Engineering Heterogeneous Internet of Things Applications: From Models to Code

Thomas W. Pusztai, Christos Tsigkanos, and Schahram Dustdar
Distributed Systems Group, TU Wien
Vienna, Austria

***Abstract*—Applications on top of the Internet of Things (IoT) show big potential, but the domain raises challenges for software engineers. Heterogeneous hardware environments and software stacks, unreliable devices, and diverse developer skillsets make the implementation of business processes spanning the entire application and the enforcement of constraints difficult. To this end, we propose a methodology and technical framework targeting heterogeneous IoT systems, where software components may be situated on IoT devices, cloud infrastructure, or edge devices – a paradigm often referred to as fog computing. Our approach leverages design-time modeling of device interfaces, data, and business processes. Design-time activities entail specification of the software architecture of the IoT application in an implementation- and language- agnostic manner. Subsequently, we automatically generate boilerplate code for participating devices, requiring developers to only implement business processes. The generated boilerplate code is correct by construction and it targets the different languages usually involved in diverse IoT software components. When the application is deployed, its execution may violate design assumptions. To counter this, constraints defined during design time are enforced at runtime, ensuring that devices operate within certain bounds. We evaluate our model-driven engineering framework over a health care system use case.**

***Index Terms*—Internet of Things, Fog computing, Model-Driven Engineering, Code Generation, UML**

I. Introduction

Contemporary pervasive systems effectively integrate heterogeneous devices, computing infrastructure and cloud services. The resulting Internet of Things (IoT) applications can be of varying types and complexities, with multiple heterogeneous components expected to satisfy design objectives in a collaborative manner.

Managing, configuring and deploying complex IoT applications are emerging problems, where multiple software components may be situated in different devices, utilizing non-local computing or data resources and communicating over the network. This is exacerbated in what has been defined as *fog computing* for the Internet of Things [1], where interconnected devices with greatly differing hardware and software characteristics, typically interact in a manner characterized by absence of centralized control structures or data sources. Engineering such applications can make use of powerful model conceptions and accompanying software tools, allowing application designers to first reason on a well-defined architecture representation of their application and subsequently use the architectural model to support development.

Research partially supported by the TU Wien Research Cluster SmartCT.

The domain of fog computing presents special challenges for engineering software applications [2]–[4]. Firstly, the heterogeneity prevalent in devices and software stacks means that devices may range from very resource constrained (e.g., microcontrollers) to quite powerful (e.g., smartphones); yet they all need to collaborate and coordinate [5] in a single application. Resource constraints on devices need to elaborate data or control, in a system characterized by dynamicity, as participating devices may join or leave within application operation. This includes failures, as components and devices may cease working without warning or a communication link may drop. Security and privacy constraints are another concern, something exacerbated by the fact that different components on different devices may need to implement security controls or protect sensitive data [6]. Developers of such applications would ideally need to be experts in handling all of them and, in addition, be experts of the target domain of the application itself, e.g., health care or industrial smart power grids. The developer of a microcontroller-based sensor might not be interested in the global architecture of the fog application that it is going to be used in; similarly, the power grid engineer designing a smart grid application might not know how to handle unreliable communication between devices.

Our fundamental intuition is that a complex IoT fog computing application – as a software artifact itself – is a set of components and connectors [7], which due to the IoT target domain may be deployed, transferred, or make use of non-local computing or data resources over the network. A component is a software artifact logically situated in a device, responsible for some functionality that can be independently developed and delivered as a unit and that can be composed unchanged, with other components to build something larger. Typically, in a fog computing application, a component is responsible for some functionality with code executed on some device, making deployment a key issue. Connectors are interaction mechanisms between components, supporting some workflow; often in fog computing, web services are used.

We advocate a model-driven engineering (MDE) approach consisting of both design-time and runtime facilities to deal with these challenges. At design-time, the system architect designs the system as a set of models, which can then be used to generate *boilerplate code* that defines the principal interfaces and components of the system. This boilerplate code is

978-1-7281-6739-8/19/$31.00 ©2019 IEEE
DOI 10.1109/CIC48465.2019.00036

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

the basis upon which application development can commence. Models defined at design-time may also be used at runtime, to monitor and enable reasoning about the satisfaction of system requirements. Given defined, well-designed models, a middleware that controls the execution of the defined processes and enforces certain defined constraints can be generated, and situated at runtime along the running system. The MDE approach we advocate, is also related to software maintenance – new developments or changes in requirements may require the replacement of a component with another that provides a different interface and behaves differently. With model-driven development this can be carried out with little disruption to the system.

In this paper, we propose a methodology and technical framework for model-driven engineering of fog computing applications. Our approach targets heterogeneous IoT systems, where software components may be situated on IoT devices, cloud infrastructure, or edge devices. Our approach leverages design-time modeling of (i) device interfaces and data, (ii) business processes, and (iii) overall application execution constraints that may be involved. This design-time activity essentially entails specification of the software architecture of the fog application in an implementation- and language- agnostic manner. After the architecture has been modeled, we automatically generate boilerplate code for all different software components within it, requiring developers to implement only device-specific details and naturally, the business processes. The generated boilerplate code is correct by construction and is able to target the different languages usually used in diverse IoT software components. Developers may complete the fog computing application components independently. When the application is completed and deployed, its execution may violate design assumptions. To this end, constraints defined during design-time are enforced at runtime; constraints are obtained from the model specification of the fog application, and during code generation are converted to runtime assertions. As such, enforcement ensures that the fog devices operate within defined bounds.

Our concrete contributions lie within model-based engineering of fog computing applications and are as follows.

- We provide a methodology where device REST interfaces and data in a fog application are modeled with UML class diagrams, and business processes within it with UML activity diagrams. Those record the application's architecture and constraints on its behavior at runtime.
- Model-to-code facilities give rise to a technical framework *FogUML2Code*[1], able to generate correct-by-construction boilerplate code from the models defined. The generated artifact captures the fog computing architecture, is able to target heterogeneous implementation languages used throughout the application, and is completed by application developers.
- Execution constraints added to the UML model specified with the Object Constraint Language (OCL) are utilized to generate runtime assertions, which are integrated directly into the generated code.
- Finally, we provide a runtime library, *fog-execution-framework-java*[2], for executing business processes defined in UML activity diagrams.

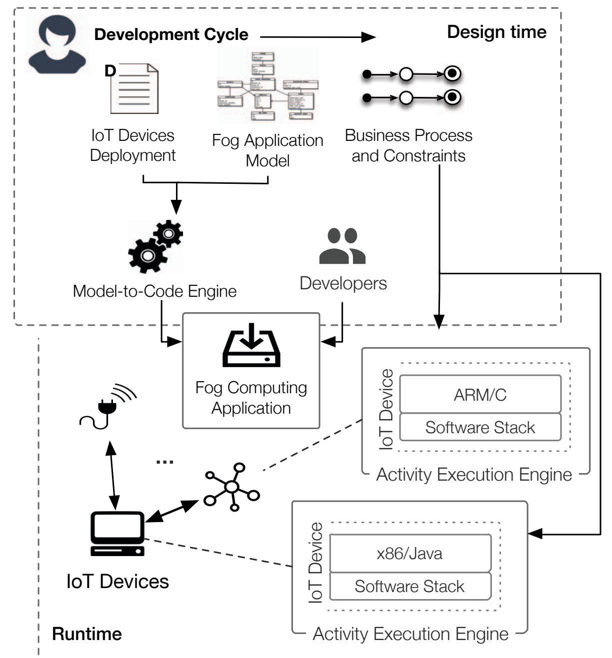

Fig. 1. Model-based Engineering of Fog Computing Applications: Overview.

The rest of the paper is structured as follows. Section II gives an overview of our approach within fog application engineering as well as illustrating a motivating example. Section III describes key methodological aspects and design activities of our approach. Section IV illustrates operational aspects, including model-to-code generation and activity realization. Section V provides an assessment of applicability over a health care fog application use case; related work is considered in Section V-C, and Section VI concludes the paper.

II. OVERVIEW

We advocate a model-driven engineering approach consisting of both design-time and runtime facilities; Figure 1 provides a bird's-eye view of our approach. At design-time, the system architect designs the system as a set of models, defining principal interfaces and components of the system, taking into account the overall business processes as well as the target IoT devices where the application should be deployed. Subsequently, boilerplate code is automatically generated for every software component modeled in the system; components –to be deployed on different IoT devices– may utilize different software libraries and be written in different programming languages. The generated code is tailored to the capabilities of each hardware platform and abstracts away details of initializing the modeled services, allowing developers to focus on

[1]https://github.com/fog-uml-2-code/fog-uml-2-code

[2]https://github.com/fog-uml-2-code/fog-execution-framework-java

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

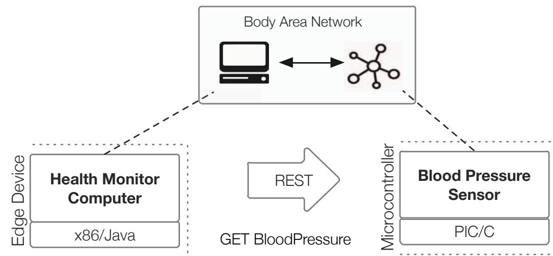

Fig. 2. Interacting components on devices within a Body Area Network.

the business logic of each component. Moreover, components may be built independently by software developers.

At system runtime, the application is deployed on IoT devices. Some components provide REST services to other components, while others execute one of the modeled business processes using an activity execution engine. Models defined at design-time may also be used at runtime [8], to monitor and enable reasoning about the satisfaction of system requirements. Constraints obtained from the UML specification of the fog application are converted to runtime assertions and enforced at system operation, ensuring that the application behaves within the defined bounds. Design and development using our MDE approach is essentially a four step process:

1) *Design Data Model & Interfaces*: Definition of key interfaces and components as well as constraints they should adhere to in operation; it thus consists of two sub-steps: a) Data model and device interfaces design with UML class diagrams, and b) Definition of constraints on data and operations using OCL;
2) *Processes Design*: Specification of the processes of the fog application with UML activity diagrams;
3) *Models-to-Code Generation*: Automatic generation of boilerplate code for each device and process, tailored to the respective hard- and software platforms;
4) *Details Implementation*: Manual implementation of the business processes.

Steps one and two will be discussed in detail in Section III, while steps three and four will be elaborated in Section IV.

A. Motivating Example

The rapid growth in sensor technologies, low-power integrated circuits, and wireless communication has enabled a new generation of wireless sensor networks for medical uses. Such Body Area Networks (BANs) [9] involve multiple, possibly heterogeneous and interconnected devices upon which various software-intensive data and control operations take place. In Figure II-A a simplistic BAN is illustrated, consisting of a *health monitor* and a *blood pressure sensor*, which could be used for 24-hour ambulatory blood pressure monitoring of a patient; a more complex scenario will be presented in Section V-A.

The *blood pressure sensor* is based on a low-powered microcontroller, running C code and exposing its functionality through a REST interface. The *health monitor* is an edge device connected to to the *blood pressure sensor* through the BAN and to the Internet or a hospital's private network through WiFi. It is a more powerful device than the sensor and runs Java, leading to a heterogeneous system with two devices having different performance and programmability characteristics. The *health monitor* issues a GET request to the *blood pressure sensor's* REST interface at regular intervals to check the blood pressure of the patient and logs the data. If the blood pressure reaches a critical level, medical personnel is automatically informed, illustrating a runtime decision.

Notice that the simple BAN presented is an instance of the IoT; several sensing or actuating devices are needed to realize it, which are connected over networks with programming interfaces. Engineering of an application within such an IoT domain, involves (i) architectural aspects, such as correct design of the involved components and connectors, as well as (ii) ensuring that execution behavior of the system does not violate certain bounds (e.g. maximum number of blood pressure checks per hour) and adheres to the defined specification. It is then evident that supporting systematic engineering of such applications is required; high-level reasoning can occur at the model level, something especially important in critical domains such as medical IoT.

III. Design

In this section the steps of *Design Data Model & Interfaces* and *Design Processes* from the methodology outlined in the previous section are examined. As development platform we adopt Eclipse Modeling Framework (EMF), because of its widespread use in the MDE community. It provides support for UML2 and OCL through their implementations within Model Development Tools (MDT)[3]. MDT provides EMF metamodels and libraries, but no graphical editors for UML diagrams (apart from a rudimentary tree-based editor). For graphically creating UML diagrams *UML Designer*[4] was adopted, but any UML modeling tool that targets Eclipse's UML2 meta-model can be used.

A. Design Data Model & Interfaces

The first step requires the software architect to design the data model that will be used across the fog application, as well as the REST interfaces of the involved IoT devices using a UML2 class diagram [10]. REST is very common in the fog/IoT domain and microservices have emerged as a widely used architectural conception. Microservices inherent in our approach (i.e., over REST), are applicable to a wide range of platforms in fog computing: from miniscule devices to cloud infrastructures. A class diagram captures the data structures and REST endpoints in a compact manner. Each interface in the diagram will be treated as a device's REST endpoint. After designing the data model and REST interfaces, the architect can define constraints on data and methods using OCL [11]. The architect may specify class invariants, as well as pre- and post-conditions for each method of the REST interfaces.

[3]https://www.eclipse.org/modeling/mdt
[4]http://www.umldesigner.org

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

1) Fog UML Profile: In order to allow a software architect to apply fog specific semantics to the elements of a class diagram, we provide a tailored UML2 `Fog` profile, consisting of stereotypes and some helper elements. Stereotypes are used during the code generation process to gain information on how elements should be treated - they are: `DataModel`, `FogDevice`, and `ActivityRealization`. The `DataModel` stereotype is used to designate a class or a package as being part of the data model of the fog application. In the following, we discuss the `FogDevice` and `ActivityRealization` stereotypes in detail.

The `FogDevice` stereotype informs the code generator that the interface, class, or package, to which it has been applied, will be deployed on a fog device. Setting specific properties on an application of this stereotype influences the code generation for these UML elements. The most important property of a `FogDevice` stereotype application is `targetLanguage`, which defines the programming language that will be used to generate code for this device. It must be set to a member of the `TargetLanguage` enumeration. Currently C and Java are supported, but other languages can be added in the future. Other properties available in the `FogDevice` stereotype are: `cpu` (CPU types, like AVR-8bit or ARM-32bit), `cpuCores`, `cpuClockMHz`, `eepromKb`, `ramKb`, `flashStorageKb`, and `batteryMAh`. In our prototype these other properties do not influence the code generation, but they may be useful in the future - e.g., the CPU type could be used to automatically generate some defaults for the compiler settings.

The `ActivityRealization` stereotype designates that a package and all contained elements are part of the realization of one of the fog application processes, modeled using activity diagrams in the *Design Processes* step. It has one property, `realizedActivity`, which must be set to the activity diagram that models the process, in whose realization the stereotyped package will be part of. Each class contained in this package will be generated as a *service class*. Services classes provide methods that can be referenced and thus called from activity diagrams. The `ActivityRealization` stereotype is designed to be used in conjunction with `FogDevice` on the same package. The latter is used to set the target programming language for the activity realization and other information about the fog device that will host the activity's process.

2) Class Diagram Design: A fog application design commences with the creation of a new UML model using Eclipse MDT and the application of the `Fog` profile to it. The design of a class diagram for a fog application is very similar to that of a class diagram for any other application, although there are some important differences:

- The `Fog` UML profile's stereotypes need to be used;
- Interfaces play a special role, due to the domain;
- Classes must either be part of the `DataModel`, implement an interface, or be part of an `ActivityRealization`.

The stereotypes described in Section III-A1 need to be applied to appropriately classify some of the model elements and to customize the code generation. Each interface in the UML model will be treated as a REST interface in the fog application. This means that during code generation both a server-side implementation (for the fog device that hosts the REST endpoint) and a client-side implementation (for the activities that call that REST endpoint) will be generated. For simplicity, each interface method that takes no parameters is considered as a GET endpoint, and each method with parameters is considered as a POST endpoint. Parameters and return values are assumed to be passed as JSON objects.

If an interface is extended by one or more subinterfaces, code will only be generated for the interface(s) at the bottom of the inheritance hierarchy; for superinterfaces nothing will be generated. For generating REST client code only the interfaces are of importance, but for generating code that implements the REST endpoint (i.e., the server-side), any classes that realize an interface are considered as well. If an interface is not realized by any class in the UML model, a class (or an equivalent for non object-oriented languages) that implements this interface will be generated as the REST endpoint controller. If an interface is realized by a class in the model, that class will be generated, including all additional attributes and methods.

For a class in the UML model to be considered during code generation, it must either:

1) Have the `DataModel` stereotype applied, or be part of a package with that stereotype (the class will be treated as a model class);
2) Implement an interface (the class will be treated as a REST endpoint controller), or
3) Be part of an `ActivityRealization` (the class will be treated as a service class, callable from the activity).

3) Define Constraints: After specifying the data model and REST interfaces, the software architect may add constraints to the UML model using OCL. Constraints may be added to all interfaces, classes, and their methods in a class diagram and they may be of the following types: *invariants*, if they are added directly to a class or interface, or *pre-* or *post-conditions*, if they are added to a method. Within our approach, adding OCL constraints occurs through the tree-based UML model editor that is part of MDT.

These constraints play a vital role in the design of the fog application, because during code generation they will be converted to assertions that will be executed at runtime. This enforcement of the defined constraints is especially important in a fog application, because it helps ensure that the fog devices operate within their defined bounds, which aids the stability of the application. It is also a major factor for the security of a fog application, because a malicious device compromised by a hacker cannot request the other devices to do something that would violate the constraints.

B. Business Process Design

The next step is to design the business processes that will be realized by the fog application. For each business process the software architect has to create a UML activity diagram. Activity diagrams are well suited for this purpose, because they capture the essence of a process without unneeded details. The captured information includes which devices are called

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

or contribute to the process, the order of the actions to be executed, as well as major control flow decisions. Activity diagrams are also part of the Foundational UML Subset (fUML) standard [12]. Since fUML models can be executed, they are well suited for code generation. *FogUML2Code* supports all node types contained in fUML. It does not support all action types, but it supports *Opaque Actions*, which have been excluded from fUML. In Section IV-A, one can observe that the information obtained from activity diagrams is detailed enough to generate boilerplate code with handler classes for the activity nodes, which significantly reduces development time (see Section V for details on this). Only the business logic in the handler classes will need to be implemented manually.

Designing an activity diagram for a fog application is a straightforward task. The activity starts with an *Initial Node*. *Call Operation Action* nodes are used to call methods from the REST interfaces designed in the previous step. This type of node allows the architect to directly reference a method from a classifier in the UML model, thus enabling a resolution of the method during code generation. The architect may call methods from any REST interface in the model or from classes in the package that will realize this activity (by being marked with the `ActivityRealization` stereotype). Classes inside an `ActivityRealization` package will be generated as singleton services, which are only available on the device that hosts the activity and are not callable via REST.

To model decisions in the control flow, the architect may add *Decision Nodes*. Each outgoing edge of a decision node should be given a name that reflects the condition under which this edge will be taken, e.g., `[LastCheckOlderThan1Hour]` or `[ContainerIsEmpty]` - the edge that will be taken in case none of the conditions is true, must be labeled with the keyword `else`. This makes the diagram easy to understand and provides the code generator with human readable method names that will be used during generation. The other activity diagram node types currently supported by our code generator are *Fork*, *Join*, *Merge*, *Accept Event Action*, and *Opaque Action* nodes. They are implemented according to their semantics defined in the UML specification [10]. An opaque action node can be used if the software architect wants to run some custom action that is not included in any REST interface or service class. For such nodes a handler class will be generated, which developers can implement themselves. *Final Nodes* are not supported, because the fog business processes modeled here will probably be continuous processes. Note that the architect need not worry about passing state information from one activity node to the next, because a generated application state class will be available in all handler classes.

Finally, the software architect connects the activity diagram to the package in the class diagram that will realize the activity, i.e., the package that contains the service classes referenced in the activity. This is done by applying the `ActivityRealization` stereotype to the desired package and then setting the stereotype application's `realizedActivity` attribute to the activity that it will realize. Furthermore, the `FogDevice` stereotype should be applied to the package as well to set the target programming language and possibly other device parameters.

IV. Operation

After the fog application has been modeled, our code generator, *FogUML2Code*, can be used to generate boilerplate code for all fog devices, requiring developers to implement only the details of the devices and business processes. Tedious tasks, like configuration of the build system or setting up and running the HTTP REST services are handled by the code generator. For C the targeted build system is *CMake*, while for Java *Apache Maven* is used.

After the code has been generated, developers can implement those parts of the code that could not be generated automatically. As mentioned in the previous section, this refers to two main parts: i) the business logic of each REST endpoint operation, and ii) the handler classes for the activity nodes.

The generated code ensures that developers can work with the data model types directly and need not deal with HTTP or JSON handling. Developers must however, write code that does not break if a setter invocation fails due to a constraint validation or if the activity state does not contain all the desired information, because a preceding action has failed.

The manual implementation of the business logic is required because modeling and implementation are two activities that are not distinct. Generating all business logic from a model would require modeling all low-level sensor details as well. Thus, in essence, the model would collapse to an implementation. Extensions mitigating this phenomenon, such as the Action Language for Foundational UML (ALF) [13], would require orthogonal facilities to our approach.

A. Code Generation

To provide concrete tool support and a proof-of-concept implementation, we realized *FogUML2Code*[5], a model-to-code generator implemented using *Eclipse Acceleo*[6], which conforms to OMG's MOF Model to Text Transformation Language (MOFM2T) specification [14]. After installing *FogUML2Code* as an Eclipse plugin, it can be run from within MDT on any EMF UML model with our `Fog` profile applied. The code generation process consists of these major steps:

1) Generation of the `common` module;
2) Generation of a REST controller module for each REST interface (deployable on the configured `FogDevice`);
3) Generation of an activity realization module for each activity.

The generation of each module can be further customized using a `codeGeneration.properties` file in the project. It contains a list of settings, which can be configured globally and individually for each module (REST controller or activity realization). The settings use the following scheme: `codegen.<module>.<setting_name> = <value>`. Using `default` as the module name, allows defining global

[5]https://github.com/fog-uml-2-code/fog-uml-2-code
[6]https://www.eclipse.org/acceleo/

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

settings for all modules, which may be overridden for each module if desired. The list of all available settings and their explanations is available in the *README* file of the FogUML2Code project. In the following subsections, each major code generation step will be examined in detail.

1) The Common Module: The `common` module contains classes for all model types and interface definitions for all REST interfaces. It is generated in all available target languages, such that each module can reference it, regardless of what language it will be generated in. Currently, *C* and *Java* are supported, as the major languages used within IoT/fog. Support for others may be similarly added by extending FogUML2Code's capabilities. The model classes also contain code to enforce the constraints that were specified in the UML model using OCL.

Language differences between C and Java dictate setting in place a way to map a UML class diagram to a non object-oriented programming language. To be able to enforce the validation of the constraints defined on the UML model, the design choice of simulating encapsulation by generating a public and a private header file for each type was adopted. The public header declares a constructor, destructor, getter/setter functions, and a pointer type to the data structure that will represent the UML classifier, but no definition of the structure's fields. The definition of the data structure is generated in the private header file. Both header files are included by the implementation (.c) file of the type. However, only the public header file is meant to be included by code written by the developers of the fog application. We identify improved tracking of C variable types (i.e., values vs. pointers) as future work.

Central header files provide common definitions of types (e.g., 32-bit integers), whose sizes may otherwise vary depending on the platform. For integers and reals, the size of the default data type can be customized for each module.

The constraint validation code for C is generated directly inside of the getter and setter functions or the functions of the REST operation implementations. Support of class invariants and operation pre-conditions is provided out-of-the-box; further facilities may be added to FogUML2Code. To transform OCL expressions to C code, our prototype implementation uses regular expressions, thus supporting a limited subset of OCL. We identify implementing an OCL to C compiler to provide support for more OCL expressions as an avenue of future work. Each setter function returns a boolean value, indicating if the constraint validation has succeeded.

For C a header file is generated for each REST interface to provide a high level abstraction for the REST calls.

For Java as target language the generation process is analogous to that for C, but being an object-oriented programming language, it allows a more straight forward translation from the UML class diagram. The data model classes are generated as plain Java classes with private fields and a getter and setter for each of them. Instead of using basic data types (`int`, `double`, etc.), we have decided to use their wrapper classes (`Integer`, `Double`, etc.), because it simplifies the generation of the validation code and it allows developers to set each attribute to `null`. All arrays in the UML model are translated to `List<T>` objects. Java does not allow `typedef` declarations like C, so the data type size settings do not have any effect on Java code generation.

Constraint validation in our generated Java code is implemented using Aspect-oriented programming (AOP). FogUML2Code adopts the implementation provided by the *Micronaut* framework[7], which is a lightweight Java framework for building microservices with low memory footprint. This design choice is because in contrast to other Java frameworks, *Micronaut* does not depend on runtime reflection but resolution occurs mostly at compile time. This method leads to better performance and lower memory requirements, which is critical for the fog domain we target. Like for C, class invariants and operation pre-conditions as constraints are supported. They are realized by custom Java annotations, which are part of the *fog-execution-framework-java*[8] library reflecting our approach. The annotations are applied to the classes and/or methods by the code generator and for each constraint an implementation class is generated, which contains Java code obtained from the OCL constraint using regular expressions. If a constraint validation fails, an exception is thrown.

For the REST interfaces defined in the UML model, Java interfaces are generated that contain Micronaut annotations declaring the type of the HTTP verb used for each method. The implementation of the REST clients and endpoint controllers are generated in the modules that require them and are realized using Micronaut annotations as well.

2) REST Controller Modules: The next major step is to generate a module for each REST endpoint controller in the UML model. A REST endpoint controller in the UML class diagram is either an interface or a class that realizes an interface. The name of a module is that of the classifier that defines the REST controller, i.e., the class in the UML model that realizes the interface or the interface itself, if there are no realizing classes. If the `FogDevice` stereotype has been applied to a REST classifier, the module is generated in the target language configured in the respective attribute, otherwise Java is used as default. For both, C and Java, the generated module will be an executable application that will start up the REST services automatically and abstract away the HTTP communication, as well as the JSON serialization/deserialization. Developers only need to implement the business logic of each REST endpoint operation as normal functions/methods operating on the data types of the `common` module.

In C, the REST services are set up using the *ulfius* framework[9]. It works on desktop systems, which is useful for testing, and can also be compiled with *lwIP*[10], a lightweight TCP/IP stack for microcontrollers. FogUML2Code generates the code necessary to launch the REST endpoint and accept incoming requests. The generated code then calls a function

[7]https://micronaut.io
[8]https://github.com/fog-uml-2-code/fog-execution-framework-java
[9]https://github.com/babelouest/ulfius
[10]https://savannah.nongnu.org/projects/lwip/

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

containing the business logic, which needs to be implemented manually by the developers. The return value of that function will be serialized and transmitted to the REST client by the generated code.

For Java the amount of code that needs to be generated is much smaller. Each REST controller module contains a .java file with the main entry point, which launches the application using the Micronaut framework. For the REST endpoint controller itself, a class is generated, which implements the corresponding Java interface from the `common` module and has the Micronaut `@Controller` annotation applied, such that it is recognized by the framework as a REST controller. The HTTP communication and serialization/deserialization of parameters and return values is all handled by Micronaut. Annotations for any defined constraints are added and validator classes are generated.

3) Activity Realization Modules: The final major step in code generation is to generate a module for each activity in the UML model. Each activity is serialized to a JSON document, containing all the information necessary to execute it using the activity execution engine in our *fog-execution-framework-java* library. Within the framework, only Java is currently supported as a target language for activity realizations. This is motivated by the fact that most fog business processes will not be run on a microcontroller but on a more powerful device capable of executing Java.

FogUML2Code generates an entry point for each activity realization module, which bootstraps the application using the Micronaut framework and instructs the activity execution engine to load the activity's JSON document and start the execution on the initial node. The basic structure of the activity JSON is shown in Listing 1.

Listing 1. Activity.json - Basic Structure

```
"activity": {
   "name": "ActivityName",
   "initialNode": "0",
   "nodes": [ {
         "id": "0",
         "type": "InitialNode",
         "name": "Start",
         "nextNode": "1"
      },...]} ...
```

Each activity has a `name`, which is the name of the activity in the UML model, an `initialNode` referenced by its ID, and an array of `nodes`. Each node has an `id`, which is used to reference it from other nodes, a `type`, and a `name`, the other properties vary depending on the node type. For an `Initial Node` for example, there is only one additional property, `nextNode`, which contains the ID of the node that should be executed next. Each supported activity node type is handled by an executor class in our *fog-execution-framework-java*. Some node types, e.g., a `Fork` node, can be executed just by its executor class and the information from the activity JSON, but other types, such as a *Call Operation Action* node need code to be implemented.

A *Call Operation Action* node executes a call to a REST endpoint controller. This is handled by the execution engine; the developer is required to provide the parameters for the REST call and to handle the result. FogUML2Code creates a class that implements a handler interface with methods to assemble the parameters for the REST request, to handle its result, and to handle an eventual error. These need to be implemented by the developers.

Another example for a node type requiring code from developers is the `Decision Node`. For each such node a handler class is generated with one method for each outgoing control flow edge, except for the `else` edge. Each of these methods must return a boolean value indicating whether the control flow edge guarded by it should be taken or not. The executor responsible for the node executes the condition methods in the order in which they are defined in the `conditions` array in the node's JSON (see Listing 2). For each condition the JSON document contains a reference to the `nextNode` that should be executed if the condition is `true`. If none of the conditions evaluate to `true`, the node referenced by the `elseNextNode` property is executed.

Listing 2. Decision Node JSON Example

```
"id": "10",
"type": "DecisionNode",
"name": "",
"handler": "healthcare.monitoring.
   handlers.decisions.AfterQueryData_Id10",
"conditions": [ {
   "condition":
       "LastBloodPressureCheckOlderThan1Hour",
   "nextNode": "11"
} ], "elseNextNode": "9" ...
```

Recall that security plays an important role in a fog application and that fog devices may fail or leave the fog without notice. To enforce security concerns, automatic constraint validation is adopted – an exception will be raised if a constraint fails to validate. An unanswered REST call due to a failed device also produces an error. For such situations the activity execution engine is designed to automatically handle the error, thus preventing the application from crashing, and to continue execution. For each node type the executors in our engine define the maximum number of times the execution of a node will be retried in case of an error and a next node, which should be executed in case the current one fails on all retries. Since this can lead to skipping of nodes if they fail too often, developers must take into account that previous nodes may have failed, when implementing handler classes.

V. Evaluation

Our approach targets heterogeneous IoT and fog systems, where software components may be situated on IoT devices, cloud infrastructure, or edge devices. Our evaluation goal targets applicability of our approach. Specifically, we aim to investigate applicability over the complete consideration of a realistic use case, featuring a modern health care scenario. Thereupon, we describe our evaluation setup and modeling activities. The complete model of the use case considered along with an illustrative complete implementation can be found on GitHub[11]. We first present our use case, demonstrating that

[11] https://github.com/fog-uml-2-code/health-care-iot-example

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

our engineering approach leads to a reduction in development time and then conclude with a discussion.

A. BAN Use Case

As a representative use case, consider a BAN application for a diabetic patient. Kwasnicki and Yang examine various usage scenarios of body sensor networks for medical uses [15], while Fioravanti et al. propose a system for diabetes management [16] that has sparked the idea for our scenario.

The health care fog application proposed can monitor vital signs such as blood sugar levels and heart rate, auto inject insulin through a pump device, or alert the hospital in case it is needed. Furthermore, it can transmit aggregated monitoring data to the doctor at regular intervals and allows the doctor to remotely adjust the treatment plan. An eventual failure of the heart rate sensor should not cause the insulin treatment to cease working, while a malfunction of the blood sugar sensor must not cause an overdose of insulin to be injected into the patient, e.g., by defining a maximum amount that can be administered per hour. The BAN scenario presented is intended as an exemplar use case highlighting the model-driven engineering facilities of the approach, while not being medically accurate.

Figure 3 shows a part of the class diagram capturing the BAN application. The `Sensors` package contains four (REST) interfaces of sensors that can be placed on or implanted in the body: `HeartRateSensor`, `TemperatureSensor`, `BloodPressureSensor`, and `BloodSugarSensor`. Each of these interfaces will lead to a distinct deployment module, in fact, each of them has the `FogDevice` stereotype applied. The `Treatment` package contains another interface with the `FogDevice` stereotype applied - `InsulinPump`. It does not model a sensor however, but a device that can be used for injecting insulin into the patient's blood. It can do this either by means of a basal rate (basically a continuous long term dose) or a bolus (a dose administered over a shorter period of time, e.g., 15 minutes).

The `Models` package (not shown in Figure 3) has the `DataModel` stereotype applied and contains most of the model classes needed in this fog application. Some of them are used as return types for operations provided by the sensors and others, namely `BasalRate` and `Bolus` are needed for the insulin pump. The `InsulinPumpState` class contains information about the amount of insulin administered within the last hour and the last 24 hours and the time when the last bolus administration has ended. The state also contains the current `TreatmentPlan`. It contains the basal rate and bolus (if one is currently being administered) and the limits for insulin administration, which are supposed to ensure the safety of the patient.

To enforce these limits certain OCL constraints are defined; class invariants for `InsulinPumpState` and `TreatmentPlan` and a pre-condition for the `startBolus()` and `adjustBasalRate()` operations of the `InsulinPump`. For example, the `maxInsulinDosage` invariant on the `InsulinPumpState` class ensures that the number of insulin

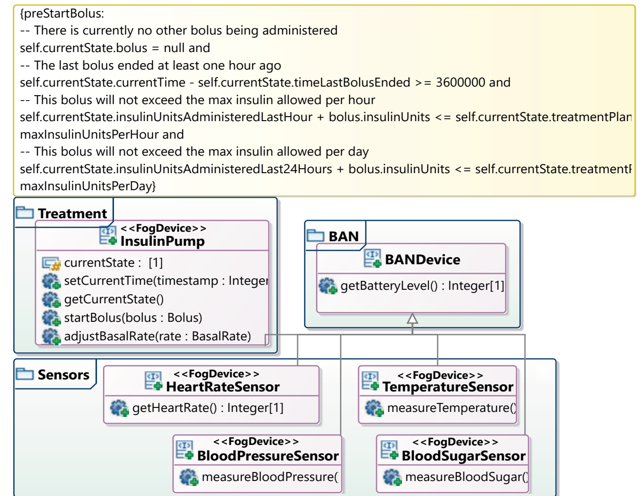

Fig. 3. Health Care Fog Application - Class Diagram (fragment).

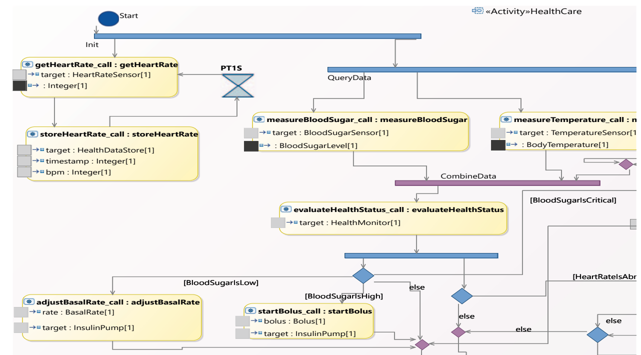

Fig. 4. Health Care Fog Application - Activity Diagram (fragment).

units administered over the last 24 hours and over the last hour do not exceed the respective maximum insulin units defined in the treatment plan. The `preStartBolus` operation pre-condition verifies four conditions before allowing a bolus to start.

The `Monitoring` package has the `FogDevice` and `ActivityRealization` stereotypes applied, because it serves as the realization of the `HealthCare` activity shown in Figure 4. The package contains two service classes used by the activity.

At first glance, the `HealthCare` activity in Figure 4 seems very complex, highlighting the advantage of reasoning on the model than on an implementation level. Furthermore, code generation will produce a module capable of executing the complicated business process at hand using an activity execution engine, requiring the developers to only write code for application details.

The activity starts by forking into two concurrent control flows: one runs a loop querying the heart rate sensor every second and the second one starts querying the other sensors for their data. The blood sugar level and the body temperature are measured; blood pressure is measured only if the last last measurement is older than one hour. These data are

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

combined and the `evaluateHealthStatus()` method of the `HealthMonitor` service is called. After this, the control flow forks again in order to make two decisions. One depends on whether the blood sugar is low; in that case the basal rate is adjusted (lowered). If the blood sugar is high, a bolus is administered and when the blood sugar is critically low or high, an ambulance is called. The second decision checks if the heart rate is abnormal, which would cause the blood pressure to be measured if the available data is older than four minutes. If the blood pressure results to be critical, an ambulance is called. If the blood pressure is acceptable or if the heart rate was normal in the first place, the control flow continues to a `Join` node, where it meets the control flow from the blood sugar evaluation. Next, the obtained data is stored. Subsequently a check entails if 24 hours have passed since the last data submission to the doctor; if this is satisfied, the data is submitted. Next, if the last treatment plan update check is older than one hour, the plan is updated. Finally the control flow leads to a timer event, which will trigger the querying of data again after one minute.

Using the facilities of our approach, boilerplate code is generated starting up REST interfaces for the sensors and the insulin pump and executing the defined activity in the `HealthCare` activity realization module. For all modules, the defined constraints are automatically enforced; required implementation is the particular business logic parts of devices and the activity realization. One can observe that the MDE approach advocated leads to a reduction in development time. Specifically, the following development items are automatically produced:

- Data model classes (in multiple programming languages);
- REST interfaces setup:
  - HTTP server and requests handling;
  - JSON serialization and deserialization;
  - Constraints enforcement upon REST operation;
- Failure-tolerant execution of business processes.

B. Discussion and Perspectives

We have demonstrated that the MDE approach advocated can aid development of fog applications. The error handling built into the activity execution engine is well suited for the volatile nature of a fog environment. The REST operation calls trigger either a result handler, if the call was successful, or an error handler, if it was not to update the application state of the activity realization. This helps ensure that this state is always well defined. The automatic enforcement of constraints provides further error resilience and aids protection from malicious attacks.

The error handling inside the activity execution engine could be further improved in the future by making application state changes only possible by means of transactions. Currently, if the application state contains an instance of a data model class that has some constraints associated with it, making a change that violates a constraint would result in an exception, which would be caught by the execution engine. If the developers do not catch the exception themselves, the application state may be left in an invalid state - with some intended changes already applied and some not. Furthermore, if the invalid change was detected by a class invariant constraint, the constraint validation was performed after setting the disallowed value. Use of a transaction system that would ensure that all the changes of an activity node are either fully committed or not committed at all, would help avoid such bugs.

The error handling for REST endpoint controllers can be improved. Since transactions would be too expensive for most microcontrollers, a different approach may be taken. If all constraints are required to be expressed as operation pre-conditions, any operation could be only executed if it will lead to a valid state. Currently the code generation facilities support pre-conditions and invariants. A future step could be to try to automatically transform all invariants into pre-conditions during code generation, such that all constraint validation would occur before an action is executed. Another improvement would be the replacement of regular expression-based OCL to code transformation with an OCL compiler, similar to *CrossEcore*[12], to support more types of assertions.

The Fog UML profile and the code generator may also be extended in the future to support the generation of code that conforms to certain IoT standards, e.g., the Open Connectivity Foundation (OCF)[13] standards, to allow for easier integration with other frameworks. Furthermore, empirical evaluation of the presented methodology may be conducted.

C. Related Work

Recursive model transformations are adopted in [17] to derive a fog-based architecture from requirements defined as UML Use Case diagrams. Such diagrams are used as input to the *four-step-rule-set* (4SRS) method, which consists of four transformation steps. The first one leads to a set of architectural components, which are grouped into three categories: interface, data, and control. The second step eliminates unnecessary components, while the third one yields semantically cohesive aggregations of components. Finally, the fourth step is used for associations between components. The output architecture is then modularized and used to create a *service Use Case diagram*, which is a refined Use Case diagram for the system with actors being either humans or computer services. This is then used as input to another execution of the 4SRS method, which yields a set of microservices, for which scenarios are defined as a variant of sequence diagrams [17]. Even though this approach makes heavy use of UML, it does not include any generation of code directly from the models.

A model-driven approach for provisioning and managing IoT services in an infrastructure comprised of cloud, network, and fog nodes that "is exposed to service administrators as a unified resource fabric" [18]. The model of a service only defines what the service has to do, but not how it is instantiated on a specific device. This allows a service to be deployed on

[12]https://github.com/crossecore
[13]https://openconnectivity.org

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.

those nodes (cloud or fog) of the system that the administrator selects or which the system itself determines to be most suitable. The authors state that there is a pilot implementation of such a system in the city of Barcelona [18].

Based on our observations and those of [17], it appears that the community has largely not pursued model-driven design in fog so far. The *OpenFog Reference Architecture* is a fog computing architecture that can be applied to any IoT scenario. It consists of a hierarchy with the cloud on top and the IoT sensors and actuators at the bottom, everything in between can be seen as part of the fog. Raw data processing takes place close to the IoT sensors and actuators and decreases as the hierarchy is traversed upwards towards the cloud. Intelligence (machine learning, etc.) on the other hand, is performed very little or not at all by the bottom layers of the hierarchy and increases towards the cloud [19]. Within [20], a hierarchical fog architecture for analyzing Big Data generated by smart cities is described. Like the OpenFog Reference Architecture, its lower layers focus more on data processing, while the higher layers focus on intelligence. In [21] authors propose a sort of operating system (FogOS) to manage all cloud, fog, and edge resources, such that fog application developers can use an API to request certain resources (e.g., video cameras in certain locations) and FogOS will use the registered devices to satisfy these requests like a computer operating system that handles an application's request for certain hardware. The idea of a fog orchestrator in [22] is similar, because the proposed orchestrator has to decide on which nodes a workflow (with its specific requirements) is deployed.

Finally, there have been various research streams regarding the use of models to support systems at runtime. Even though not directly fog computing related, such a conception may be used in fog computing as well. For example, [23] proposes an alternative to the EMF with lower resource requirements, thus making it more suitable for constrained devices.

VI. Conclusions

IoT computing applications bring significant challenges to software engineers. The fog/IoT environment is very volatile, where devices can fail and/or leave without notice, while dependability is a highly sought property. We proposed a methodology and supporting framework consisting of a four step process: (i) design of the data model and REST interfaces using UML class diagrams and definition of constraints using OCL, (ii) design of business processes using UML activity diagrams, (iii) automatic code generation facilities designed specifically for fog computing, and (iv) implementation of details that cannot be easily captured in a model.

We identify several directions for future work; communication methods other than REST (e.g., MQTT or CoAP) and the interplay between them may be investigated, support for more target programming languages and conformance of the generated code to specific IoT standards (e.g., OCF) may be added, a full OCL to code compiler could be implemented, the error resilience may be improved, and automatic translation of invariants to pre-conditions could be realized.

References

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. of MCC Workshop on Mobile Cloud Computing*. New York, NY, USA: ACM, 2012, pp. 13–16.

[2] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, Dec 2016.

[3] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, Nov 2015, pp. 73–78.

[4] Z. Hao, E. Novak, S. Yi, and Q. Li, "Challenges and software architecture for fog computing," *IEEE Internet Computing*, vol. 21, no. 2, pp. 44–53, Mar 2017.

[5] C. Tsigkanos, I. Murturi, and S. Dustdar, "Dependable resource coordination on the edge at runtime," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1520–1536, 2019.

[6] N. Li, C. Tsigkanos, Z. Jin, S. Dustdar, Z. Hu, and C. Ghezzi, "Poet: Privacy on the edge with bidirectional data transformations," in *2019 IEEE International Conference on Pervasive Computing and Communications, PerCom 2019, Kyoto, Japan, 2019*. IEEE, 2019.

[7] C. Tsigkanos and T. Kehrer, "On formalizing and identifying patterns in cloud workload specifications," in *13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016, Venice, Italy, April 5-8, 2016*, 2016, pp. 262–267.

[8] N. Bencomo, R. B. France, B. H. Cheng, and U. Aßmann, *Models@ run. time: foundations, applications, and roadmaps*. Springer, 2014, vol. 8378.

[9] IEEE, "Ieee standard for local and metropolitan area networks - part 15.6: Wireless body area networks," *IEEE Std 802.15.6-2012*, pp. 1–271, Feb 2012.

[10] Object Management Group, "Unified modeling language - version 2.5.1," https://www.omg.org/spec/UML/2.5.1/, December 2017.

[11] ——, "Object constraint language - version 2.4," https://www.omg.org/spec/OCL/2.4/, February 2014.

[12] ——, "Semantics of a foundational subset for executable uml models - version 1.4," https://www.omg.org/spec/FUML/1.4/, December 2018.

[13] ——, "Action language for foundational uml - version 1.1," https://www.omg.org/spec/ALF/1.1/, June 2017.

[14] ——, "Mof model to text transformation language - version 1.0," https://www.omg.org/spec/MOFM2T/1.0/, January 2008.

[15] R. M. Kwasnicki and G.-Z. Yang, *Clinical Applications of Body Sensor Networks*. Wiley-Blackwell, 2014, ch. 16, pp. 479–504.

[16] A. Fioravanti, G. Fico, A. G. Patón, J.-P. Leuteritz, A. G. Arredondo, and M. T. A. Waldmeyer, *Health-Integrated System Paradigm: Diabetes Management*. Wiley-Blackwell, 2014, ch. 22, pp. 623–632.

[17] N. Santos, H. Rodrigues, J. Pereira, F. Morais, R. Martins, N. Ferreira, R. Abreu, and R. J. Machado, *Specifying Software Services for Fog Computing Architectures Using Recursive Model Transformations*. Springer International Publishing, 2018, pp. 153–181.

[18] F. van Lingen, M. Yannuzzi, A. Jain, R. Irons-Mclean, O. Lluch, D. Carrera, J. L. Perez, A. Gutierrez, D. Montero, J. Marti, R. Maso, and a. J. P. Rodriguez, "The unavoidable convergence of nfv, 5g, and fog: A model-driven approach to bridge cloud and edge," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 28–35, 2017.

[19] O. C. A. W. Group, "OpenFog reference architecture for fog computing," OpenFog Consortium, Tech. Rep., February 2017.

[20] B. Tang, Z. Chen, G. Hefferman, T. Wei, H. He, and Q. Yang, "A hierarchical distributed fog computing architecture for big data analysis in smart cities," in *Proc. of the ASE BigData SocialInformatics 2015*, ser. ASE BD'15. New York, NY, USA: ACM, 2015, pp. 28:1–28:6.

[21] N. Choi, D. Kim, S. J. Lee, and Y. Yi, "A fog operating system for user-oriented iot services: Challenges and research directions," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 44–51, 2017.

[22] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, Mar 2017.

[23] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel, "An eclipse modelling framework alternative to meet the models@ runtime requirements," in *Intl. Conference on Model Driven Engineering Languages and Systems*. Springer, 2012, pp. 87–101.

Authorized licensed use limited to: Universitaetsbibliothek der TU Wien. Downloaded on August 13,2020 at 14:12:17 UTC from IEEE Xplore. Restrictions apply.