)*
$c6$	redirect request	*LBSc(cs)*
$c7$	add resource to VM	*scaleUpV(v)*
$c8$	remove resource from VM	*scaleDownV(v)*
cs: the managed Cloud layer of sort *CS* *v*: virtual machine of sort *VM*		

```
CS<x/VML>
```

A virtual machine is given by the constructor (line 8):

```
VM{y,SL:RES:state}
```

In a Cloud server specification, `x` encodes the Cloud upper hosting threshold in terms of VMs and `VML`, similarly to `SL`, is a list of VMs. For a virtual machine, `y` encodes its upper hosting threshold in terms of services. The term `RES`) expresses resources offering within a VM in terms of CPU, RAM and bandwidth. VM states are calculated similarly to services, i.e., proportionally to their load and upper hosting thresholds. A VM is underused if its allocated resources given by the getter operation (in line 21) are not fully used. It is overloaded if its load (i.e., amount of hosted services) exceeds its hosting upper threshold (`y`). The VM is unused if its load reaches zero. These values are given by operations giving information about the system such as the Cloud server's load, i.e., number of hosted VMs (line 19) and a VM's load, i.e., number of hosted services (line 20). To calculate states, we define a set of monitoring predicates (in first-order logic) which give information about the managed Cloud layer configuration. For instance, *AoverV()* is a predicate for "all VMs are overloaded" and *EunCS()* is a predicate for "there exists an unused cloud service instance" (lines 12 − 16). Table 1 gives the correspondence between Cloud monitoring predicates ($\varphi 1 - 9$) and their encoding into the *CloudSpec* functional module.

3.2.2. Cloud adaptation actions

In the functional module (i.e., Listing 2) adaptation actions are defined as rewrite operations which operate over the Cloud layer sort terms (lines 23 − 26). The different actions express atomic adaptation behaviors for horizontal scale elasticity (i.e., Cloud services deployment, migration and replication and VM replication), for load balancing (i.e., request redirection) and for vertical scale elasticity (i.e., VMs resource resizing). For instance, *newV()/Vin()* are used to deploy or remove a VM, *MIG()/LBS()* are for migration and load balancing and *scaleUpV(), scaleDownV()* are for adding/removing resource to/from a VM. Table 2 gives the correspondence between the possible Cloud adaptation actions ($c1 - 8$) and their encoding into Maude.

3.3. Modeling the Fog layer

The Fog layer is structurally similar to the Cloud layer. The Fog environment provides resources through a set of Fog nodes that host sets of services. In this sub-section, we define the Fog layer structure and adaptation actions in the functional module *FogSpec* (cf. Listing 3 in Appendix A). Similarly to the previously provided specification, we reuse the services specification and resources description (sorts `S`, `SL`, `RES` and `state`) from the previous specification in Listing 1 by inclusion (line 3). In addition, we define Fog structure constructors, monitoring predicates, access operations and atomic adaptation (reconfiguration) actions.

3.3.1. Fog structures and monitoring predicates

The Fog layer and a Fog node are defined as sorts FS and N. A set of nodes is given by the term NL. Each sort is built with its associated constructor (lines 6 − 8) to exhibit its configuration. Similarly to the Cloud layer structure, the Fog layer (expressed as cluster of Fog nodes) is built by the constructor (line 6):

```
FS<x/NL>
```

A Fog node is given by the constructor (line 8):

```
N{y,SL:RES:state}
```

In a Fog cluster, the term `x` gives the upper hosting threshold in terms of nodes and `NL` is a list of nodes. For a Fog node, `y` encodes its upper hosting threshold in terms of services (sort `SL`). The term `RES` expresses available resources within a node in terms of CPU, RAM and bandwidth. Nodes states, similarly to VM, are calculated proportionally to their load and upper hosting thresholds. A node is underused if its allocated resources given by the getter operation in line 20 are not fully used. It is overloaded if its load (i.e., amount of hosted services) exceeds its hosting upper threshold (`y`). The node is unused if its load reaches zero. These values are given by operations giving information about the system such as the Fog layer's load, i.e., number of online nodes (line 18) and a node's load, i.e., number of hosted services (line 19). To calculate states, we define a set of monitoring predicates that give information about the managed Fog layer configuration. For instance, *AoverN()* is a predicate for

Table 3
Fog monitoring predicates encoded into Maude

	Fog monitoring predicate	Maude encoding
$\varphi 10$	a node is overloaded	*EoverN(fs)*
$\varphi 11$	all nodes are overlaoded	*AoverN(fs)*
$\varphi 12$	a node is unused	*EunN(fs)*
$\varphi 13$	a node is underused	*EunderN(fs)*
$\varphi 14$	service relocation is possible	*MIGpredF(fs)*
$\varphi 15$	a service is overloaded	*EoverFS(fs)*
$\varphi 16$	all services are overloaded	*AoverFS(fs)*
$\varphi 17$	a service in unused	*EunFS(fs)*
$\varphi 18$	request redirection is possible	*LBSpredF(fs)*
fs: the managed Fog layer of sort *FS*		

Table 4
Fog adaptation actions encoded into Maude

	Fog adaptation actions	Maude encoding
$f1$	turn-on node	*onN(fs)*
$f2$	turn-off node	*offN(fs)*
$f3$	migrate service	*MIGf(fs)*
$f4$	deploy new service	*newFS(fs)*
$f5$	destroy service	*SinF(fs)*
$f6$	redirect request	*LBSf(fs)*
fs: the managed Fog layer of sort *FS*		

"all nodes are overloaded" and *EunFS()* is a predicate for "there exists an unused fog service" (lines $12-15$). Table 3 gives the correspondence between Fog monitoring predicates ($\varphi 10-18$) and their encoding into the *FogSpec* functional module.

3.3.2. Fog adaptation actions

Fog adaptation actions in the functional module *FogSpec* (i.e., Listing 3) are defined as rewrite operations which operate over the Fog layer sort terms (lines $22-23$). The different actions globally express atomic adaptation behaviors for Fog services mobility (relocation) and replication and nodes switching on/off. For instance, *onN()/offN()* are used to switch on/off a node and *MIGf()/LBSf()* are for service mobility and requests redirection. Table 4 gives the correspondence between the possible Fog adaptation actions ($f1-6$) and their encoding into Maude.

4. Implementing and verifying Cloud-Fog self-adaptation and orchestration

This section describes the Cloud-Fog orchestrator's behaviors presented in Section 2. Basing on the proposed specifications for services, Cloud and Fog layers in Section 3, the idea is to feed the orchestrator with the different monitoring predicates thus producing global knowledge about the entire Cloud-Fog environment. From the periodically collected observations, the goal is to perform local complementary adaptations (at Cloud and/or Fog) layers in order to achieve a high-level adaptation, i.e., following the introduced strategies in Section 2. In sub-section 4.1, we give the Maude-based specification of the Cloud-Fog orchestrator's behavior. We give conditional rewrite rules that express guarded reconfiguration implementing the presented strategies. Precisely to define a logic describing when and how the previously identified atomic actions are triggered. In sub-section 4.2, we express temporal properties of the specified behaviors using the *linear temporal logic* (LTL) and discuss their verification capabilities using the Maude-associated model-checker.

4.1. The Cloud-Fog orchestrator's behavior

To define the Cloud-Fog orchestrator's behaviors, we propose a set of conditional rewrite rules in a Maude system module *OrchBehavior* in Listing 4 (in Appendix A). A conditional rewrite rule is given as follows:

$$crl[R] : term => term' if (condition).$$

A rule (`crl`) named `R` describes how the previously defined actions are triggered. It rewrites the left-hand side `term` of the rule into its right-hand side `term'` on which an action is applied. A conditional rewrite rule is triggered when its specified `condition` is satisfied. Conditions are expressed as disjunctions and/or conjuctions of the previously defined monitoring predicates for Cloud, Fog layers and services. Note that the previous functional modules are imported by inclusion (line 3). To allow reasoning on the entire Cloud-Fog environment, we define a sort `ENV` describing the orchestrated Cloud-Fog environment through the constructor `cloud || fog` (line 6). This representation allows applying monitoring predicates and conducting rewrites (i.e., reconfiguration) on both layers separately or simultaneously.

The previously defined actions' conditional triggering is described for Cloud layer in lines $14-37$ and for Fog layer in lines $38-56$. To improve the readability, conditional rewrite rules hold the same action labels as shown before. Rewrite rules specific to Cloud are named ci (cf. Table 2) and fi for the Fog (cf. Table 4). The symbol i refer to a rule's number. Variables of different sorts are declared in lines $7-13$ for symbolic interpretation of elements in the rewrite rules. Notice that adding service instances, VMs and switching on Nodes have two different implementations with suffixes *Low* and *Hi*. This modeling is directly inspired from previous work [14] where we define different models for low and high resource availability. We apply this approach for Cloud servers and services as well as for Fog nodes and services provisioning. To implement Cloud-Fog orchestration actions, we define two conditional rewrite rules (lines $57-76$):

- *Offload*: this rule named `o1` offloads the Cloud layer by relocating a service instance into the Fog layer. It consists of migrating a service from a VM in the Cloud to a node in the Fog. Migration is made when the initial VM host is overloaded or fails ensuring the service's requirements, and if the potential Fog node host carries enough resource to answer the service's requirements. This condition will be referred to as $\varphi 19$.

- *Backup*: the rule o2 allows using the Cloud layer as a backup solution if the Fog layer fails meeting resource requirements. It allows migrating a service instance from a Fog node to a Cloud VM if all nodes are overloaded and if the VM provides enough resource to ensure the service's requirements. This condition will be referred to as $\varphi 20$.

As part of our modeling effort, notice that we designed the rewrite rules to be complementary and composable. Precisely, the rules' triggering conditions are pretty much exclusive. It means that for a given set of monitoring predicates which are satisfied, the rules that can be triggered will not result in contradictory actions. For example, load-balancing is applied when there is an unused instance service and an overloaded service instance. But for load-balancing to be ever possible, it requires that (1) creating a new service instance is not possible if there is an unused service instance and that (2) an unused service instance cannot be deleted if there is an overloaded one. This modeling approach enables producing action plans that can draw a high-level adaptation (i.e., a strategy). In the next sub-section, we explain our formal verification approach to study whether such high-level strategies are indeed ensured.

4.2. Formal verification of the orchestrator's behavioral correctness

Formal verification consists of ensuring the defined behaviors correctness. Precisely, it consists of verifying the introduced Cloud-Fog orchestrator's ability to manifest the described strategies. To proceed, we propose a LTL state-based model-checking technique. The first difficulty here is controlling the set of possible system states (i.e., configurations). The proposed Maude specification allows modeling any configuration in a Cloud/Fog environment, which results in a potentially infinite set of structural system states. Thus, we use a Kripke structure to identify symbolic states to manage the set of states complexity and to tackle the state explosion problem. Symbolic state allows reasoning on a system configuration by focusing only on the predicates (i.e., the specified monitoring predicates $\varphi 1 - 20$) that this configuration satisfies. Then we define a set of LTL propositional formulas to describe the desired transitions between those states. The LTL formulas express the expected orchestrator's high-level behaviors (i.e., strategies) to be achieved for both Cloud and Fog layers. They describe the system's temporal evolution basing on the system's symbolic states evolution over time.

4.2.1. Specifying the Cloud-Fog orchestrator's behavior

A *Kripke* structure is a calculus model of temporal logic [2] which allows to identify symbolic states and define desired transitions between them. Given a set AP of atomic propositions, a Kripke structure is formally defined [5] as a triple $\mathbf{A} = (A, \rightarrow_{\mathbf{A}}, L)$, where A is a set of symbolic states, $\rightarrow_{\mathbf{A}}$ is a transition relation, and $L : A \rightarrow AP$ is a labeling function associating to each state $a \in A$, a set $L(a)$ of atomic propositions φi in AP that hold in a. $LTL(AP)$ denotes the formulas of the propositional linear temporal logic. The semantics of $LTL(AP)$ is defined by a satisfaction relation: $\mathbf{A}, a \vDash \Phi$, where $\Phi \in LTL(AP)$.

The set of symbolic states A express classes of equivalence gathering numerous structural configurations with respect to the defined state predicates [17], i.e., $\varphi 1 - 9$ for the Cloud layer, $\varphi 10 - 18$ for the Fog layer and $\varphi 19 - 20$ for the Cloud-Fog environment. Notice that different Cloud(Fog) configurations can have the same state in terms of resources offering, VMs(nodes) state of under/overprovisioning or global state of load-balancing at application/infrastructure levels. In other terms, a symbolic state allows reasoning on a system configuration by focusing only on the predicates that this configuration satisfies. For example, consider two Cloud configurations *C1* and *C2*. *C1* hosts 3 overloaded VMs and *C2* hosts 20 overloaded VMs. *C1* and *C2* express two different structural configurations. Nevertheless, both are considered of the same symbolic state as both satisfy the system monitoring predicate $\varphi 1$ expressing "*all VMs are overloaded*". To avoid using additional identifiers for the symbolic states tags regarding the atomic proposition in AP (i.e., monitoring predicates φi) and to ease the understanding of the paper, by abuse of language we will consider that $A = AP$ meaning that a symbolic state a_i corresponds to a satisfied monitoring predicate φi or $ai \equiv \varphi i$.

Now that symbolic states are identified, we provide the set of *Linear Temporal Logic* (LTL) propositional formulas $LTL(AP)$ to express the self-adaptation and orchestration strategies for Cloud and Fog layers. LTL as an analytic support is particularly powerful. It is expressive enough to accurately describe (in a declarative fashion) a system's temporal evolution in terms of system states evolution. It is also generic enough to describe desired high-level goals, i.e., strategies. Our defined formulas globally express the *liveness* fundamental property (i.e. the insurance that a given state is reachable) which can be interpreted as a qualitative indicator of behavioral correctness. Each formula describes the Cloud-Fog orchestrator's high level behaviors basing on the global environment temporal evolution. The evolution is expressed in terms of time traces focusing on system states. The states are expressed with the previously introduced φi predicates. Table 5 lists the main LTL symbols and operators that will be used in this paper. Further detail about LTL syntax and semantics can be found in [23].

We give $LTL(AP)$ as three sets of LTL formulas P_C, P_F, P_O describing Cloud, Fog self-adaptation and orchestration properties respectively as follows:

$$P_C = \{ScaleOut_C^{\{VM,S\}}, ScaleIn_C^{\{VM,S\}}, ScaleUP_C, ScaleDown_C, LoadBalance_C, Migrate_C\}$$

$$P_F = \{Provision_F^{\{N,S\}}, Deprovision_F^{\{N,S\}}, LoadBalance_F, Mobility_F\}$$

$$P_O = \{Offload_C, Backup_F\}$$

The proposed LTL property formulas are defined as follows and are explained in Appendix B:

$$ScaleOut_C^{VM} \equiv \square\big[(\varphi 2 \vee (\varphi 1 \wedge \neg\varphi 3)) \rightarrow \Diamond\neg\varphi 2\big] \quad (1)$$

$$ScaleOut_C^{S} \equiv \square\big[(\varphi 7 \vee (\varphi 6 \wedge \neg\varphi 8)) \rightarrow \Diamond\neg\varphi 7\big] \quad (2)$$

$$ScaleIn_C^{VM} \equiv \square\big[((\varphi 3 \vee \varphi 4) \wedge \neg\varphi 1) \rightarrow \Diamond\neg\varphi 3\big] \quad (3)$$

$$ScaleIn_C^{S} \equiv \square\big[(\varphi 8 \wedge \neg\varphi 6) \rightarrow \Diamond\neg\varphi 8\big] \quad (4)$$

$$ScaleUP_C \equiv \square\big[\varphi 2 \vee (\varphi 1 \wedge \neg(\varphi 3 \vee \varphi 4)) \rightarrow \Diamond\neg\varphi 1\big] \quad (5)$$

$$ScaleDown_C \equiv \square\big[(\varphi 4 \wedge \neg\varphi 1) \rightarrow \Diamond\neg\varphi 4\big] \quad (6)$$

$$LoadBalance_C \equiv \square\big[((\varphi 6 \wedge \varphi 8) \rightarrow \bigcirc\varphi 9)\ \mathcal{U}\ \neg\varphi 6\big] \quad (7)$$

$$Migrate_C \equiv \square\big[(\varphi 1 \wedge (\varphi 3 \vee \varphi 4) \rightarrow \bigcirc\varphi 5)\ \mathcal{U}\ \neg\varphi 1\big] \quad (8)$$

$$Provision_F^{N} \equiv \square\big[(\varphi 11 \vee (\varphi 10 \wedge \neg\varphi 12)) \rightarrow \Diamond\neg\varphi 11\big] \quad (9)$$

$$Provision_F^{S} \equiv \square\big[(\varphi 16 \vee (\varphi 15 \wedge \neg\varphi 17)) \rightarrow \Diamond\neg\varphi 16\big] \quad (10)$$

$$Deprovision_F^{N} \equiv \square\big[((\varphi 12 \vee \varphi 13) \wedge \neg\varphi 10) \rightarrow \Diamond\neg\varphi 12\big] \quad (11)$$

$$Deprovision_F^{S} \equiv \square\big[(\varphi 17 \wedge \neg\varphi 15) \rightarrow \Diamond\neg\varphi 17\big] \quad (12)$$

$$LoadBalance_F \equiv \square\big[((\varphi 15 \wedge \varphi 17) \rightarrow \bigcirc\varphi 18)\ \mathcal{U}\ \neg\varphi 15\big] \quad (13)$$

$$Mobility_F \equiv \square\big[(\varphi 11 \wedge (\varphi 12 \vee \varphi 13) \rightarrow \bigcirc\varphi 14)\ \mathcal{U}\ \neg\varphi 10\big] \quad (14)$$

$$Offload_C \equiv \square\big[(\varphi 1 \wedge (\varphi 12 \vee \varphi 13) \rightarrow \bigcirc\varphi 19)\ \mathcal{U}\ (\neg\varphi 1 \vee \varphi 11)\big] \quad (15)$$

$$Backup_F \equiv \square\big[((\varphi 11 \wedge \neg\varphi 2) \rightarrow \bigcirc\varphi 20)\ \mathcal{U}\ (\neg\varphi 11 \vee \varphi 2)\big] \quad (16)$$

In the *Linear Temporal Logic* semantics, a formula expresses a property as a temporal description in terms of states evolution. Precisely, it describes how a given system is expected to evolve over time to achieve a distinct behavior using the suitable temporal operators. During a system's execution, a LTL formula (describing a desired behavior) tells if the analyzed system manifests or not that particular behavior. A LTL property (formula) is ensured or satisfied if there exists a path within the system's execution, from a given initial state, which follows the specified temporal description [23]. In the context of self-adaptive systems, a system is subject to solicitations and changes that affect its state. The self-adaptation mechanisms allow such system to cope with changes by triggering actions in any kind. In our model for Cloud/Fog self-adaptation and orchestration, the proposed LTL formulas allow describing the desirable system behaviors over time. Precisely, each LTL formula expresses a sequence of system states describing its temporal evolution, and finally, the satisfaction of such sequence indicates the corresponding strategy satisfaction, or, in other words, the qualitative correctness of the designed behaviors.

Table 5
used LTL operators and symbols

LTL operator/symbol	Meaning
$\wedge$	conjunction / and
$\vee$	disjunction / or
$\rightarrow$	implies
$\neg$	negation / not
$\square$	globally / always
$\Diamond$	eventually or "in the future"
$\bigcirc$	next time
$\mathcal{U}$	until

The specified properties allow verifying if the designed strategies (for Cloud/Fog self-adaptation and Cloud-Fog orchestration) are indeed applied during the system execution as follows:

- In the Cloud layer (i.e., P_C properties), *ScaleOut/ In* refer to horizontal scale elasticity: where instances of services and/or VMs are provisioned/ deleted. *ScaleUp/ Down* refer to vertical scale elasticity: where VMs are resized in terms of resources (by adding/ removing computing resources). *Migrate* refers to migration mechanisms where service instances are moved across VMs and *LoadBalance* refers to redirecting requests in order to equilibrate service instances' load [13, 16].
- In the Fog layer (i.e., P_F properties), *Provision/ Deprovision* refer to switching on/off Fog nodes and/or adding/ removing service instances. *LoadBalance* is about requests redirecting and *Mobility* refers to Fog services mobility (relocation) across Fog nodes.
- The orchestration behaviors of both layers (P_O properties), *Offload* refers to moving a service instance from the Cloud layer to the Fog layer and *Backup* refers to moving a service instance from the Fog layer to the Cloud.

At this point of our presented contributions, the provided rewriting logic Maude-based specification implement the Cloud Fog orchestrator's behaviors and the introduced LTL formulas describe their qualitative correctness. Now we turn our attention towards the executable automated analysis and the formal verification of these behaviors to check whether they are ensured or not in our proposed implementation.

4.2.2. Verifying the orchestrator behavior's correctness

Maude comes with a built-in model-checker that fully supports the *Kripke* and LTL semantics. The model-checker is used to conduct state-based verification [2] of the system's execution, basing on the defined LTL formulas. Precisely, the model-checker verifies the satisfaction of the defined LTL property formulas during the system's execution.

To enable the Maude model-checker reasoning over the designed Cloud-Fog self-adaptation and orchestration behaviors modulo the specified LTL property formulas (expressing strategies), it has to be configured by defining a system module for properties definition. Such a module directly implements the *Kripke* structure symbolic states and encodes the LTL property formulas into the Maude language [6, 15].

In the module *Properties* (listing 5) shown in Appendix A, we give the principles of encoding the *Kripke* labeling function of symbolic states using conditional equations and we directly encode the introduced LTL formulas with equations. Precisely, from a given Cloud/Fog configuration (sorts `CS`/`FS` as subsorts of the generic sort `state` in line 4), we label (using the symbol (`|=`) a configuration symbolically to a proposition *pi* if the associated predicate φi (thus its Maude encoding predicate as shown in Tables 1 and 3) is true or satisfied (lines $16-19$). In addition, the main used LTL operators (as shown in Table 5) are directly encoded into Maude as follows (lines $26-29$) : the conjunction operator $\wedge$ is encoded as `/\`, disjunction $\vee$ is encoded as `\/`, the negation $\neg$ is encoded as `~`, the implication operator $\rightarrow$ is encoded as `->`, the henceforth operator $\Box$ is encoded as `[]`, eventually $\Diamond$ is encoded as `<>`, the next state operator $\bigcirc$ is encoded as `O` and finally, the until operator $\mathcal{U}$ is encoded as `U`.

As inputs, a system designer gives the model-checker an initial state -which is expressed structurally using the previously defined constructors- and a LTL property formula $\Phi \in P_C \cup P_F \cup P_O$ to be checked from that state. As outputs, the model-checker shows *True* if the property is ensured during the system's execution. Otherwise, it prints a counter-example showing the execution path that has lead to the violation of the property. Note that the execution path is shown as a succession of the triggered rewrite rules (among the defined rules in Section 4.1).

Graph based representation of the system behavior: A *Kripke* structure models the desired behavior of a system and can be seen as a graph, or more accurately as a *Labeled Transition System* (LTS) [25] where nodes represent the symbolic states of the system and where edges represent state transitions and events. To provide such representation, we give the LTS based modeling of the system's behavior from the orchestrator point of view, being the principal controller in our modeling proposition. Precisely, we give in Figure 6 the Cloud layer based vision of the system's behavior, and in Figure 7, the Fog layer based vision. In terms of states, notice that the specified monitoring predicates express only undesirable system states. Thus, in the absence of satisfied predicates $\in \varphi 1-20$, the monitored Cloud (or Fog) system is said to be in a *stable* symbolic state (i.e., where no adaptation is needed) which incarnates our main desirable state for both Cloud/Fog layers. The transitions are categorized with the implemented rewrite rules describing the orchestrator's behavior (i.e., $c1-8$, $f1-6$ and $o1-2$). As requests traffic determines the system's state evolution, events stand for the requests coming and exiting the system (input/output) shown as *in*/*out* transitions.

To simplify the visual representation of the system behavior giving its complexity and to ease the readability of the proposed implementation, we have gathered the different symbolic states categorized by the predicates $\varphi 1-20$ into states of a higher level of abstraction.

In the provided representations, we focus on the high-level desirable and undesirable states in terms of resources provisioning. More accurately, we focus for both layers on the state of *Underprovisioning* which requires provisioning more resources (via the *Scale-Out/ Up* Cloud strategies and the *Provisioning* Fog strategy), the states of *Overprovisioning* which requires freeing the unused ones (via the *Scale-In/ Down* Cloud strategies and the *Deprovision* Fog strategy), the state of *Unbalancing* which requires equilibrating the system's load at application or infrastructure levels (via the *Migration/ Load-Balance* Cloud strategies and the *Mobility/ Load-Balance* Fog strategies), the state of *Orchestration* which requires triggering the proposed orchestration mechanisms (using the Cloud-Fog *Offload/ Backup* strategies) and finally, the desired *Stable* state where no particular self-adaption nor orchestration strategy is required as no monitoring predicate is satisfied.

For Cloud layer based vision of the System, the higher-level states are given with $C_{st} = \{Sc, Uc, Oc, Bc, Orch\}$. Each higher-level state (excluding $Orch$) is categorized by the satisfaction of at least one Cloud-specific monitoring predicates $\varphi 1-9$ as follows:

- *Stable*: $Sc \equiv \emptyset$
- *Underprovisioning*: $Uc \equiv \{\varphi 1, \varphi 2, \varphi 6, \varphi 7\}$
- *Overprovisioning*: $Oc \equiv \{\varphi 3, \varphi 4, \varphi 8\}$
- *Unbalancing*: $Bc \equiv \{\varphi 5, \varphi 9\}$

Similarly, the higher-level states for the Fog layer based vision are given with $F_{st} = \{Sf, Uf, Of, Bf, Orch\}$ where each state (excluding $Orch$) is categorized by the satisfaction of at least one Fog-specification monitoring predicates $\varphi 10-18$ as follows:

- *Stable*: $Sf \equiv \emptyset$
- *Underprovisioning*: $Uf \equiv \{\varphi 10, \varphi 11, \varphi 15, \varphi 16\}$
- *Overprovisioning*: $Of \equiv \{\varphi 12, \varphi 13, \varphi 17\}$
- *Unbalancing*: $Bf \equiv \{\varphi 14, \varphi 18\}$

Finally, the state $Orch$ is common to both the Cloud and Fog layers as it is categorized by monitoring both layers, i.e., by the satisfaction of at least one of the Cloud-Fog monitoring predicates $\varphi 19-20$ as follows:

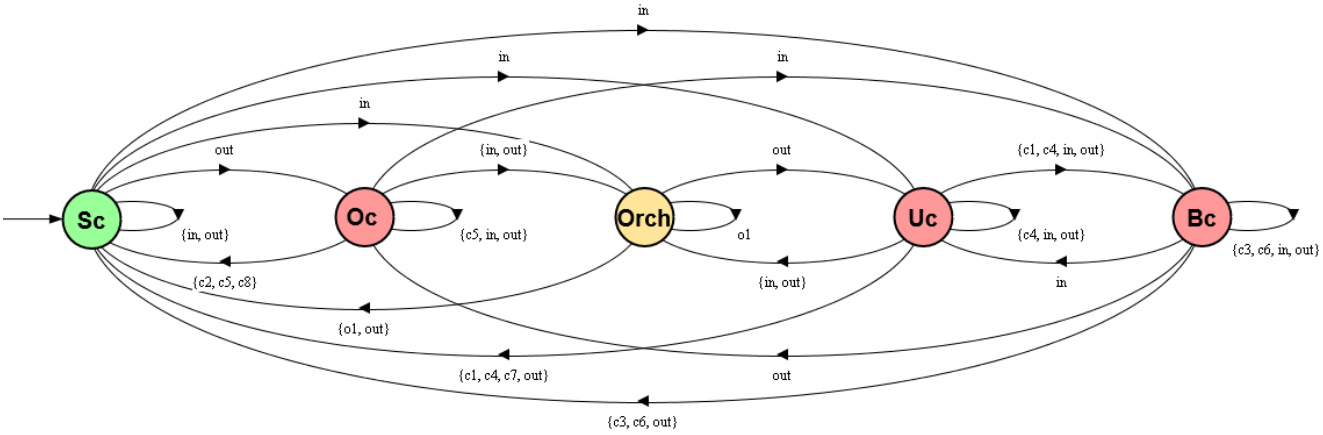

Figure 6: LTS-based representation of the Cloud layer states and transitions from the orchestrator's perspective

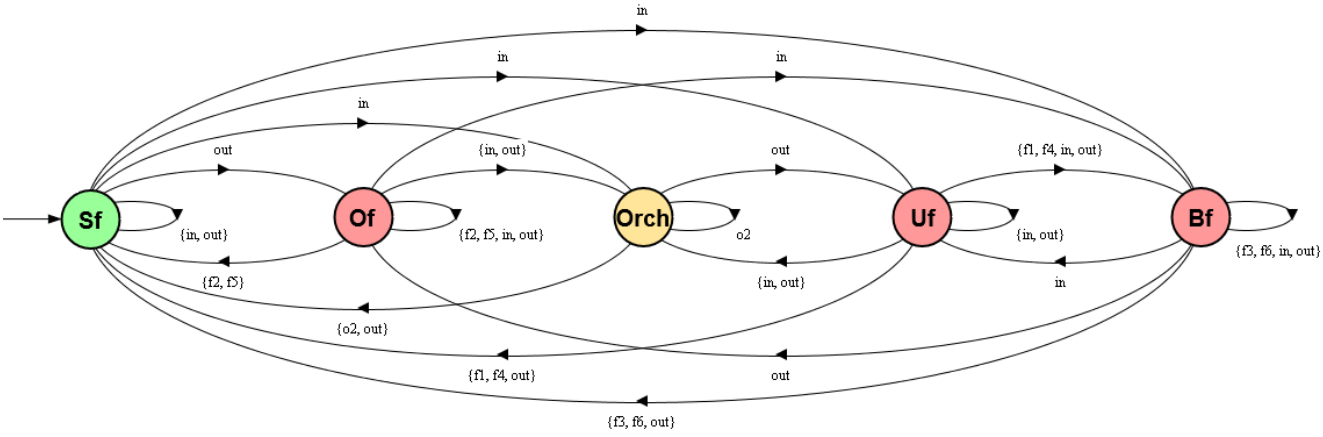

Figure 7: LTS-based representation of the Fog layer states and transitions from the orchestrator's perspective

- *Orchestration*: $Orch \equiv \{\varphi 19, \varphi 20\}$

In the shown representations, the stable state for both Cloud/Fog systems is initial, but any state can be the initial one as it is determined by monitoring. This shows the ability of the orchestrator into controlling the Cloud and/or Fog layer self-adaptation and orchestration strategies towards reaching, returning or converging towards their respective *stable* and desirable state.

5. Case study: a smart city scenario

To illustrate our solution for the Cloud/Fog self-adaptation and orchestration, we show how the introduced modeling and analysis approach can be applied through a case study in a Cloud/Fog-based smart city scenario.

5.1. The studied Cloud-Fog system

Consider a smart city application for security and traffic monitoring analysis as shown in Figure 8. The application is deployed on a highway parcel crossing two countries or states border. It consists of a video processing system designed as a set of micro-services distributed across different Fog nodes and a service deployed on the Cloud layer. On the Fog layer, the application deploys a service (S1) which records vehicles' speed and frequency, and a service (S2) recording license plates. Different surveillance cameras are deployed on the border crossing and speed sensors are distributed across the highway. The purpose of such a system is to analyze the traffic's fluidity through the service S1, and to analyze license plates to report stolen vehicles through the service S2. The speed sensors collect the number of vehicles and their speed then transmits data to S1. The cameras take pictures of the license plates then transmit data to S2 for pre-processing (e.g. tagging the pictures of contextual data such as location, date, etc.). The service S2 is linked to a service (S3) which is deployed on a dedicated server (VM) in the Cloud layer. The service S3 performs image processing to extract the textual form of the license plates. It then interacts with a Cloud-based database in search for stolen vehicles from the received license plates and reports matches to the authorities. We consider that both S1 and S2 require moderate amount of resource to operate while S3 needs consequent resource capacity. In terms of locality deployment, the Cloud layer consists of a VM deployed on a distant datacenter, and the Fog layer consists of back-end servers (Fog nodes) deployed on a nearby facility (such as a border office).

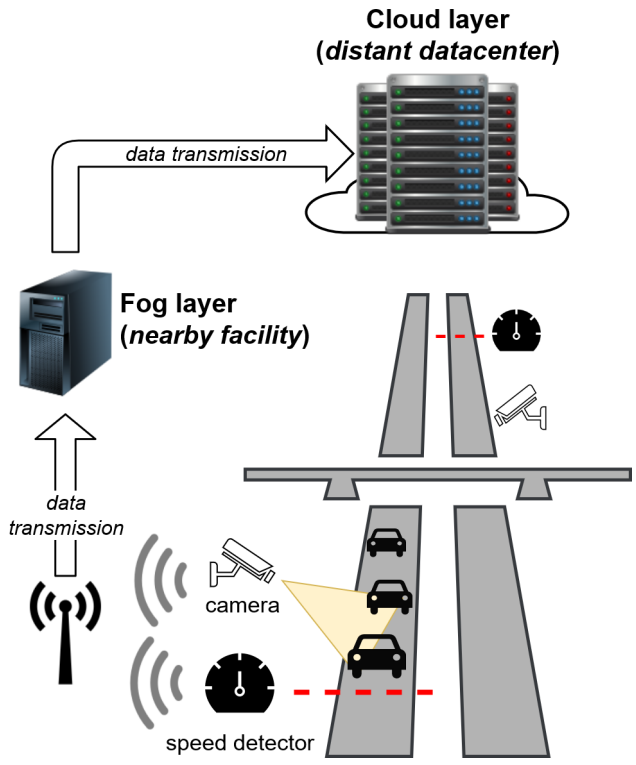

Figure 8: Highway traffic analysis: a Cloud/Fog-based smart city case study

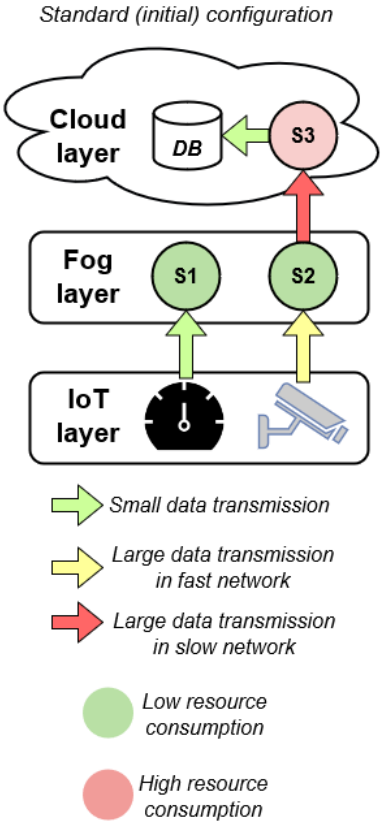

Figure 9: The system's default (initial) configuration

Therefore, in terms of networking capability, the Cloud layer is characterized with relatively poor data transmission performance and low bandwidth (as it relies on the internet). On the other hand, the Fog layer is accessible via a LAN (or WAN) network which results in better data transmission performance and a better bandwidth. To analyze such system's performance in efficiently analyzing the traffic (i.e., in reasonable delays), it is clear that it depends on the traffic itself. During week-days and regular working days, the border-crossing traffic is generally known to be moderate. On week-ends or holidays however, it is more likely to be of an important fluctuating activity. Through this system, we show how our modeling approach enables this smart city Cloud-Fog based application to adapt to traffic activity, according to the designed behaviors. We propose a scenario that illustrates the approach principle.

5.2. Adaptation scenario in high traffic activity

When the traffic activity rises, leading to a dense traffic jam, a concern particularly rises: data transmission between S2 and S3. When the number of vehicles rises, cameras take an important number of license plates pictures. After pre-processing, pictures need to be transmitted to the Cloud-based S3 service. This scenario creates an important bottleneck in the entire system as a result of the important data to be sent to the Cloud-based service S3 for the Fog-based service S2. On the other hand, if we consider that speed sensors transmit data only from a given speed threshold, the service S1 becomes unused in a traffic jam scenario (thus, it unnecessarily occupies computing resources). Such an observation of having an overloaded VM at Cloud layer and an unused/underused node at Fog layer is a typical situation that requires offloading the Cloud VM by relocating a service instance (S3) to the Fog node. At the same time, the Cloud layer can operate as a temporary backup solution for service availability. It is possible to temporarily move S1 to the Cloud in order to free Fog resource for S3.

Maude-based modeling of the system configuration: via our proposed modeling approach, the system designer/administrator can express any configuration within the presented Cloud-Fog system using the described Maude-based syntax. Ultimately, the designed can witness the provided self-adaptation and orchestration behaviors by simulating the system's execution from any configuration via the Maude rewriting engine. Furthermore, they can qualitatively verify the satisfaction of the defined self-adaptation and orchestration strategies via the Maude model-checker.

The described system standard or initial configuration is shown in Figure 9. It focuses on the nature of data transmissions and the services requirements. The Maude-based encoding of this Cloud-Fog configuration is given as follows :

```
CS<1/VM{1,S[maxS3,qS3,rtS3:ReqS3:over]:ReV1:over> ||
FS<1/N{1,S[maxS2,qS2,rtS2:ReqS2:over]:ReN1:over}
  | N{2,S[maxS1,qS1,rtS1:ReS1:unused]:ReN2:under}>
```

Service S3 is hosted in the Cloud VM, service S2 is deployed within a Fog node (N1) and S1 is hosted in another Fog Node (N2). With respect to the defined constructors, for each service S i, `maxS i` refers to its maximum response time threshold, `qS i` gives the number of handled requests, `ReqS i` gives its requirements in terms of resources and `rtS i` gives its current response time. Similarly, `Re(V/N) i` gives for each VM/Fog node its provided quantity of resources.

Explaining the designed behaviors: To understand the designed rewriting-based solution, we explain the orchestrator's reasoning process. Fist, the orchestrator monitors both Cloud and Fog layers to diagnose a set Mi of the satisfied monitoring predicates φi at every moment i. We assume that the Fog node containing the S1 service resource (`ReN2`) are sufficient to host the S3 service (i.e., $ReN2 > ReqS3$). We also assume that both S2 and S3 services are overloaded due to the explained bottleneck situation, i.e., $rt3 > maxS3$ and $rt2 > maxS2$. By monitoring the presented initial configuration, the set of satisfied monitoring predicates is given with $M1 = \{\varphi1, \varphi7, \varphi10, \varphi13, \varphi17, \varphi19\}$.

After diagnosing the system's state, the orchestrator triggers a self-adaptation action among $c1 - 8$ (at the Cloud layer), $f1 - 6$ (at the Fog layer) and $o1 - 2$ (for Cloud-Fog orchestration). As specified in the Cloud-Fog orchestrator's behavior (in the functional system *OrchBehavior*), notice that the only action than can be triggered is the *o1* rewrite rule (i.e., offloading the Cloud to the Fog).

When the rule is applied, it produces the following configuration:

```
CS<1/VM{1,nils:ReV1:unused}> ||
FS<1/N{1,S[maxS2,qS2',rtS2':ReqS2:over]:ReN1:over}
  | N{2,S[maxS1,qS1',rtS1':ReS1:unused]
  +S[maxS3,qS3',rtS3':ReqS3:over]:ReN2:over}>
```

Notice that the system is rewritten to move the service S3 from the Cloud VM to the less loaded Fog node, making all Fog nodes to be overloaded and the Cloud VM to be unused. From this configuration, the set of satisfied monitoring predicates evolves to: $M2 = \{\varphi3, \varphi11, \varphi17\}$. In this case, the rewrite rule *o2* (backup the Fog to the Cloud) becomes applicable. Its triggering results in the following configuration, which incarnates the desired configuration in the high traffic scenario (as shown in Figure 10):

```
CS<1/VM{1,S[maxS1,qS1',rtS1':ReS1:unused]:ReV1:stable}> ||
FS<2/N{1,S[maxS2,qS2",rtS2":ReqS2:over]:ReN1:stable}
  | N{2,S[maxS3,qS3",rtS3":ReqS3:over]:ReN2:stable}>
```

After this adaptation, both S2 and S3 are now hosted within the same network leading to considerably speed-up data transmission between them. Thus, we will consider that S2 and S3 respective response time drops (i.e., $rtS2'' < maxS2$ and $rtS3'' < maxS3$) as the data transmission bottleneck no longer exists, making the Fog host nodes to be in a stable state of provisioning. The set of monitoring predicates evolves to $M3 = \{\varphi17\}$ as the unused service S1 is detected. As soon as a speed sensor sends data to S1, monitoring would show $M = \emptyset$, meaning that no particular anomaly is detected, thus no adaptation is needed.

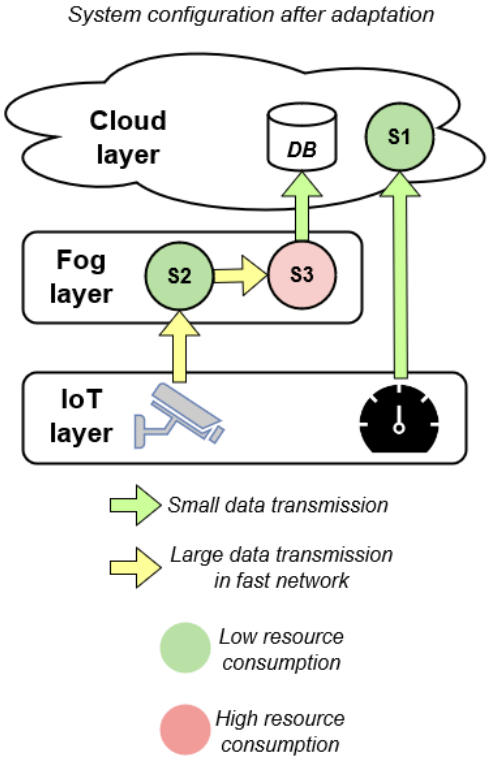

Figure 10: The system's configuration after adapting following the specified behaviors

In terms of qualitative verification via the Maude model-checker, the system designed provides the initial configuration encoding and a property formula $\in LTL(AP)$ as presented in Section 4.2.1. Particularly, notice that the first adaptation (rewrite rule *o1*) satisfies the LTL formula $Offload_C$: it made the system evolve from a global state (described with the set of predicates $M1$) where $\varphi1$, $\varphi13$ thus $\varphi19$ held, to a system state ($M2$) where $\varphi11$ holds and $\varphi1$ no longer holds ($\neg$). Similarly, the second adaptation (rewrite rule *o2*) satisfies the LTL formula $Backup_F$: it made the system evolve from a global state ($M2$) where $\varphi11$ held and $\varphi2$ did not, to a system state ($M3$) where $\varphi11$ no longer holds.

6. Related work

In the few last years, many research papers studied the problem of dynamic resource provisioning in the Cloud-Fog environments. Here we discuss some of this work then position our contribution accordingly. Overall, the currently published work highlight the Cloud-Fog resource provisioning principles by proposing system models, architectures and optimization algorithms. The different available contributions go towards the optimization of response time of Fog-enabled applications with awareness of network and server usage.

Authors of [22] propose a novel method for service deployment on Fog landscape focusing on application's response time, network congestion and server usage. They propose policies for Cloud and Fog service deployment strategies in order to control service placement using resource and response time constraint formulas. They provide an execution plan to achieve "best service deployment" in order to provide trade-off among minimizing network congestion, minimizing application's response time and minimizing Cloud layer's server usage. Authors provide simulation-based experimentation to motivate their approach's feasibility and results. In [26], authors study the location of Fog nodes in Fog-Cloud infrastructure. The main approach's goal is to decide on the locations where Fog nodes should be deployed and how each node should configured in terms of resource capacity. They provide a multicriteria decision model based on multi-objective constraints to optimize Fog nodes placement and usage. The solution's goal is to minimize overall infrastructure costs while maximizing overall service performance. Authors provide a simulation-based experimental study to show their solution performance in improving Fog service and reducing cost. In [29] authors present a conceptual framework for resource provisioning in IoT/Fog landscape. They formalize an optimization problem to design delay-sensitive utilization of the Fog layer's computational resources. They propose a system architecture highlighting a Cloud-Fog middleware as a central unit which manages resource provisioning: by controlling the Cloud layer and by interacting with a Fog orchestration controller. The central middleware's purpose is processing into the layer Cloud non delay-sensitive tasks or tasks that cannot execute in the Fog layer. Authors extend their approach in [28] to detail the service placement optimization and the Fog orchestration considerations. The approach is concertized in [27] to enable real-world implementation of the proposed solution through a framework named *FogFrame*. In their work, authors study Cloud-Fog resource management using several approaches (a greedy first fit heuristic, a genetic algorithm, and an exact optimization method). The results showed different behaviors and cost/performance trade-offs by optimizing service placement in the Clouf-Fog environment. These research works [22, 26, 27] give significant information to correctly design Cloud-Fog environments through system models and architectures. They give substantial guidelines to accurately implement Fog resource management through strategies and constraints. Finally, they highlight relevant methods and approaches to thoroughly conduct quantitative evaluation of the designed behaviors.

In this paper, we tend to formalize the extracted techniques in order to enable Cloud-Fog resource management. We model the Cloud-Fog environment in terms of structure (as a pool of resource) and behavior, by reproducing the main methods including service mobility in the Fog, service migration in the Cloud, load-balancing, etc. We inspire from the various presented formulas and constraints to design conditional behaviors which lead, for example, to deploy a service in the Cloud rather than the Fog and vice-versa. Through our formal approach, we tend to show that how all these behaviors can be ensured in an autonomic manner and therefore be qualified as self-adaptation. We introduce a Cloud-Fog orchestrator which manages this self-adaptation by controlling when, how and where to trigger the proper actions basing on both Fog and Cloud observations. Our main focus is to conduct qualitative verification of the designed behaviors (i.e., study their correctness). The proposed Cloud-Fog orchestrator behaviors can be simulated as they are executable through the Maude-based rewriting system. The introduced temporal properties in Linear Temporal Logic allow verifying the orchestrator's behavioral correctness via the Maude-integrated model-checker.

Authors in [30] present Fog orchestration as a technique leading to resource management in the Cloud-Fog environment. They introduce the main issues, challenges and research directions. Authors put the basis of Cloud-Fog orchestration in order to ensure the low-latency requirements of IoT environments. Precisely, they highlight a list of criteria that need to be ensured to fully address the Cloud-Fog resource problem including dynamicity, scale, and complexity, among others. These criteria state that a proper solution for Cloud-Fog resource orchestration needs (1) to capture the highly dynamic nature of the IoT-Fog-Cloud environment by accurately monitoring events and states evolution. It needs to (2) support systems scalability and complexity resulting from the increasing IoT manufacturers and Cloud/-Fog providers, which lead to interoperability and heterogeneous concerns and overlapping requirements.

In this paper, we provide a formal model to orchestrate the Cloud-Fog environment's management. We propose a way to manage self-adaptation of both Cloud and Fog layers in order to achieve this goal. In our modeling approach, the designed self-adaptation mechanisms answer the dynamicity criteria: we modeled several atomic actions such as Cloud and/or Fog service replication, service mobility across Fog nodes and migration across Cloud VMs as well as service offloading from Cloud to Fog (and vice-versa), among others. Our approach also supports scalability by managing the adaptation's triggering in such a way to ensure different requirements in terms of Cloud VMs' and Fog nodes load, and services resource requirements. Furthermore, we model the entire Cloud-Fog environment as sets (or pools) of resource (VMs, Nodes and their allocated computational resource). Load, resource and service placement are easily monitored through different predicates that capture their state and evolution, which answers the dynamicity criteria. In addition, we reduce the problem's complexity by providing a generic modeling approach which is technology-free and provider agnostic.

Finally, our LTL-based encoding of the system's temporal evolution through properties describing its behavioral correctness go towards supporting (and ensuring) both dynamicity and complexity. In this line of work, authors of [24] provide a formal modeling approach to manage self-adaptive behaviors in Fog-based systems. They propose a Bigraphical Reactive Systems (BRS for short) modeling of Fog systems'

structure and behavior. They provide axiomatic construction rules to describe their spatial distribution and Bigraphical reaction rules to describe their temporal evolution. Generic adaptation actions are proposed and formal verification of the behavior's qualitative correctness are discussed.

7. Conclusion

In this paper, we proposed a formal based solution to design, implement and verify Cloud-Fog self-adaptation and orchestration, aiming at optimizing the use of resource pools available at both Cloud and Fog layers in order to accurately meet service requirements. First, we modeled Cloud and Fog layers in terms of structure and behavior to identify a set of monitoring predicates and a set of atomic adaptation actions. The predicates were used to diagnose both layers' states in terms of resource provisioning. The actions were used to identify adaptation mechanisms to apply. In addition, we introduced a Cloud-Fog orchestrator which decides of the actions to be triggered in order to adapt at Cloud and/or Fog layers. The orchestrator considers the observed states (monitoring predicates) of both layers and then applies the proper sequence of actions to achieve an adaptation at one or both layers. Finally, we provided a set of temporal properties to be satisfied to study the orchestrator's behaviors and ensure their qualitative correctness.

To achieve all these goals, we proposed a formal modeling approach of self-adaptive Cloud and Fog orchestration based on rewriting logic. We used the formal specification language called Maude and its associated tools including a model-checker for formal qualitative verification. We showed that Maude provides an adequate expressiveness to model the structure of a Cloud-Fog environment through declarative constructors and its state through first order predicates. We demonstrated that the rewriting logic semantics were relevant in designing the modeled adaptation actions through conditional rewrite rules that we designed to be complementary and composable. Furthermore, we expressed temporal properties with *Linear Temporal Logic* (LTL) to study the managed Cloud-Fog environment temporal evolution in a qualitative point of view. We showed how to enable formal verification of the defined behaviors through the Maude-integrated model-checker. The model-checker conducts state-based verification of the LTL properties by implementing a *Kripke*structure to tackle the state explosion problem. Finally, we illustrated our modeling approach and discussed the qualitative verification of the introduced behaviors' correctness through a case study in a Cloud/Fog-based smart city scenario.

As future work, we are considering three main extensions for this work: (1) enabling time-aware modeling and analysis, (2) integrating the in-Maude strategies and (3) providing solutions for the orchestrator's fault-tolerance and resilience. The first extension is to provide time-enabled modeling and analysis of the designed behaviors. Such modeling would push the decision-making even further by integrating strict/loose response time modeling of services' requirements. It would also enable quantitative evaluation and validation of the proposed behaviors. The second extension is to define a set of Maude strategies to control the condition rewrite rules triggering. Maude strategies can describe accurate behavior patterns in order to apply precedence and priorities between the rules. Such modeling would improve the model's behavioral correctness as well as optimizing it by reducing the possible intermediary states. Finally, the third extension is to provide restoring capabilities and continuity solutions to ensure the orchestrator's behavior fault-tolerance as it may be subject of anomalies if any kind, linked to software or hardware failure.

A. Maude specification modules for Cloud/Fog self-adaptation and orchestration

Listing 1 shows the Maude functional module *ServiceSpec*. It encodes our specification of software services running on the Cloud and/or the Fog layers. The specification is detailed under Section 3.1.

Listing 1: Functional module: ServiceSpec

```
fmod ServiceSpec is
protecting NAT FLOAT BOOL .
sorts  S SL state RES.
subsort S < SL .
---Service and resource construction axioms
op S[_,_,_:_:_] : Nat Nat Nat RES state -> S [ctor] .
op -_,_,_- : FLOAT FLOAT FLOAT -> RES [ctor] .
ops stable overloaded unused  underused : -> state [ctor]
---service lists
op nils : -> SL [ctor] .
op _+_ : SL SL -> SL [ctor assoc comm id: nils] .
---Monitoring predicates
ops overS(_) unusedS(_) stableS(_): S -> Bool .
...
---Operations
op loadS(_) : S -> Nat .
op rtS(_) : S -> Nat .
op reqS(_) : S -> RES .
ops _>_, _<_, _=_ : RES RES -> BOOL .
...
endfm
```

Listing 2 shows the Maude functional module *CloudSpec*. It encodes our specification of Cloud systems. The specification is commented under Section 3.2.

Listing 2: Functional module: CloudSpec

```
fmod CloudSpec is
protecting NAT FLOAT BOOL .
including ServiceSpec
sorts CS VM VML .
subsort VM < VML .
---Cloud layer construction axioms
op CS<_/_> : Nat VML -> CS [ctor] .
op VM{_,_:_:_} : Nat SL RES state -> VM [ctor] .
---VM lists
op nilv : -> VML [ctor] .
op _|_ : VML VML -> VML [ctor assoc comm id: nilv] .
---Monitoring predicates
ops AoverV(_) EoverV(_) EunV(_) AoverCS(_)
        EoverCS(_) unS(_) MIGpredC(_)
        LBSpredC(_): CS -> Bool .
ops overV(_) unV(_) underV(_) stableV(_): VM -> Bool .
...
---Access operations
op loadCS(_) : CS -> Nat .
op loadV(_) : VM -> Nat .
op resV(_) : VM -> RES .
...
```

```
---Reconfiguration actions
ops newV(_) newCS(_) MIGc(_) LBSc(_)
        Vin(_) CSin(_) : CS -> CS .
ops scaleUpV(_) scaleDownV(_) : VM -> VM .
...
endfm
```

Listing 3 shows the Maude functional module *FogSpec* encoding our specification of Fog systems. The specification is detailed under Section 3.3.

Listing 3: Functional module: FogSpec

```
fmod FogSpec is
protecting NAT FLOAT BOOL .
including ServiceSpec
sorts FS N NL.
subsort N < NL .
---Fog layer construction axioms
op FS<_/_> : Nat NL -> FS [ctor] .
op N{_,_:_:_} : Nat SL RES state -> N [ctor] .
---Node lists
op niln : -> NL [ctor] .
op _|_ : NL NL -> NL [ctor assoc comm id: niln] .
---Monitoring predicates
ops AoverN(_) EoverN(_) EunN(_) AoverFS(_)
        EoverFS(_) unFS(_) MIGpredF(_) : FS -> Bool .
ops overN(_) unN(_) underN(_): N -> Bool .
...
---Access operations
op loadFS(_) : FS -> Nat .
op loadN(_) : N -> Nat .
op resN(_) : N -> RES .
...
---Reconfiguration actions
ops onN(_) newSF(_) MIGf(_) offN(_) SinF(_) : FS -> FS .
...
endfm
```

Listing 4 shows the Maude system module *OrchBehavior* encoding our Cloud-Fog orchestrator specification and behavior. The specification is commented under Section 4.1.

Listing 4: System module: Cloud-Fog orchestrator behaviors

```
mod OrchBehavior is
protecting NAT FLOAT .
including ServiceSpec CloudSpec FogSpec.
sort ENV .
---Cloud-Fog environment specification
op _||_ : CS FS -> ENV [ctor] .
---Variables
var cs : CS . var fs : FS .
var vm : VM . var node : N .
var vml : VML . var nl : NL .
vars s : S . vars sl sl2 : SL .
vars x y z x2 y2 z2 : NAT .
vars st st2 : state . vars res res2 : RES .
---Cloud local adaptation
crl [c1-Low]: cs || fs => newV(cs) || fs
        if (AoverV(cs)) .
crl [c1-Hi]: cs || fs => newV(cs) || fs
        if (EoverV(cs)
        and not (EunV(cs) or EunderV(cs))) .
crl [c2]: cs || fs => Vin(cs) || fs
        if (EunV(cs) and not AoverV(cs)) .
crl [c3] : cs || fs => MIGc(cs) || fs
        if (MIGpredC(cs)) .
crl [c4-Low]: cs || fs => newCS(cs) || fs
        if (AoverCS(cs)) .
crl [c4-Hi]: cs || fs => newCS(cs) || fs
        if (EoverCS(cs)) .
crl [c5]: cs || fs => SinC(cs) || fs
        if (EunCS(cs) and not EoverCS(cs)) .
crl [c6]: cs || fs => LBSc(cs) || fs
        if (LBSpredC(cs)) .
crl [c7]: CS<x,y,z/vm|vml:st> || fs
        => CS<x,y,z/scaleUpV(vm)|vml:st> || fs
        if (overV(vm)) .
crl [c8]: CS<x,y,z/vm|vml:st> || fs
        => CS<x,y,z/scaleDownV(vm)|vml:st> || fs
        if (underV(vm)) .
---Fog local adaptation
crl [f1-low]: cs || fs => cs || onN(fs)
        if (AoverN(fs)) .
crl [f1-Hi]: cs || fs => cs || onN(fs)
        if (EoverN(fs))
        and not (EunN(fs) or EunderN(fs))) .
crl [f2]: cs || fs => cs || offN(fs)
        if (EunN(fs)
        and not (AoverN(fs) or EunderN(fs))) .
crl [f3] : cs || fs => cs || MIGf(fs)
        if (MIGpredF(fs)) .
crl [f4-Low]: cs || fs => cs || newFS(fs)
        if (AoverFS(fs)) .
crl [f4-Hi]: cs || fs => cs || newFS(fs)
        if (EoverFS(fs) and not EunFS(fs)) .
crl [f5]: cs || fs => cs || SinF(fs)
        if (EunFS(fs) and not EoverFS(fs)) .
crl [f6]: cs || fs => cs || LBSf(fs)
        if (LBSpredF(fs)) .
---Cloud-Fog actions
crl [o1]:
        CS<x,y,z/VM{y,s+sl:res:st}|vml> ||
        FS<x2,y2,z2/N{y2,sl2:res2:st2}|nl>
        =>
        CS<x,y,z/VM{y,sl:res:st}|vml> ||
        FS<x2,y2,z2/N{y2,s+sl2:res2:st2}|nl>
        if ( (st==overloaded
                or net(res)< net(resS(s)))
                and st2!=overloaded
                and res2>resS(s) ) .
crl [o2]:
        FS<x,y,z/N{y,s+sl:res:st}|nl> ||
        CS<x2,y2,z2/VM{y2,sl2:res2:st2}|vml>
        =>
        FS<x,y,z/N{y,sl:res:st}|nl> ||
        CS<x2,y2,z2/VM{y2,s+sl2:res2:st2}|vml>
        if ( st==overloaded and AoverN(sl)
                and st2!=overloaded
                and res2>resS(s) ) .
endm
```

Listing 5 shows the Maude system module *Properties* implementing the Kripke structure and LTL formulas to allow formal verification of the designed behaviors via the Maude model-checker. The specification is commented under Section 4.2.2.

Listing 5: System module: Properties decalarations

```
mod Properties is
including MODEL-CHECKER LTL-SIMPLIFIER SATISFACTION .
protecting OrchBehavior .
subsort CS < State . subsort FS < State .
---Atomic propositions (monitoring predicates)
ops p1 p2 p3 ... p20 : -> Prop [ctor] .
---Properties expressing strategies satisfaction
ops ScaleOutVM ScaleOutS ScaleInVM ScaleOutS
        ScaleUp ScaleDown Migration LoadBalacingC
        Provision Deprovision Mobility LoadBalaneF
        OffloadC BackupF : -> Prop [ctor] .
---Variables for symbolic reasoning
var cs : CS .
var fs : FS .
var P : Prop .
---Defining Cloud symbolic states
ceq cs |= p1 = true if EoverV( cs ) == true .
ceq cs |= p2 = true if AoverV( cs ) == true .
...
ceq cs |= p9 = true if LBSpredC( cs ) == true .
---Defining Fog symbolic states
ceq fs |= p10 = true if EoverN( fs ) == true .
...
ceq fs |= p15 = true if EoverFS( fs ) == true .
...
---Encoding LTL formulas
eq ScaleOutCVM = [] ( p2 \/ (p1 /\ ~ p3) -> <> ~ p2 ) .
...
eq LoadBalanceC = [] ((p6 /\ p8)-> O p9 ) U ~ p6) .
...
endm
```

B. Explaining the defined LTL property formulas

The LTL property formulas introduced under Section 4.2.1 are explained here. We detail each property formula for the Cloud layer self-adaptation (eq. 1 to eq. 8), the Fog layer self-adaptation (eq. 9 to eq. 14) and their orchestration (eq. 15 and eq. 16). For remainder and to ease understanding the formulas, we duplicate the used LTL operators and symbols Table 5 of Section 4.2.1 as Table 6 in this Section.

Table 6
used LTL operators and symbols

LTL operator/symbol	Meaning
$\wedge$	conjunction / and
$\vee$	disjunction / or
$\rightarrow$	implies
$\neg$	negation / not
$\Box$	globally / always
$\Diamond$	eventually or "in the future"
$\bigcirc$	next time
$\mathcal{U}$	until

B.1. Cloud self-adaptation properties

$$ScaleOut_C^{VM} \equiv \Box\big[(\varphi 2 \vee (\varphi 1 \wedge \neg\varphi 3)) \rightarrow \Diamond\neg\varphi 2\big] \quad (1)$$

The $ScaleOut_C^{VM}$ formula describes the system's ability to scale-out at the infrastructure level of the Cloud layer by adding a VM instance, in response of a global state of under-provisioning (i.e., overloading) at the infrastructure level. It states the following: when all Cloud VMs are overloaded ($\varphi 2$) or when a VM is overloaded ($\varphi 1$) and no VM is unused ($\neg\varphi 3$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) end up by reaching a state where all VMs are not overloaded ($\neg\varphi 2$) and this pattern is always true ($\Box$) i.e., repeats indefinitely.

$$ScaleOut_C^{S} \equiv \Box\big[(\varphi 7 \vee (\varphi 6 \wedge \neg\varphi 8)) \rightarrow \Diamond\neg\varphi 7\big] \quad (2)$$

The $ScaleOut_C^{S}$ formula describes the system's ability to scale-out at the application levels of the Cloud layer by adding a service instance, in response of a global state of under-provisioning at the application level. It states the following: when all Cloud services are overloaded ($\varphi 7$) or a service is overloaded ($\varphi 6$) and no service is unused ($\neg\varphi 8$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) end up by reaching a state where all services are not overloaded ($\neg\varphi 7$) and this pattern is always true ($\Box$) i.e., repeats indefinitely.

$$ScaleIn_C^{VM} \equiv \Box\big[((\varphi 3 \vee \varphi 4) \wedge \neg\varphi 1) \rightarrow \Diamond\neg\varphi 3\big] \quad (3)$$

The $ScaleIn_C^{VM}$ formula describes the system's ability to scale-in at the infrastructure level of the Cloud layer by removing a VM instance, in response of a global state of over-provisioning at the infrastructure level. It states the following: when a Cloud VM is unused ($\varphi 3$) or is underused ($\varphi 4$) and no VM is overloaded ($\neg\varphi 1$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) end up by reaching a state where no VM is unused ($\neg\varphi 3$) and this pattern repeats indefinitely ($\Box$).

$$ScaleIn_C^{S} \equiv \Box\big[(\varphi 8 \wedge \neg\varphi 6) \rightarrow \Diamond\neg\varphi 8\big] \quad (4)$$

The $ScaleIn_C^{S}$ formula describes the system's ability to scale-in at the application level of the Cloud layer by removing a Service instance, in response to a state of over-provisioning of the application level. It states the following: when a Cloud service is unused ($\varphi 8$) and no Cloud service is overloaded ($\neg\varphi 6$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where no Cloud service is unused ($\neg\varphi 8$) and this pattern repeats indefinitely ($\Box$).

$$ScaleUp_C \equiv \Box\big[\varphi 2 \vee (\varphi 1 \wedge \neg(\varphi 3 \vee \varphi 4)) \rightarrow \Diamond\neg\varphi 1\big] \quad (5)$$

The $ScaleUp_C$ formula describes the system's ability to scale-up by adding more resources to a VM instance, in response of a global state of under-provisioning (i.e., overloading) at the infrastructure level. It states the following: when all Cloud VMs are overloaded ($\varphi 2$) or when a VM is overloaded ($\varphi 1$) and no VM is unused or underused ($\neg(\varphi 3 \vee \varphi 4)$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where no VM is overloaded ($\neg\varphi 1$) and this pattern repeats indefinitely ($\Box$).

$$ScaleDown_C \equiv \Box\big[(\varphi 4 \wedge \neg\varphi 1) \rightarrow \Diamond\neg\varphi 4\big] \quad (6)$$

The $ScaleDown_C$ formula describes the system's ability to scale-down by removing resources from a VM instance, in response of a global state of over-provisioning at the infrastructure level. It states the following: when a VM is underused ($\varphi 4$) and no VM is overloaded ($\neg\varphi 1$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where no VM is underused ($\neg\varphi 4$) and this pattern repeats indefinitely ($\Box$).

$$LoadBalance_C \equiv \Box\big[((\varphi 6 \wedge \varphi 8) \rightarrow \bigcirc\varphi 9)\, \mathcal{U}\, \neg\varphi 6\big] \quad (7)$$

The formula $LoadBalance_C$ for the Cloud layer describes the system ability to balance the Cloud services load by redirecting requests across the services. It states the following: when a service is overloaded ($\varphi 6$) and another service is unused ($\varphi 8$), it implies that the next ($\bigcirc$) expected state is when requests redirecting across Cloud services is applied ($\varphi 9$), until ($\mathcal{U}$) no service instance is overloaded ($\neg\varphi 6$) and this pattern is always true ($\Box$).

$$Migrate_C \equiv \Box\big[(\varphi 1 \wedge (\varphi 3 \vee \varphi 4) \rightarrow \bigcirc\varphi 5)\, \mathcal{U}\, \neg\varphi 1\big] \quad (8)$$

The formula $Migrate_C$ for the Cloud layer describes the system ability to balance the Cloud VMs load by relocating (migrating) services across the VMs. It states the following:

when a VM is overloaded ($\varphi 1$) and another VM is unused or underused ($\varphi 3 \vee \varphi 4$), it implies that the next ($\bigcirc$) expected state is when services migration across Cloud VMs is applied ($\varphi 5$), until ($\mathcal{U}$) no VM instance is overloaded ($\neg\varphi 1$) and this pattern is always true ($\Box$).

B.2. Fog self-adaptation properties

$$Provision_F^N \equiv \Box\big[(\varphi 11 \vee (\varphi 10 \wedge \neg\varphi 12)) \rightarrow \Diamond\neg\varphi 11\big] \quad (9)$$

The $Provision_F^N$ formula describes the system's ability to provision more Fog nodes by switching them on, in response of a global state of under-provisioning at the nodes level. It states the following: when all Fog nodes are overloaded ($\varphi 11$) or when a Fog node is overloaded ($\varphi 10$) and no node is unused ($\neg\varphi 12$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where not all Fog nodes are overloaded ($\neg\varphi 11$) and this pattern repeats indefinitely ($\Box$).

$$Provision_F^S \equiv \Box\big[(\varphi 16 \vee (\varphi 15 \wedge \neg\varphi 17)) \rightarrow \Diamond\neg\varphi 16\big] \quad (10)$$

The $Provision_F^S$ formula describes the system's ability to provision more Fog services by deploying new service instances, in response of a global state of under-provisioning at the application level. It states the following: when all Fog services are overloaded ($\varphi 16$) or when a Fog service is overloaded ($\varphi 15$) and no services is unused ($\neg\varphi 17$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where not all Fog services are overloaded ($\neg\varphi 11$) and this pattern repeats indefinitely ($\Box$).

$$Deprovision_F^N \equiv \Box\big[((\varphi 12 \vee \varphi 13) \wedge \neg\varphi 10) \rightarrow \Diamond\neg\varphi 12\big] \quad (11)$$

The $Deprovision_F^N$ formula describes the system's ability to deprovision Fog nodes by switching them off, in response of a global state of over-provisioning at the Fog nodes level. It states the following: when a Fog node is unused ($\varphi 12$) or is underused ($\varphi 13$) and no node is overloaded ($\neg\varphi 10$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where no Fog node is unused ($\neg\varphi 12$) and this pattern repeats indefinitely ($\Box$).

$$Deprovision_F^S \equiv \Box\big[(\varphi 17 \wedge \neg\varphi 15) \rightarrow \Diamond\neg\varphi 17\big] \quad (12)$$

The $Deprovision_F^S$ formula describes the system's ability to deprovision Fog services by destroying the unused ones, in response of a global state of over-provisioning at the Fog application level. It states the following: when a Fog service is unused ($\varphi 17$) and no service is overloaded ($\neg\varphi 15$), it implies ($\rightarrow$) that the system will eventually ($\Diamond$) reach a state where no Fog service is unused ($\neg\varphi 17$) and this pattern repeats indefinitely ($\Box$).

$$LoadBalance_F \equiv \Box\big[((\varphi 15 \wedge \varphi 17) \rightarrow \bigcirc\varphi 18)\,\mathcal{U}\,\neg\varphi 15\big] \quad (13)$$

The formula $LoadBalance_F$ for the Fog layer describes the system ability to balance the Fog services load by redirecting requests across the services. It states the following: when a service is overloaded ($\varphi 15$) and another service is unused ($\varphi 17$), it implies that the next ($\bigcirc$) expected state is when requests redirection across Fog services is applied ($\varphi 18$), until ($\mathcal{U}$) no Fog service instance is overloaded ($\neg\varphi 15$) and this pattern is always true ($\Box$).

$$Mobility_F \equiv \Box\big[(\varphi 10 \wedge (\varphi 12 \vee \varphi 13) \rightarrow \bigcirc\varphi 14)\,\mathcal{U}\,\neg\varphi 10\big] \quad (14)$$

The formula $Mobility_F$ for the Fog layer describes the system ability to balance the Fog nodes load by relocating services across the nodes. It states the following: when a Fog node is overloaded ($\varphi 10$) and another Fog node is unused or underused ($\varphi 12 \vee \varphi 13$), it implies that the next ($\bigcirc$) expected state is when services mobility across Fog nodes is applied ($\varphi 14$), until ($\mathcal{U}$) no Fog node is overloaded ($\neg\varphi 10$) and this pattern is always true ($\Box$).

B.3. Cloud-Fog orchestration properties

$$Offload_C \equiv \Box\big[(\varphi 1 \wedge (\varphi 12 \vee \varphi 13) \rightarrow \bigcirc\varphi 19)\,\mathcal{U}\,(\neg\varphi 1 \vee \varphi 11)\big] \quad (15)$$

The $Offload_C$ formula describes the system's ability to offload the Cloud layer towards the Fog layer by relocating a Cloud service from a Cloud VM to Fog node. It states the following: when a Cloud VM is overloaded ($\varphi 1$) and a Fog node is unused or underused ($\varphi 12 \vee \varphi 13$), it implies that the next ($\bigcirc$) expected state is when services relocating from Cloud VMs to Fog nodes is applied ($\varphi 19$), until ($\mathcal{U}$) no Cloud VM is overloaded or all Fog nodes are overloaded ($\neg\varphi 1 \vee \varphi 11$) and this pattern is always true ($\Box$).

$$Backup_F \equiv \Box\big[((\varphi 11 \wedge \neg\varphi 2) \rightarrow \bigcirc\varphi 20)\,\mathcal{U}\,(\neg\varphi 11 \vee \varphi 2)\big] \quad (16)$$

The $Backup_F$ formula describes the system's ability to backup the Fog layer towards the Cloud layer by relocating a Fog service from a Fog node to Cloud VM. It states the following: when all Fog nodes are overloaded ($\varphi 11$) and not all Cloud VMs are overloaded ($\neg\varphi 2$), it implies that the next ($\bigcirc$) expected state is when services relocating from Fog nodes to Cloud VMs is applied ($\varphi 20$), until ($\mathcal{U}$) not all Fog nodes are overloaded or all Cloud VMs are overloaded ($\neg\varphi 11 \vee \varphi 2$) and this pattern is always true ($\Box$).

References

[1] Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., Merle, P., 2017. Elasticity in cloud computing: state of the art and research challenges. IEEE Transactions on Services Computing 11, 430–447.

[2] Baier, C., Katoen, J.P., 2008. Principles of model checking. The MIT Press, Cambridge, Mass, USA. OCLC: ocn171152628.

[3] Bonomi, F., Milito, R., Zhu, J., Addepalli, S., 2012. Fog computing and its role in the internet of things, in: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, ACM. pp. 13–16.

[4] Cisco, 2015. Fog computing and the internet of things: Extend the cloud to where the things are. URL: https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf.

[5] Clavel, M., Duran, F., Eker, S., Escobar, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Talcott, C., 2017. Maude Manual (Version 2.7.1) , 521.

[6] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2007. All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic. Springer-Verlag.

[7] De Oliveira, F.A., Ledoux, T., Sharrock, R., 2013. A framework for the coordination of multiple autonomic managers in cloud environments, in: 2013 IEEE 7th International Conference on Self-adaptive and Self-organizing Systems, IEEE. pp. 179–188.

[8] Delicato, F.C., Pires, P.F., Batista, T., 2017. The resource management challenge in iot, in: Resource management for Internet of Things. Springer, pp. 7–18.

[9] Erl, T., 2005. Service-oriented architecture: concepts, technology, and design. Prentice Hall International.

[10] Galante, G., Bona, L.C.E.d., 2012. A Survey on Cloud Computing Elasticity, in: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing, IEEE Computer Society, Washington, DC, USA. pp. 263–270. URL: http://dx.doi.org/10.1109/UCC.2012.30, doi:10.1109/UCC.2012.30.

[11] Herbst, N.R., Kounev, S., Reussner, R., 2013. Elasticity in Cloud Computing: What It Is, and What It Is Not, in: Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13), USENIX, San Jose, CA. pp. 23–27. URL: https://www.usenix.org/conference/icac13/technical-sessions/presentation/herbst.

[12] Kephart, J.O., Chess, D.M., 2003. The vision of autonomic computing. Computer , 41–50.

[13] Khebbeb, K., 2019. Formalizing and Evaluating Cross-Layer Elasticity Strategies in the Cloud. Theses. LIUPPA - Laboratoire Informatique de l'Université de Pau et des Pays de l'Adour ; LIRE - Laboratoire d'Informatique Répartie de l'Université Constantine 2. URL: https://tel.archives-ouvertes.fr/tel-02271523.

[14] Khebbeb, K., Hameurlain, N., Belala, F., 2018a. Modeling and evaluating cross-layer elasticity strategies in cloud systems, in: International Conference on Model and Data Engineering, Springer. pp. 168–183. doi:https://doi.org/10.1007/978-3-030-00856-7_11.

[15] Khebbeb, K., Hameurlain, N., Belala, F., 2019. Formal Modeling and Verification of Cloud Elasticity with Maude and LTL, in: Attiogbé, C., Ferrarotti, F., Maabout, S. (Eds.), New Trends in Model and Data Engineering, Springer International Publishing, Cham. pp. 64–77. doi:10.1007/978-3-030-32213-7_5.

[16] Khebbeb, K., Hameurlain, N., Belala, F., 2020. Formalizing and simulating cross-layer elasticity strategies in cloud systems. Cluster Computing doi:10.1007/s10586-020-03080-8.

[17] Khebbeb, K., Hameurlain, N., Belala, F., Sahli, H., 2018b. Formal modelling and verifying elasticity strategies in cloud systems. IET Software 13, 25–35. URL: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2018.5030, doi:10.1049/iet-sen.2018.5030.

[18] Maamar, Z., Baker, T., Faci, N., Ugljanin, E., Khafajiy, M.A., Burégio, V., 2019. Towards a Seamless Coordination of Cloud and Fog: Illustration Through the Internet-of-things, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, ACM, New York, NY, USA. pp. 2008–2015. URL: http://doi.acm.org/10.1145/3297280.3297477, doi:10.1145/3297280.3297477. event-place: Limassol, Cyprus.

[19] Martí-Oliet, N., Meseguer, J., 1996. Rewriting logic as a logical and semantic framework. Electronic Notes in Theoretical Computer Science 4, 190–225.

[20] Mell, P., Grance, T., 2011. The NIST Definition of Cloud Computing , 7.

[21] Menychtas, A., Gatzioura, A., Varvarigou, T., . A business resolution engine for cloud marketplaces, in: 2011 IEEE Third International Conference on Cloud Computing Technology and Science, IEEE. pp. 462–469.

[22] Pham-Nguyen, H.N., Tran-Minh, Q., 2019. Dynamic resource provisioning on fog landscapes. Security and Communication Networks 2019.

[23] Rozier, K.Y., 2011. Linear temporal logic symbolic model checking. Computer Science Review 5, 163–203.

[24] Sahli, H., Ledoux, T., Rutten, É., 2019. Modeling self-adaptive fog systems using bigraphs.

[25] Schoren, R., . Correspondence between kripke structures and labeled transition systems for model minimization.

[26] da Silva, R.A., da Fonseca, N.L., 2019. On the location of fog nodes in fog-cloud infrastructures. Sensors 19, 2445.

[27] Skarlat, O., Karagiannis, V., Rausch, T., Bachmann, K., Schulte, S., 2018. A framework for optimization, service placement, and runtime operation in the fog, in: 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), IEEE. pp. 164–173.

[28] Skarlat, O., Nardelli, M., Schulte, S., Borkowski, M., Leitner, P., 2017. Optimized iot service placement in the fog. Service Oriented Computing and Applications 11, 427–443.

[29] Skarlat, O., Schulte, S., Borkowski, M., Leitner, P., 2016. Resource provisioning for iot services in the fog, in: 2016 IEEE 9th international conference on service-oriented computing and applications (SOCA), IEEE. pp. 32–39.

[30] Wen, Z., Yang, R., Garraghan, P., Lin, T., Xu, J., Rovatsos, M., 2017. Fog orchestration for internet of things services. IEEE Internet Computing 21, 16–24.

[31] Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M., 2013. On patterns for decentralized control in self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II. Springer, pp. 76–107.

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

Khaled Khebbeb received a Ph.D. degree in computer science from the University of Pau and countries of Adour, France and the University of Constantine 2, Algeria in 2019. Since September 2018, he is a research and teaching assistant at the University of Pau and is affiliated to the MOVIES team of the LIUPPA laboratory. His research interests include software engineering and formal modeling of self-adaptive software systems applied on cloud and service-oriented computing.

Nabil Hameurlain received a Ph.D. degree in computer science from the University of Toulouse, France in 1998 and a HdR (French Habilitation to become Research Activity Supervisor) in Computer science from the University of Pau in 2011. Since October 1999, he is associate professor at the University of Pau and the head of the MOVIES team at the LIUPPA Laboratory. His main research interests include software engineering for distributed and self-adaptive software systems, with a particular focus on cloud and service oriented computing.

Faiza Belala received a Ph.D. degree in computer science from Mentouri University of Constantine in 2001. She is currently a Professor at the same university and head of the GLSD team (LIRE Laboratory). Her current research focuses on architecture description languages, formal refinement (Rewriting Logic, Bigraphs, Petri nets, ect.), mobility and concurrency aspects in software architectures, formal analysis of distributed systems. She has organized and chaired the international conferences on Advanced Aspects of Software Engineering ICAASE 2014/2016/2018, she is the author of many refereed journal articles and peer reviewed international and regional conference papers. She has supervised over sixty Master and Ph.D. theses.