FEDCON: An embeddable Framework for Managing MOC Functions and Interfaces in Federated Software Networks

1st Michael Steinke
Computer science department
Universität der Bundeswehr München
Neubiberg, Germany
michael.steinke@unibw.de

2nd Wolfgang Hommel
Computer science department
Universität der Bundeswehr München
Neubiberg, Germany
wolfgang.hommel@unibw.de

***Abstract*—Software Networks (SN), combining network virtualization, system virtualization, NFV, and SDN, are highly suitable for the establishment of federated loose infrastructures, e. g., due to their flexibility, expandability, and concealment of geographical segregation of IT resources. Current management platforms for suchlike SNs, however, almost entirely omit network federation with its organizational aspects and cannot handle network heterogeneity.**

In this paper, in order to contribute to close this gap, we do not strive for a stand-alone solution, but present the FEDerated CONtrol (FEDCON) framework, which is designed to be embeddable into existing platforms. It facilitates access to a federated SN's functionality by providing the modeling and secure invocation of functions and interfaces of network components, like SDN-controllers or hypervisors. FEDCON considers federation-specific organisational aspects like geographically distributed data centers, administrative domains, and use-case-specific adaptable access controls. FEDCON is implemented in Lua, exploiting its focus on embeddability into existing software.

***Index Terms*—SDN, NFV, federated infrastructure, network management, network programmability, management platform**

I. Introduction

Software Networks (SNs) comprise systems- and network-virtualization, *Network Functions Virtualization* (NFV) as well as *Software-Defined Networking* (SDN) for easing network management and operations [1]; SNs form the basis of many use cases like cloud-, edge- or fog-computing, and many more [2]. The term *Federated Software Network* (FSN) describes a particular organizational form of SNs. FSNs are composed and operated by multiple sovereign and often geographically distributed (cf. [3]) parties with a common or complementary objective. Policies for the use of the federated infrastructure are determined in an IT service contract.

According to [4], operational characteristics and gaps of FSNs are, amongst others, an *immediate remote control* and *heterogeneity of components* just as *multiple intersecting administrative domains* which often miss out in current management platforms. This paper particularly addresses these gaps for functions and interfaces management in order to better cope with them in future network management platforms.

978-3-903176-32-4 © 2021 IFIP

Networks consisting of especially heterogeneous technologies are promising; consider, for instance, solely the differing advantages of hardware virtualization (security and isolation) and containerization (performance). And even these two forms of system virtualization are realized in a rather broad variety of implementations, e. g., KVM, Xen, Virtualbox, Docker, LXC, Kubernetes, and many more, each providing custom APIs, protocols, functions, and parameters. Moreover, the functional range may differ for each component version, aggravating a universal solution. And especially in federated networks, the need for a heterogeneous shared approach is more distinct, since each partner may require and introduce different kinds of components into the infrastructure. Yet, platforms for managing FSNs often leave a substantial amount of *manual* controlling efforts to the network managers by not fully providing this amount of heterogeneity.

Solving this problem is another footstep towards (e. g., rule- or policy-based) automated controls, and we believe that a normalisation of component functions sovereignly from their APIs is needed. For instance, Qemu just as LXC both provide a function which allows to *freeze* the state of a virtual instance, yet, in a different form. Policy- or rule-based controls (e. g., "*if malware_detected(vm) then freeze(vm) end*"), however, require this normalization of (e. g., for tokenizing) identical functions.

In order to overcome challenges in controlling FSNs and avoid another individual management solution, this paper provides the design and implementation of an easy-to-use framework for managing functions and interfaces of components and their virtual representation, which are called *Managed Object Classes* (MOC). A *Managed Object* (MO) is a managed component in the network. The design as a framework brings along three benefits: It is *a)* inherently extensible and *b)* provides sufficient flexibility for different use cases and it *c)* can be integrated into existing management platforms. Thus, proven functionality of existing management platforms can be expanded. A continuously improved prototype of our framework's key concepts is published at https://github.com/ubwmst/FEDCON. FEDCON bases upon *Lua*, which is designed to be embeddable in a broad spectrum

of run-time environments and serves well in networking tools like Wireshark or Nmap.

This paper's structure is organized as follows: In Section II, a simple yet realistic scenario illustrates a form of FSN for corporate data analysis. In Section III, we deduce requirements for the management of component functions and interfaces in FSNs, followed by a review of existing and similar approaches in Section IV. Section V describes the design of the framework itself. Section VI evaluates the framework against requirements; an evaluation showing its practical applications is performed in Section VII. Section VIII concludes this paper with our key findings and future work.

II. Scenario: Health Data Analysis Federation

FSNs are usually set up upon a cooperation of organisations. For instance, consider three health research institutes *Alpha*, *Beta*, and *Gamma*, collaborating in a joint health data analysis project. They plan to correlate huge data sets of several ten thousands of anonymized patients' data including age, various aspects concerning nutrition as well as physical and sports key parameters, just like the patient's anamnesis.

The institutes use a flexible federated infrastructure based upon SDN and virtualization (cf. Figure 1), which, in favor of data protection, allows the original health data to remain in each institute and only share evaluation functions and resources. Institute Alpha provides three KVM hypervisors, institute Beta provides a LXC containerization platform and institute Gamma supplies a slice of virtualized resources via a certain OpenNebula-managed project (i. e., authentication credentials) with limited resources. Each host provides an inventory of virtual resources (VMs).

The provided resources, located in the three distributed organization-internal data centers are connected via VPN; networking is virtualized and realized via OpenFlow- and OVSDB-enabled switches. The data centers have a locally deployed and different SDN-controller, e. g., OpenDaylight at institute Alpha and Floodlight at institutes Beta and Gamma.

All partners pursue a secure design of the federated infrastructure, hence only allowing partners to use absolute necessary functions of their provided infrastructure. That is, creating a virtual instance, monitoring, stopping, and restarting own VMs (e. g., Institute {Alpha, Beta, Gamma} can only monitor/stop/restart instances it previously created). Hence, administrative domains imply responsibilities for managing the FSN, which must be enforced via a technical implementation.

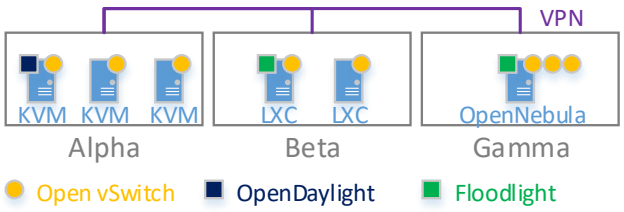

Fig. 1: A FSNs combining IT resources and partners in a health data analysis scenario.

III. Requirements in FSNs

We emphasize two groups of requirements: Those addressing *a)* the handling of component functions, and others addressing *b)* an approach's suitability in FSN, as summarized in Table I.

ID	Description
Requirements from API and functions handling	
R.1	Detailed modeling of component types and their APIs
R.2	Component management and assignment to component types
R.3	Normalization of component functions
R.4	En- and decoding of function parameters and results
R.5	Integrability of API clients
R.6	API authentication and transport security
Requirements for suiting FSNs	
R.7	Decentralized deployment
R.8	Cross-platform approach
R.9	Assistance in handling management domains
R.10	Provision of a multi-tenancy capable approach
R.11	Authorization and visibility functionality
R.12	Programmable arrangement of component functions
R.13	No impact on the host platform's state
R.14	No impact on the operational platform

TABLE I: Requirements for API and functions handling

A. API and Component Function Handling

Heterogeneity challenges from the paradigms in FSNs require a **[R.1]** grouping and modeling of components their functions in the network, which must be **[R.2]** allocatable to component types (e. g., an SDN-controller). Also versions must be considered since APIs differ across them. A normalization **[R.3]** of similar components is required in order to ease the integrability of novel components in FSNs: Since APIs of components not only vary in their kind of API paradigm (e. g., REST, RPC etc.) and protocols, but also in parameters and formats, **[R.4]** they must also be en- and decoded from/into a normalized form.

Another important aspect is the **[R.5]** provisioning and integrability of reusable communication clients. For instance, over-the-network-callable APIs like most RESTful services are based upon HTTP, SOAP relies on HTTP/XML, SNMP is UDP-based and product-specific interfaces also often exist. A related requirement is the system's ability to inherently provide different and adaptable **[R.6]** authentication mechanisms (e. g., HTTP Basic-authn).

B. Suitability in Federated Software Networks

Further important requirements address challenges from FSNs (cf. [4]), like a **[R.7]** decentralized, **[R.8]** cross-platform deployment due to the geographical distribution of physical resources. In order to handle multiple tenants, the resulting framework must **[R.9]** be able to assist in domain arrangement and cross-domain management. This also includes a remote manageability of components across data centers. Hence, not only must domains be considered in the framework, it should also provide **[R.10]** a multi-tenancy capable approach: The framework must provide **[R.11]** authorization and visibility functionality, which ensures that

authorized users can exclusively access and call functions in their responsibility. Automated management and control additionally requires a **[R.12]** programmable arrangability of the components' functions, i. e., allowing batch processing. From a security point of view, framework instances must not **[R.13]** impact or manipulate the state of its host platform, the communication must be handled in a controlled manner. But the framework instances themselves must **[R.14]** not be able to impact the underlying carrier system. It should, for instance, be running in a sandboxed environment.

IV. Related Work

In this matter, there are three relevant research areas: *(1)* network management platforms and their API handling, *(2)* API gateways, and *(3)* modeling approaches for APIs. Research projects working in the area of federated infrastructures are the European *FED4FIRE+* project, which strives to federate and openly provide multiple individual next generation internet (NGI) testbeds [24]. The *GENI* project (www.geni.net/) is a contentual counterpart operating in the US. Other recently completed research projects as *SENDATE* (https://www.sendate.eu/) or *NECOS* (www.h2020-necos.eu/) made efforts in order to realize and manage distributed infrastructures with a special focus on paradigms such as cloud computing, SDN, and NFV.

A. API management in (F)SN management platforms

Management platforms usually focus on a specific paradigm only and ignore a holistic FSN approach. From the SDN-perspective, *OpenDaylight* (ODL) as well as *ONOS* are two of the most sophisticated SDN controllers. A deep-dive analysis of both with a special focus on actual vulnerabilities and open issues is described in [7]. They both stand out due to their flexible approach based on Karaf and microservices [8] [9]. Both provide only limited functionality for federated networks – a decentralized clustering deployment, component management and security features. In contrast to ONOS, ODL provides a *model-driven approach* via YANG (cf. Section IV-C) for the description of APIs. ONOS in contrast provides a service for managed infrastructure device handling.

From the NFV paradigm, suchlike considered platforms belong to the ETSI NFV Management and Orchestration (MANO) layer: SONATA [10] also pursues a flexible approach. Here, especially the VIM-adaptors [11] have the most correspondence to our work. In SONATA, the implementation of MO functions is organized as Java/Python-based client modules. Similar more established approaches are Open-Source MANO (OSM) [12] and the Open Platform for NFV (OPNFV) [13]. These however, address supply frameworks for providing ETSI NFV-conform network architectures.

Most network federation approaches stem from the context of cloud computing. *BEACON* [5], *RESERVOIR* [15], *Contrail* [14], and *CYCLONE* [16] strive for a federated environment of cloud platforms (e. g., OpenStack, OpenNebula, and public clouds), yet – different from our approach – they usually assume the utilization of the federated cloud infrastructure from the perspective of a single user entity and restrict heterogeneity. We by contrast also consider organizationally challenging aspects like administrative domains and responsibilities in fully heterogeneous networks.

B. API Gateways

Systems for API management (also known as *API gateways*), like *Tyk* (https://tyk.io), *Gravitee* (https://gravitee.io/), *Kong* (https://konghq.com/kong/) face similar issues in API heterogeneity. These systems, however, are generally only focused on centralizing API endpoints and the provision of a unified endpoint with additional functionality like batch requests of APIs, individual authorization and policy-controlled access (e. g., calls per time unit, maximum allowed requests etc.).

C. API modeling and description

The *SNMP Management Framework* and SMIv2 [17] as de facto standards do not consider the definition of API endpoints. The *YANG Modeling Language* [18], a more contemporary approach, is limited to the RPC-based NETCONF protocol to manage NETCONF-enabled components and disregards REST-based APIs. Mature approaches for modeling REST-APIs are the *OpenAPI Specification* [19] and the *RESTful API Modeling Language* (RAML) [20]. They allow and are limited to an extensive description of REST/HTTP-based APIs and none of them stipulates a normalization of functions/APIs.

Several other publications cover this matter from different point of views: In [21], the authors present their modeling approach *EMF-REST*, which is based upon the Eclipse Modeling Framework, which generates Web APIs. In [22] the authors present a methodology for the development of APIs in network management. Therefore, based on an architecture models, interfaces for API development must be selected and described using UML for modeling REST and YANG. The authors of [23] approached the problem of heterogeneity of interfaces and functions using ontology modeling techniques with OWL for the SDN paradigm. Their approach extracts ontology objects and relationships from components' command line interfaces.

V. The FEDCON Framework

The purpose of this framework is to enable a coordinated access to functions of all MOs in an FSN, especially considering the heterogeneity problem; it strives to provide *a)* sufficient flexibility and *b)* a broad integrability.

A. Architecture

The architecture of FECON instances is shown in Figure 2. A FEDCON instance is deployed in each data center respectively (as we also proposed in a general management architecture in [25]). Hence, *Data Center Alpha* is controlled by instance FEDCON 1, *Data Center Beta* by instance FEDCON 2, and so on. The FEDCON instances form a peer-to-peer (P2P) network for coordination purposes like function call handling across data centers and synchronization. Consequently, single

point of failures, e. g., a master-slave architecture would entail, are eliminated, since the failure of one FEDCON node does not affect the remaining systems.

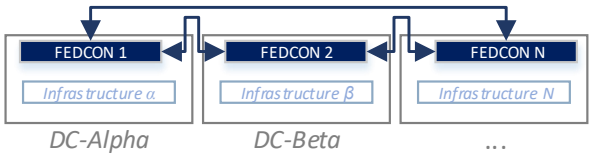

Fig. 2: Overview of the FEDCON framework architecture

B. Data Modeling

FEDCON introduces two new data structures for MOC modeling: The *MOCTypeModel* graph in combination with the *MOMapping* hash-map. These and several auxiliary structures are present in each FEDCON instance. Each element is, however, expandable or customisable, in order to fit multiple FSN scenarios.

1) MOCTypeModel: The MOCTypeModel is a tree-structure where nodes can have multiple parents. Nodes are MOCTypes with each having a functional description that can be inherited to children nodes.

Consider, for instance, the MOCTypeModel shown in Figure 3. Each node is a tree element, containing a unique identifier, a version (optional), parents, provided functions and an *abstract*-flag that indicates whether a node can be instantiated: *(identifier, version, parentList, functionList, abstract)*. If the *abstract* flag of a node is `true`, MOs cannot be assigned to this MOCType. It rather serves as a grouping node for function normalization of children nodes. For instance, in Figure 3, the node *sdncontroller* may describe the function set [`getVersion`], which is also inherited by *openflow-controller*, adding further functions [`addStaticFlow`, `deleteFlow`,...]. Node *ovsdb-controller* works analogous. *floodlight* is an *openflow-controller*, inheriting all its functions, but also adds additional functions. Floodlight then again has different versions, which extend previous version by additional function (e. g., Floodlight v1.2 introduces a function `getPortBandwidthConsumption` [6]).

Multiple inheritance is needed for MOs implementing functions from multiple distinct specifications (e. g., OpenDaylight is an OpenFlow as well as OVSDB-capable SDN controller).

A function in the FEDCON framework can be described by a tuple containing the *function ID*, an ordered list of ordinary data types describing *parameters* of the function and an ordered list of types of *return values* and a *driver* function (only for non-abstract nodes). These data types, for instance, encompass *string*, *integer*, *number*, *boolean* and a *list* type. The driver describes how to access the function of an MO. Also the first non-abstract node for each branch must implement a driver function for all inherited functions.

A valid driver function has the signature `driver(params, mo, ap)` $\rightarrow$ `ret`. Here, *params* are the actual values a function call; they must comply with the function definition. The driver function should make use of the attributes, which are delivered by the targeted managed object *mo*. The function returns a list of values *ret*, which corresponds to return values definition.

The parameter *ap* describes the access point of an API function; it is looked-up by the FEDCON framework itself: The auxiliary *APMap* structure manages alternative access points for certain API functions. This is needed for differing API access points (even for the same MOC). APMap is a map of entries for each entry having the shape *ffid* $\rightarrow$ *mocid* $\rightarrow$ *aap*, where (*ffid*) is the full function id having the form `MOCTypeID:MOCTypeVersion:FunctionID`, *mocid* is the actual MOCID of the differing API access points and *aap* is the alternative point of access (e. g., "`/api/v1-1/`"). For each function call, FEDCON checks if an alternative access point *aap* of *ffid* and returns it. Otherwise it uses the default access point from the MOMapping.

2) MOMapping: The MOMapping assigns unique MOIDs to an element in the MOCTypeModel. It also maps an MOID to the affiliated *data center ID*, data center internal network *IP address*, the *service port*, a table containing use case dependent unstructured authentication *credentials*, its *API access point*, default *access class* and (a list of) *access exceptions* (cf. Section V-B4). A MO can be added to arbitrary administrative domain in order to be able to describe management responsibilities (cf. next section). The assignment of the MOID to a node in the MOCTypeModel implies that the corresponding MO provides all functions from the assigned MOCType up to the root-node of the MOCTypeModel. Since an MO's MOCType may have multiple parents inheriting multiple functions, a function must be identified by its full function ID (FFID).

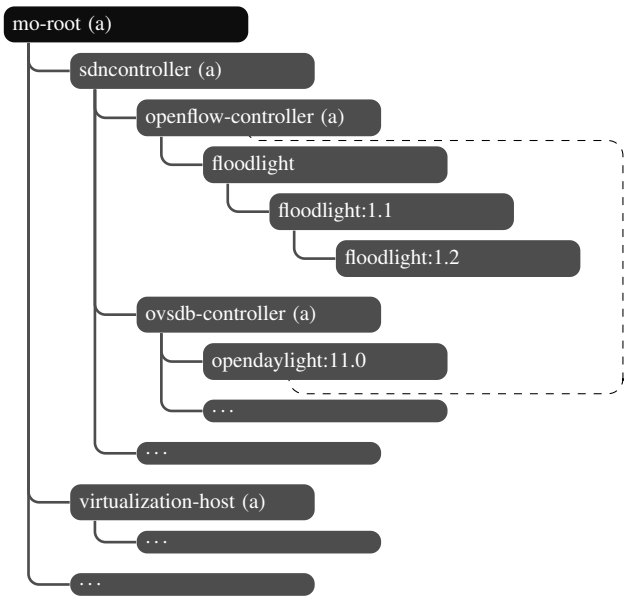

Fig. 3: Exemplary realization of the MOCTypeModel. Nodes marked with "(a)" are abstract.

3) Domain Structures: FEDCON inherently supports administrative domains, users and MOIDs can be assigned to. The actual handling (especially user authentication) of users must performed by the FEDCON instances' carrier application. Based on this information, access to MOC functions can be described in the AccessModel.

4) AccessModel: The *AccessModel* allows the modelling of authorization aspects for function calling. It introduces *access classes*, i. e., template-functions referring to custom implemented functions: *aclass → (tf: (user, mo, func) → Boolean)*. A template function is provided with *user*, *MO*, and *mo-function*. A developer can use these parameters as criteria for access determination (cf. example in Section VII-B).

Each MOCType defines a default access class which all its (and inherited) functions are assigned to. Individual assignments to different access classes can be done via *access exceptions [(ffid → access class)]*, as exemplarily shown in Listing 1. Object *acex* can be used MOMapping entries.

```
1 acex={["floodlight:1.1:addStaticFlow"] = free,
2       [...]}
```

Listing 1: Exemplary exceptional access class definition

C. Clustering and Synchronization

The distributed deployment of FEDCON instances requires synchronization of FEDCON instances and function call routing to a suitable FEDCON instance. All structures, *MOCTypeModel*, the *MOMapping*, the *AccessModel*, and *domains* are designed to be serializable, facilitating synchronization. Only *functions* (drivers, access class implementations) must be exchanged as file content – e. g., also via *git* or *subversion* – and interpreted at each FEDCON instance. The communication between the FEDCON instances must can be secured via TLS.

Functions of MOs defined in the MOMapping are callable from any FEDCON instance via function call routing to the appropriate instance/data center. Function routing is provided via the *ClusterEnvironmentModel*. It maps data center IDs (DCID) to the corresponding service IP address and service port of the responsible FEDCON instance (shown in Listing 2). By assigning the DCID to an MO, the responsible instance can be identified.

```
1 ClusterEnvironmentModel = {
2   dc1 = {ip = "192.168.55.5", port = 44544},
3   dc2 = {ip = "192.168.42.10", port = 44544},
4   [...]}
```

Listing 2: Exemplary ClusterEnvironmentModel

A function call forward contains the `user`, `MOID`, `MOCID`, `function ID` and `params` as a serialized object.

D. Host Application Integration

Due to the Lua-based implementation, any native application, just as many VM-based languages as Java or C# can serve as carrier for FEDCON.

Interfaces between a host application and the FEDCON framework are illustrated in Figure 4. FEDCON provides

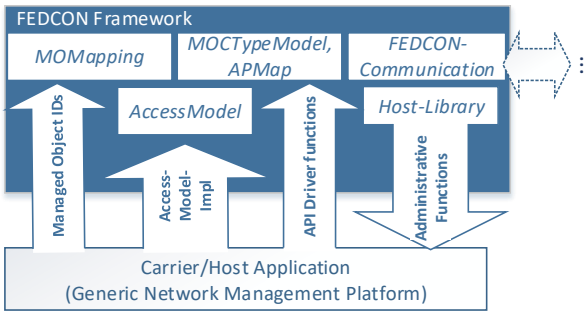

Fig. 4: FEDCON's interfaces towards carrier applications.

a library with administrative and operational functions to the host application. On the one hand, the host application provides FEDCON with MOIDs, an implementation of the AccessModel and driver functions for API access. FEDCON on the other hand, provides the host platform with functions for managing domains, users and MOCs and MOs as well as for calling APIs of MOs.

1) Access and Template Methods: On the one hand, administrative functions serve the initialization and handling of the FEDCON cluster as well as domains, users, and MOs for access control mechanisms. Also, the host application can retrieve the current AccessModel. Furthermore, the interface between the host application as well as the FEDCON framework allows the manipulation of the MOCTypeModel, i. e., adding, retrieving, and removing MOCTypes and functions as described in Section V-B1.

Third, the framework allows access to the MOMapping and itreturns MO objects to the host application, allowing them to use their functions and add them to domains.

The number and complexity of template methods are intentionally clear and cover the implementation of (1) drivers for functions and API calls, and the implementation of (2) access classes for the AccessModel.

The registration must be performed via the corresponding interfaces of the framework. Drivers must be created and registered in the context of the function-adding method for a certain MOCType (which can be retrieved via appropriate interfaces to the framework). The AccessModel must be implemented and submitted to the initialization of the framework.

VI. Requirements Evaluation

As summarized in Table II, FEDCON meets a broad range of requirements: The MOCTypeModel allows the description of models of network components (**R.1**), which can be reused for their instantiation (via the MOMapping **R.2**) and derivation of new models. The normalization of functions is one of FEDCONs key features (**R.3**) and is realized via inheritance of functions of one MOCType to all derived ones. Drivers provide sufficient flexibility to handle different technologies (e. g., RPC or REST) and secure transport (**R.6**), breaks them down in a unified method for parameter passing, function execution, and return value retrieval (**R.4**).

ID	Status	Statement
R.1	✓	Via the MOCTypeModel structure.
R.2	✓	Via the MOMapping structure.
R.3	✓	Provided by MOCTypeModel.
R.4	✓	Provided by *drivers*.
R.5	●	Not explicitly featured, but usable as Lua modules.
R.6	✓	Reusable modules and *drivers*.
R.7	✓	TLS-secured peer-to-peer architecture.
R.8	✓	Usable for many OS and RTEs.
R.9	✓	Domains are inherently provided.
R.10	✓	Controlled via *AccessModel*.
R.11	✓	Controlled via *AccessModel*.
R.12	✓	Functions can be used by host application.
R.13	✓	Unidirectional access to FEDCON.
R.14	●	Partially provided by Lua's sandbox mechanisms.

TABLE II: Requirements comparison (✓: fit; ●: partial fit)

FEDCON does not provide libraries for calling API methods, yet it uses Lua's libraries and modules for that (**R.5** partially). FEDCON's architecture is completely decentralized (**R.7**), P2P-based (avoiding single points of failures) and cross platform compatible (**R.8**). It provides administrative domains (**R.9**), which can be used for API access control mechanisms via the AccessModel (**R.11**) and multi-tenancy compliant function calls (**R.10**). Functions modeled in the MOCTypeModel are supplied to the host application by an internal API (**R.12**). The mode of access between FEDCON and a carrier application is to the greatest possible extent unidirectional from the carrier to FEDCON (**R.13**). It furthermore runs in a sandbox environment due to Lua's functionality to completely limit run-time capabilities by removing libraries and unnecessary functions (**R.14**). FEDCON, however needs improvements with regards to manipulation prevention due to its decentralized design. We address this in our future works.

VII. Practical Evaluation

A practical evaluation was conducted in a laboratory environment, modeling several characteristic yet heterogeneous FSN components (cf. Figure 5) using *Lua 5.3* (in the following Listings instantiated in object `fc`). With our evaluation by example, we especially emphasize the modeling aspects as FEDCON's main novelty we described in this paper. The evaluation is also a representative workflow showing how to use FEDCON.

A. Laboratory Setup

Data centers *Alpha* and *Beta* form two administrative domains *domAlpha* and *domBeta*, operated by users *aAdmin* and *bAdmin* respectively. This setup in FEDCON instance *FEDCON-A* can be done via its `init` function (FEDCON-B works analogously), as shown in Listing 3, using an a priori implemented AccessModel `currentAM` (cf. Section VII-B).

Data centers *Alpha* and *Beta* run commercial-off-the-shelf hardware, providing virtualization via miniONE, an OpenNebula-based LXC quick-setup system as a full-fledged management and orchestration platform and LXC (*Layer 0*).

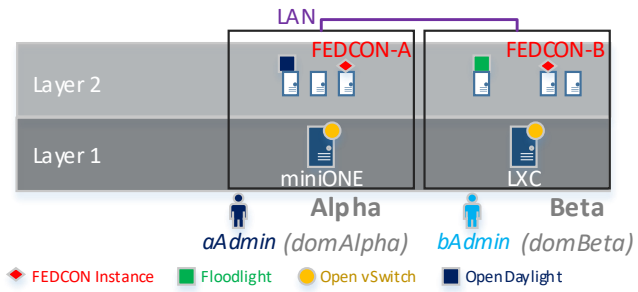

Fig. 5: Overview of the evaluation setup.

Each host furthermore runs an *Open vSwitch* (OvS) instance for inter-VM-communication. The switches are interconnected and controlled by OpenDaylight (ODL) or a Floodlight.

```
fc = require("fedcon")({
  thisDC = "alpha",
  myIP = "192.168.111.5",
  myPort = 5555,
  [..] -- cert options
  -- driver functions directory
  driverDir = "drivers/",
  -- implemented in following sections
  accessmodel = currentAM});

  fc.register("beta", "192.168.130.5", 44544)
  fc.addDomain("domAlpha"):addUser("aAdmin")
  fc.addDomain("domBeta"):addUser("bAdmin")
```

Listing 3: Initializing instance FECON-A.

We decided to operate from a plain native application for Linux serving as a *baseline* carrier for the FEDCON framework.

B. AccessModel Implementation

Our exemplary access model for this evaluation can classify functions that as *a)* free to use (access class *free*) or *b)* usable if users and MO share a domain (access class *inDomain*). The actual implementation of the AccessModel is shown in Listing 4.

```
currentAM = {
  free = function (u,m,f) return true end,
  inDomain = function (u,m,f)
    local domains = m:getDomains()
    for _,d in ipairs(domains) do
      if (d:checkUser(u)) then return true end
    end
    return false
  end
}
```

Listing 4: Laboratory AccessModel implementation.

C. MOCType Modeling for OpenNebula

In our setup, models are required for miniONE, the LXC instance, and both SDN controllers. Switches as the OvS instances are by paradigm controlled indirectly via the SDN

controllers and are not considered here. The resulting MOCTypeModel is shown in Figure 6.

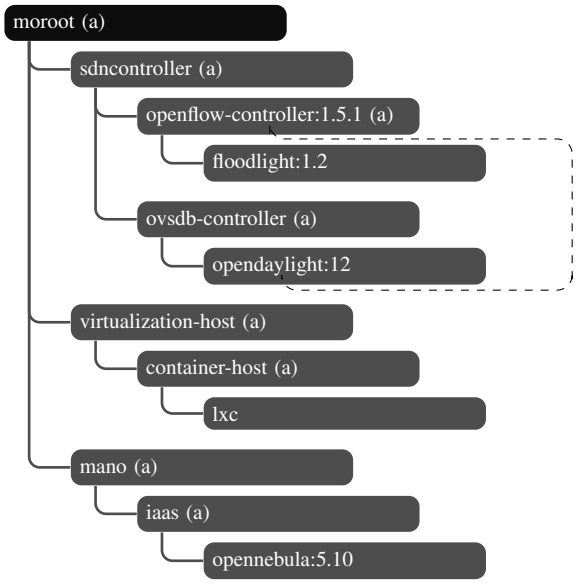

Fig. 6: MOCTypeModel for our evaluation setup. Nodes marked with "(a)" are abstract.

Node miniONE belongs to a group of IaaS-systems which can be classified as MANO systems. In this paper MANO systems are systems that manage multiple resources like hypervisors and storage nodes. Hence, the node *mano* describes MANO-specific functions, e. g., `addcompnode`, `addstoragenode` etc. The node *iaas* inherits and expands them by IaaS-typical functions like `vmboot` and `vmpoweroff`. The node *opennebula* in turn inherits all functions from *iaas* and extends them by ONE-specific functions like market place management functionality `marketupdate`, `marketshow` etc. [26]

```
fc.addMOCType("sdncontroller", "", {"mo-root"}, true)
fc.addMOCType("openflow-controller", "1.5.1", {"sdncontroller"}, true)
fc.addMOCType("floodlight", "1.2", {"openflow-controller"}, false)
fc.addMOCType("ovsdb-controller", "", {"sdncontroller"}, true)
fc.addMOCType("opendaylight", "12", {"ovsdb-controller", "openflow-controller"}, false)
-- flow modification function
-- params: DPID, table-id, flow-id, content
fc.getMOCType("openflow-controller:1.5.1"):addFunction ("addFlow", {"string", "number", "number", "string"}, {"boolean"})
```

Listing 5: MOCTypeModel for ODL with exempl. function.

D. MOCType Modeling for OpenDaylight

We explicitly demonstrate the modeling of multiple parents for the *opendaylight:12* node. The direct predecessor is the abstract node *ovsdb-controller*, which itself extends the abstract node *sdn-controller*. Since ODL also supports OpenFlow, node *openflow-controller:1.5.1* is its secondary parent. Crucial parts are shown in Listing 5.

In order to break down complexity of the OpenFlow protocol we implemented specific functions for the class *openflow-controller*; i. e., instead of modeling a general function `addFlow`, we distinguish between operations with MAC addresses, IP addresses, port-based operations etc.

E. Driver Implementation for OpenNebula

We exemplarily set up the node's *iaas* function `vmboot` (named *resume* in ONE): We add a corresponding entry (cf. Listing 6) which requires the ID of the VM that needs to be started as only parameter and returns a boolean value indicating the operation's success. For the sake of brevity, the implementation of the driver rather simple and pragmatic. The XML-encoded RPC call is composed via Lua's *formatstring* method; it replaces parameters like username and password from the MOMapping's object's credentials table and the only parameter in *params*. `httppost` (implemented based on Lua's socket library) handles the RPC call, after the point of access is composed. Finally, the driver function is registered to the original MOCType *iaas* and the MO with ID *miniONEd1*.

```
-- module name: miniONEd1.lua
function oneVMBootDriver (params, mo, ap)
-- assemble RPC call
local xmlreq = string.format([[
  <?xml version="1.0" ?>
  <methodCall>
  <methodName>one.vm.action</methodName>
  <params>
  [..] -- params substitution
  </params>
  ]],
  mo.cred.user, mo.cred.passwd, params[1])
  -- execute RPC call with httppost
  local ret = httppost(
    -- assemble point of access
    string.format("http://%s:%d/%s",
    mo.ip, mo.sport, ap),
    xmlreq)
  -- return false on error code
  -- otherwise return true
  return ret and not ret:match("Error")
    or (false, ret)
  end

  -- register driver for function "vmboot"
  fc.getMOCType("opennebula:5.10"):addDriver("
      iaas:", "vmboot", "miniONEd1")
```

Listing 6: Driver for function `vmboot`.

```
-- ffid, moctype, alternative access point
fc.addAPMapEntry("openflow-controller::addFlow", "opendaylight:12", "/restconf/config/opendaylight-inventory:nodes/node/openflow:<DPID>/table/<TABID>/flow/<FLID>")
```

Listing 7: Alternative point of access for function `addFlow` with placeholders `<DPID>`, `<TABID>`, and `<FLID>`.

F. Driver Implementation for OpenDaylight

For the node *opendaylight:5.10* we highlight the multi inheritance aspect and model the function `addFlowMAC` from its secondary *openflow-controller* parent node. ODL has a REST-based API where the request body and the HTTP resource path of the point of access must be adapted. Also in ODL the API access point is divided into a *config* and an *operational* access point. Hence, an alternative access point entry in the *APMap* is needed (see Listing 7). In the HTTP resource path, <*DPID*> is the data path ID, <*TABID*> is the flow-table id and <*FLID*> is flow ID supposed to be installed. These placeholders are replaced by the actual parameters (via function `gsub`) in the driver implementation (see Listing 8).

```
1  -- module name: odld1.lua in drivers directory
2  function odlAddFlowDriver (params, mo, ap)
3    -- assemble resource path
4    local apiap = string.gsub(ap,
5      "<DPID>", params[1]):gsub("<TABID>",
           params[2]):gsub("<FLID>", params[3])
6    -- [..] function call and result handling
7  end
8  -- driver registration
```

Listing 8: Driver for function `addFlow`.

G. Mapping MOs to their corresponding MOCType

After modeling in the previous sections, we show the *instantiation* of MOs according to MOCTypes via the MOMapping in Listing 9 for an ODL instance with MOID *odlight1*. This example also shows the usage of exceptional access classes, assigning the function `opendaylight:12:network-topology` to access class *free*. Default access class is *inDomain*. Also authentication credentials of the MO are considered.

```
1  -- add domain domBeta with user bAdmin
2  fc.addDomain("domBeta"):addUser("bAdmin")
3
4  -- Map MOID "odlight1" to node opendaylight:12
5  -- last parameter adds access class exception
6  fc.addMo("odlight1", "192.168.130.15", 8181,
7  "opendaylight:12", "restconf", "beta",
8  {user="admin", passw="l33tpw!"},
9  "inDomain", {["opendaylight:12:network-
       topology"] = "free"})
10
11 -- add MO odlight1 to domain domBeta
12 fc.getDomain("domBeta").addMO("odlight1")
```

Listing 9: Set up dom *domBeta* for *bAdmin* and MO *odlight1*.

H. Function Calling Example and Workflow

Defined functions of FSNs' components can be used from the FEDCON carrier application. Consider calling function `addFlow` on MO *odlight1*, triggered by user *aAdmin* at instance FEDCON-A (cf. Listing 10). As described in Section V-B3, user authentication is performed by the carrier application; it passes the user ID as first parameter to the function `fc.getMo(MOID):exec`. User *aAdmin* can call the function in the local data center, since the user is authorized as she shares the domain *domAlpha* with MO *odlight1*. Assuming user *aAdmin* triggers the function call at FEDCON instance FEDCON-B, the function call would be forwarded to the relevant instance FEDCON-A, which is performed according to the ClusterEnvironmentModel. Yet, the remote calling functionality is explicitly not part of the implementation of this paper but of our future works.

```
local ret = fc.getMo("odlight1"):exec("aAdmin"
    , "openflow-controller:1.5.1:addFlow", {"1
    ", 5, 15, "<flow> [...] </flow>"})
```

Listing 10: Function call `addFlow` on MO *odlight1*.

I. Findings

Our research led us to two key findings: (1) The component type structure of FSNs is very suitable for exploiting inheritance effects between components. As FSNs already inherently predefine some aspects (e. g., a classification of SDN-components etc.), others must fit to the corresponding managed infrastructure (e. g., multi-purpose components like ODL). Current management platforms hardly exploit these dependencies and inhibit automation. (2) Management platform must be more flexible in federated infrastructures in terms of organizational aspects. We provided this amount of flexibility by combining multiple adaptable, yet minimal framework designs to fit the needs of a specific managed federated infrastructure.

VIII. Conclusion and Future Work

In this paper, we presented the embeddable FEDCON framework for augmenting existing management platforms for FSNs. It provides access to components' APIs and functions in FSNs. We introduced the MOCTypeModel and APMap structures, allowing the inheritance-based modeling of MOCs' APIs, fostering transparency and automation. The MOMapping *instantiates* MOs by assigning MOIDs to a node in the MOCTypeModel. Organizational aspects are described via FEDCON's *ClusterEnvironmentModel* and the adaptable AccessModel for use case suited access control. Our evaluation emphasizes on the modeling parts via an implementation and realisation of a representative environment. Our research recommends the exploitation of the FSN components structure for modeling, as well as framework designs in network management platforms, which leverage automation and use case dependent customisations.

In our future work, we want to improve the usability of our framework, e. g., by methods for importing existing approaches like YANG, OpenAPI, or RAML. Furthermore, we want to put more research into improving models for the mentioned inheritance-based dependencies in FSNs' components, also for MOCs' attributes. Finally, besides already elaborated core concepts for FEDCON in this paper, we are going to implement yet missing parts, especially concerning the management of distributed FEDCON instances, as well as hardening. These will be evaluated under consideration of overhead in processing, storage and communication.

REFERENCES

[1] Kellerer, Wolfgang and Chemouil, Prosper and Kamiyama, Noriaki and Martini, Barbara and Pasquini, Rafael and Schembra, Giovanni and Schmid, Stefan and Zhani, Mohamed Faten and Zinner, Thomas, "Guest Editorial: Special Issue on Latest Developments for the Management of Softwarized Networks", Transactions on Network and Service Management. IEEE, 2019.

[2] Maenhaut, Pieter-Jan and Volckaert, Bruno and Ongenae, Veerle and De Turck, Filip, "Resource Management in a Containerized Cloud: Status and Challenges", Journal of Network and Systems Management 28.2. Springer, 2020.

[3] He, Mu and Huang, Mei-Yuan and Kellerer, Wolfgang, "Optimizing the Flexibility of SDN Control Plane", IEEE/IFIP Network Operations and Management Symposium. IEEE, 2020.

[4] M. Steinke, W. Hommel, "Overcoming network and security management platform gaps in federated software networks", 14th International Conference on Network and Service Management (CNSM). IEEE, 2018.

[5] R. Moreno-Vozmediano, E. Huedo, M. Ignacio, R, Montero, P. Massonet, M. Villari, G. Merlino, A. Celesti, A. Levin, L. Schour, "BEACON: A cloud network federation framework", European Conference on Service-Oriented and Cloud Computing. Springer, 2015.

[6] Ryan Izard, "Floodlight v1.2", https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/24805419/Floodlight+v1.2, 2016.

[7] Vizarreta, Petra and Trivedi, Kishor and Mendiratta, Veena and Kellerer, Wolfgang and Mas Machuca, Carmen, "DASON: Dependability Assessment Framework for Imperfect Distributed SDN Implementations", Transactions on Network and Service Management. IEEE, 2020.

[8] Administrator and M. Thengvall, "ONOS", https://wiki.onosproject.org/, 2019.

[9] OpenDaylight Project, "OpenDaylight Karaf Features", https://docs.opendaylight.org/en/stable-boron/getting-started-guide/karaf_features.html, 2016.

[10] Soenen, Thomas and Van Rossem, Steven, et. al, "Insights from SONATA: Implementing and integrating a microservice-based NFV service platform with a DevOps methodology", NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, 2018.

[11] Soenen, Thomas, "VIM Adaptor", https://github.com/sonata-nfv/son-sp-infrabstract/tree/master/vim-adaptor, 2018.

[12] Israel, Adam and Tierno Sepúlveda, Alfonso, et. al, "OSM Release FIVE Technical Overview", https://osm.etsi.org/images/OSM-Whitepaper-TechContent-ReleaseFIVE-FINAL.pdf. 2019.

[13] OPNFV Project, "Technical Overview", https://www.opnfv.org/software/technical-overview. 2018.

[14] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, G. Righetti, "Cloud Federations in Contrail", European Conference on Parallel Processing. Springer, Berlin, Heidelberg, 2011.

[15] Rochwerger, Benny, et al. "The reservoir model and architecture for open federated cloud computing", IBM Journal of Research and Development 53.4 (2009): 4-1.

[16] Slawik, Mathias, et al. "CYCLONE unified deployment and management of federated, multi-cloud applications", 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC). IEEE, 2015.

[17] K. McCloghrie and D. Perkins and J. Schoenwaelder, "Structure of Management Information Version 2 (SMIv2)", https://tools.ietf.org/html/rfc2578, April 1999.

[18] M. Bjorklund, "The YANG 1.1 Data Modeling Language", https://tools.ietf.org/html/rfc7950, August 2016.

[19] SmartBear Software, "OpenAPI Specification Version 3.0.3", https://swagger.io/specification/, February 2020.

[20] C. Heidenreich and K. Hahn and J.Stoikovitch et al., "RAML Version 1.0: RESTful API Modeling Language", https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/, June 2019.

[21] Ed-Douibi, Hamza and Izquierdo, Javier et al., "EMF-REST: generation of RESTful APIs from models", Proceedings of the 31st Annual ACM Symposium on Applied Computing, 2016.

[22] Pugaczewski, Jack and Cummings, Tara and Hunter, Donald and Michalik, Bartosz, "Software engineering methodology for development of APIs for network management using the MEF LSO framework", IEEE Communications Standards Magazine, 2017.

[23] Martinez, A and Yannuzzi, Marcelo et al., "An Ontology-Based Information Extraction System for bridging the configuration gap in hybrid SDN environments", 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2015.

[24] FED4FIRE+ Consortium, "About FED4FIRE+", https://www.fed4fire.eu/the-project/, 2020.

[25] Steinke, Michael and Hommel, Wolfgang, "A data model for federated network and security management information exchange in inter-organizational it service infrastructures", NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, 2018.

[26] OpenNebula Project, "XML-RPC API", http://docs.opennebula.io/5.10/integration/system_interfaces/api.html#onevm, 2020.

[27] OpenDaylight Project, "OpenFlow Plugin Project User Guide", https://docs.opendaylight.org/en/stable-neon/user-guide/openflow-plugin-project-user-guide.html, 2018.