International Journal of Architectural, Civil and Construction Sciences
ISSN: 2415-1734
Vol:12, No:6, 2018

Introduction of an Approach of Complex Virtual Devices to Achieve Device Interoperability in Smart Building Systems

Thomas Meier

***Abstract*—**One of the major challenges for sustainable smart building systems is to support device interoperability, i.e. connecting sensor or actuator devices from different vendors, and present their functionality to the external applications. Furthermore, smart building systems are supposed to connect with devices that are not available yet, i.e. devices that become available on the market sometime later. It is of vital importance that a sustainable smart building platform provides an appropriate external interface that can be leveraged by external applications and smart services. An external platform interface must be stable and independent of specific devices and should support flexible and scalable usage scenarios. A typical approach applied in smart home systems is based on a generic device interface used within the smart building platform. Device functions, even of rather complex devices, are mapped to that generic base type interface by means of specific device drivers. Our new approach, presented in this work, extends that approach by using the smart building system's rule engine to create complex virtual devices that can represent the most diverse properties of real devices. We examined and evaluated both approaches by means of a practical case study using a smart building system that we have developed. We show that the solution we present allows the highest degree of flexibility without affecting external application interface stability and scalability. In contrast to other systems our approach supports complex virtual device configuration on application layer (e.g. by administration users) instead of device configuration at platform layer (e.g. platform operators). Based on our work, we can show that our approach supports almost arbitrarily flexible use case scenarios without affecting the external application interface stability. However, the cost of this approach is additional appropriate configuration overhead and additional resource consumption at the IoT platform level that must be considered by platform operators. We conclude that the concept of complex virtual devices presented in this work can be applied to improve the usability and device interoperability of sustainable intelligent building systems significantly.

***Keywords*—**Complex virtual devices, device integration, device interoperability, Internet of Things, smart building platform.

I. Introduction

DEVICE interoperability is one of the most important enabling factors for the success of sustainable smart building systems. We still find a confusing mess of incompatible and disparate devices from different vendors that a smart building system needs to deal with. Apart from different communication protocols we find an unpredictable set of functionality and data structures that these devices use during communication with the smart building system. Even worse, we cant give an answer for which standard wins out since industry cycles take long to settle down. We believe that even in the future we always find different standards that a smart building system must be prepared deal with. The *Internet of Things* (IoT) definition given by the International Telecommunication Union (ITU) [1] leads to the conclusion that demand for interoperability in IoT exists on different layers such as business layer, application layer or technical layer. During this paper we primarily focus on technical and application interoperability. In the past, various smart building systems have been developed that tackle the device interoperability problem in different ways. A sustainable smart building system is supposed to support a variety of heterogeneous and vendor agnostic devices and to provide rich scenarios or services based on the data of these devices.

In Section II, we introduce the terms and principles of typical IoT architecture that will be used in the remainder of this work and emphasize their contribution to device interoperability in smart building systems. Section III gives a brief presentation of a smart building system we have developed in the past [2]. After that, we discuss approaches that have been implemented in OpenSB to tackle device interoperability problem (Section IV). Finally, in Section V, we summarize our findings and present our plans for future work in this area.

II. IoT Architecture

Usually, IoT architectures are structured by layer or stages as presented in [3]- [6]. The different variants differ slightly in terms of naming and granularity. Fig. 1 depicts the layers of a typical IoT architecture.

Sensor or actuator devices are located at the device level, which is sometimes referred to as perception or sensing layer [4]. The network layer provides the connectivity between devices and the IoT platform layer by means of standardized network technology and protocols. Due to the heterogeneity of the devices used and their variety of physical interfaces and protocols, an IoT gateway is usually necessary. The platform layer, sometimes referred to as system layer, represents the domain specific IoT system (e.g. smart building, industry 4.0, vehicle telemetric). Typically, the system business logic performs rule processing (e.g. by means of a script or rule engine), data persistence and data analytics. Additionally, the platform layer provides appropriate public interfaces (e.g. a REST compliant Web Application Programming Interface [7]). The application layer leverages platform's external interfaces

T. Meier is with Hochschule fuer Telekommunikation Leipzig, University of applied sciences, Leipzig, D-04277, Germany (corresponding author, e-mail: meier@hft-leipzig.de).

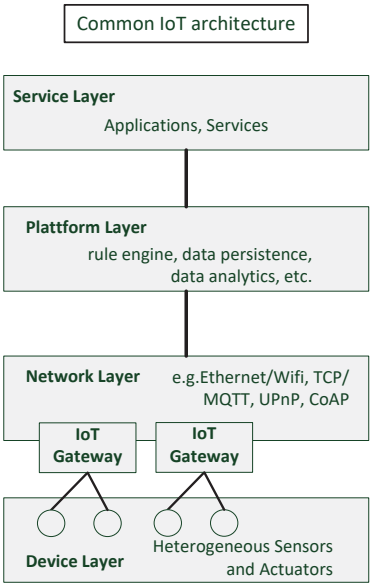

Fig. 1 Layers of an IoT Architecture

to interact with the connected devices. On this layer, we find applications and services, e.g. web-based management applications and intelligent services that leverage data from multiple platforms to provide rich scenarios.

Device interoperability is necessary on platform and application layer so that the applications, services and the platform's rule engine are able to change device states depending on the state of other devices. Hence, a smart building system needs to provide appropriate device integration mechanisms to enable device interoperability. In the following we give a brief description of each layers contribution to device integration.

A. Integration at Device Layer

Due to their heterogeneous nature and different communication techniques and protocols devices are not able to communicate with each other locally. Hence, device integration at the lowest layer is accomplished by means of an IoT gateway function [3]. Depending on the specific IoT system, many IoT gateways can be connected to the system. In a few cases, the IoT gateway function can be part of the IoT platform component.

An IoT gateway function is supposed to provide:

- Support of communication protocol interworking, i.e. the gateway connects with devices (sensors, actuators) that use various communication protocols (e.g. serial communication, LTE, Ethernet, WiFi, Bluetooth, ZigBee [8]) and provides a standardized protocol interface to the IoT platform (e.g. MQTT [9], [10] or CoAp [10]).
- Transfer device data to the platform layer or receive data from platform layer and forward that data to specific devices.

Consequently, besides communication protocol transformation, an IoT gateway needs to convert data from device specific formats into a format that is understood by the platform layer and visa verse [11]. Typically, device vendors or developers provide appropriate IoT gateway functionality for specific device, e.g. by means of device specific gateways or gateway device driver plug-ins.

B. Integration at Platform Layer

In the course of this paper, the key tasks of an IoT-based smart building platform can be summarized as follows. The smart building platform:

- Gathers data from possibly many devices
- Runs business logic using device data
- Sends data to devices to cause device state changes
- Provides appropriate public interfaces for applications (see Section II-C) to enable domain specific services and rich scenarios to be built.

Device communication can be achieved by appropriate IoT gateway functionality (see Section II-A).

Business logic typically involves a rule engine. Rule engines need to have access to device functions by means of internal platform interfaces (e.g. to query or change device state). Implementation of internal platform depends on the specific platform and on the data characteristic that the IoT gateway provides. We will discuss details in Section III-A.

Smart Building platforms also need to provide appropriate external interfaces that allow application developers to build client applications and rich services at application layer. The device representation of an external interface depends on the platforms data model, i.e. on the internal device representation. The consequences of internal device representation to device interoperability at this level are discussed in Section III-C).

C. Integration at Application Layer

Application layer integration, in the context of this paper, denotes the possibilities to allow applications (e.g. smart building web applications, management clients, smart services) to access device functionality by means of an external platform interfaces (see Section II-B). In the following we point out two conditions for external platform interfaces that allow application level interoperability:

- Completeness of device information:
  Ideally, the external interface of the platform should contain all the information that is needed to run applications or services e.g. display user friendly device representations or provide means to change device state without the application having to implement additional device-specific knowledge.
- Interface stability:
  Ideally, the platform external interface definition must not be altered or extended when new devices are to be connected to the system (i.e. no additional or altered data attributes or functions)

Compliance with these two conditions enables sustainable smart building systems that allow new devices to be connected to the running system without the need to change or upgrade applications or services. A discussion of different approaches to fulfill these requirements is given in Section IV.

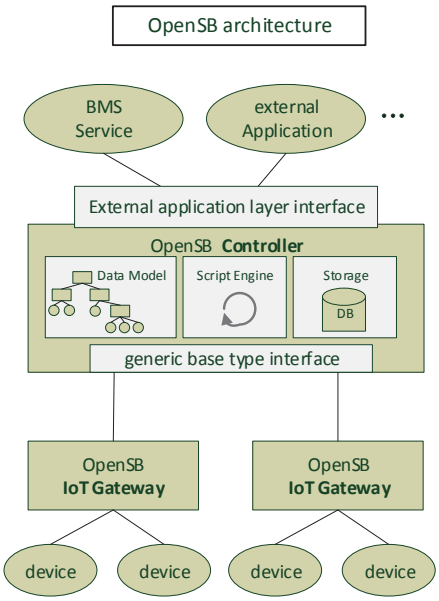

Fig. 2 OpenSB System Overview

D. The People in Smart Building Systems

Before we dive into the details of the smart building system and its contribution to device interoperability we will describe each of the different roles that people play in the development, administration and use of smart building system. Device vendor or device driver developers provide device drivers that allow real devices to connect with the IoT gateway. Hence, they provide the knowledge to translate device specific functions and data into a generic format understood by the IoT gateway. Platform developer can adjust the platform depending on specific requirements. This task should be avoided or minimized since it caused platform downtime and requires appropriate platform regression testing. Thus, when new devices are to be connected to the system, no platform adaption should become necessary. Platform users or operators are using the smart building platform according to their specific use cases. These people are supposed to have means to adjust the device representation and interoperability according to their needs. Application or service developers are implementing client applications that are used by platform users and operators for their individual use cases. Ideally, no changes to the platform code or application code should become necessary upon introduction of new devices, i.e. no development activities at these levels should become necessary when connecting new equipment.

III. OpenSB System Overview

Fig. 2 shows the overall architecture of the smart building system we have developed in the past [2], which is referred to as *OpenSB* in this paper.

A. OpenSB Gateway

The OpenSB gateway is an OSGi-based component that allows device drivers to be installed or removed at runtime by means of separate OSGi bundles. Basically, a device driver is responsible for connection with a physical device and translating between device specific functions and a generic device interface (base type interface) that defines the device data structures being passed between OpenSB gateway and OpenSB controller. Currently, OpenSBs base type interface defines numeric string based sensor and actuator types. More detailed information on OpenSB generic base type is given in [2]. Because of the ability to use many IoT gateway instances, the system can be applied to large building environments.

B. OpenSB Controller

The OpenSB controller component represents the smart building systems platform layer functionality (see Fig. 2). It allows external applications and services to access device management and data processing by means of a device agnostic RESTful Web Service interface. The controller maintains an internal data model that supports three main entities: devices, groups and scripts. Devices represent the internal devices. A group represents any logical or location entity (e.g. buildings, rooms, floors, lights, heaters). They can be created by operators as needed. Groups are organized in a tree structure, i.e. a group has one or more subgroups. Devices can be assigned to groups by operators as needed. Applications can access the groups and device structure by means of the external interface. Automatic processing of device data is performed by a script-based rule engine. System operators can register rule scripts without interrupting the running controller. Rule scripts have access to device and group functionality, e.g. to act upon state changes of devices. Relevant details of the script engine will be described in Section III-D. Besides the mentioned functions the controller component supports device shadow, i.e. device information will be stored to persistent memory. Thus, applications can access device data even when the real device is currently not connected or accessible. Additionally, the controller logs device data into persistent memory to provide device data history.

C. OpenSB Type Concept

To prevent misunderstanding and for easier understanding, a few definitions will be given that are used in the course of this paper. Fig. 3 illustrates the definitions based on an exemplary heating thermostat.

- *Real Device*: A real device is a, typically, physical device of any complexity (e.g. a heating thermostat, a SMS modem device or a room light).
- *Base Device Type*: A base device type represents a simple and generic type that presents just a single device functionality (e.g. a temperature sensor or light switch). Exemplary base device types are a number sensor/actuator, a string sensor/actuator or number range sensor/actuator. Besides its single functionality a base device type has a predefined set of data attributes. Base device types are generic, device agnostic and they can be universally applied.

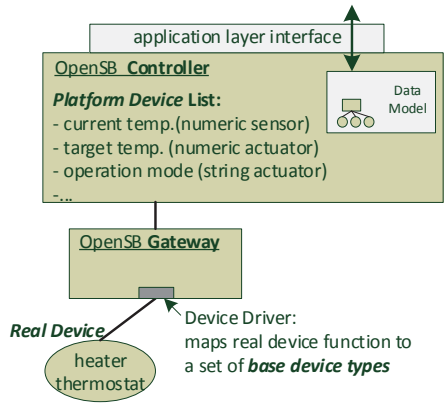

Fig. 3 Mapping between a real device and its resulting base type platform devices

```
"data":{
  "id": "76ad6daef7e",
  "type": "parent device",
  "attributes": {
     ...
    "name": "Heater Thermostat",
    "devices": [{
      "id": "feaa528002"
      "attributes": {
          "base-type": "number sensor",
          "name":"current temperature",
          "current-value": 22.3,
          "unit": "Celsius",
           ... }
      }, {
      "id": "fa45bfed678",
      "attributes": {
          "base-type": "string actuator",
          "name": "operation mode",
          "current-value": "AUTOMATIC",
          "target-value": "AUTOMATIC",
          "unit": "Mode" }
     }
   ]
}
```

Fig. 4 Device representation of a heater thermostat

- *Platform Device*: Platform device denotes the device representation as it is used internally by the OpenSB controller. A platform device applies the base type concept, i.e. each platform device is of the type of a base type. Hence, the functionality of a real device is composed of a set of base type platform devices. This composition is achieved by means of a parent device type which serves as a container for the respective base type platform devices. Fig. 4 demonstrates a simplified exemplary parent device data set representing of a heater thermostat as it is passed by the IoT gateway to the OpenSB controller.

Other smart building or IoT platforms follow similar approaches to enable device-independent data processing at the platform level or device-independent representation on the external application interface. For example: Eclipse Smart Home and the OpenHAB Open Source projects apply a *Thing-Item* concept where *Things* represent real devices which are composed by a set of some base type *Items* [12]- [14]. The EdgeX Foundry Open Source Project composes real devices by means of a set of base type *DeviceResources* [15]. The IoTivity [16] Open Source project allows device manufacturers to create device drivers that map real device functionality to IoTivity platform format by means of *ResourceContainer* which contains a set of base type *Resources*.

D. OpenSB Rule Script Engine

The OpenSB controller leverages a script engine that performs arbitrary data processing. The external application interface provides an appropriate script interface so that platform users or operators can add new scripts, alter existing scripts or remove scripts from the running controller. OpenSB applies an embedded JavaScript based script engine that allows users to implement rules in JavaScript language. The embedded rule engine concepts of other smart building or IoT systems are quite diverse and depend on platform architectural principles and other decisions. Some of them follow the same approach, i.e. using a script language engine to define rules [14], [17]. Other systems use a declarative approach, e.g. rule definition by means of JSON (JavaScript Object Notation) or XML (eXtensible Markup Language) data structures [2], [12], [18], or domain specific language (DSL) approaches [19]. In the recent past, the OpenSB controller rule engine design has been switched from a declarative XML rule engine approach [2] to a JavaScript rule engine mainly because of its increased flexibility. Besides the basic JavaScript language features, the OpenSB controller scripting engine provides an easy-to-use scripting API to perform the appropriate operations on the platform data model.

IV. OpenSB Device Interoperability

In the following we discuss techniques and approaches that enable device interoperability in the OpenSB smart building system.

A. OpenSB Platform Layer Device Interoperability

The OpenSB controller provides mechanisms to support device interoperability. Due to its data model and the device base type concept, new or unknown devices can connect to the system since the controller is able to handle the appropriate base types the new device is composed of. Hence, no development activities are necessary at platform layer to support new devices. The internal rule script engine allows devices to communicate with each other or more generally, the script engine supports device data processing at platform layer. No platform development or platform configuration becomes necessary when new devices are supposed to participate on data processing since rule scripts can be added, changed or removed via the external application interface by users or operators.

The script engine applies query techniques based on jQuery [20] and Sizzle [21] which are well established technologies in the web application domain for object selection and

```
ript.start = function() {
$("device").on("currentValue", "logCurrentValueInfo");

ript.stop = function() {
$("device").off("currentValue");

ript.logCurrentValueInfo = function(e,id,val) {
ript.logMsg("Dev."+id+" changed value to "+val);
```

Fig. 5 A simple rule script

manipulation. This approach provides two advantages: First, the selection mechanisms are proven to be powerful and flexible. Second, because of their simple syntax and wide spread, these programming methods are easy to learn and already familiar to many developers. An internal script API library, referred to as *SbScript*, provides methods to:

- Query the controller data model, i.e. fetch platform device objects or group objects
- Add/remove devices to/from groups
- Get current device state information
- Set device state (of actuator devices)
- Get or change device attributes or group attributes
- Register/unregister event handlers to react on device state changes or other events
- Trigger new events
- Start timer events

Fig. 5 illustrates a very simple rule script that generates a log message every time that the current value of any connected device changes.

The `start()` callback method is invoked by the script engine upon rule startup. The rule script registers a *currentValue* state change listener callback method, `logCurrentValueInfo()`, for every connected device. The rule `stop()` callback method unregisters the state change callback, i.e. it wont be executed after the rule script has been stopped. Rule scripts can be started or stopped by users or operators, i.e. the OpenSB external application interface offers appropriate mechanisms.

Together, the device independent internal data model and the power of the embedded script engine, allows configuring almost any device interoperability scenario. Nevertheless, this approach has some limitation if it comes to interoperability at application layer as pointed out below.

B. OpenSB Application Layer Device Interoperability

External applications or services leveraging the external interface can use just the platform devices and scripts. This causes restrictions in terms of supporting user-friendly use cases on application or service layer. Two examples scenarios should help to clarify these limitations.

Device Aggregation Scenario:
Operation staff is supposed to change the target value of all heater thermostat devices on a given floor to a given value. Hence, operators need an easy way to change a value of, possibly, hundreds devices at once.

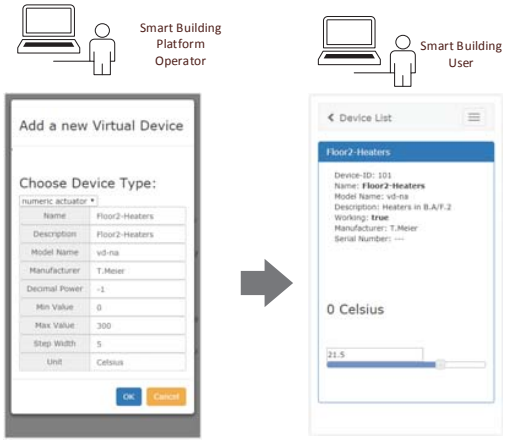

Fig. 6 Virtual device: creation and resulting representation

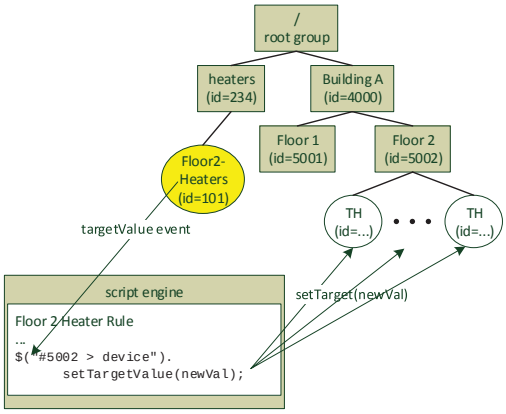

Fig. 7 An aggregated device scenario using a virtual device

Mixed Device Representation Scenario:
Operators want to have a representation of different platform base type devices that do not belong to the same physical device (device composition). This scenario improves device interoperability at application layer significantly.

Application scenarios based on aggregated or mixed device representations would demand additional development efforts on application or service layer. To avoid these costs and the associated burden of custom application customizations, we support *virtual devices* (see Section IV-B) and *complex script devices* (see Section IV-B).

1. Virtual Devices: A virtual device is a platform device that is not associated with a real device. Unlike real platform devices, virtual devices can be added or removed by users or operators via the external application interface. In virtual device creation, the user defines the same attributes and properties as real platform devices have (e.g., name, description, base type, etc.). As real platform devices, virtual devices can be assigned to groups etc. In fact, virtual devices become part of the controllers data model the same way as real platform devices and hence, they are immediately accessible via the external application interface as illustrated in Fig. 6.

```
Floor 2 Heater Rule
ript.start = function() {
"#101").on("targetValue", "targetChanged");

ript.stop = function() {
"#101").off("targetValue");

ript.targetChanged = function(e,id,oldVal,newVal) {
"#5002 > device").setTargetValue(newVal);
```

Fig. 8 Sets all target values of all devices in a group

Because virtual devices are anchored in the data model, the embedded script engine has full access to virtual devices. This approach has been applied by other systems as well ([12], [14]).

Aggregated device scenarios, as described above, can be easily configured by means of virtual devices and appropriate script engine rules. Hence, no further development or adaptations become necessary on application layer in this case. Fig. 7 illustrates an example of a device aggregation scenario in which changing the state of a virtual device changes the states of several other platform devices.

Fig. 8 shows the corresponding script rule that fires when the target value of a virtual device Floor2-Heaters (Id=101) has been changed, e.g. by an external application. Upon target value change, the script sets the target value of all heater thermostat platform devices that are assigned to a group Floor 2 (Id=5002) to the target value of the virtual device.

Applying the virtual device approach for mixed device representation scenarios would, technically, be possible but have a smell of being complicated. Multiple scattered virtual device configurations and additional rule configuration makes this task somewhat complicated and error prone. Thus, OpenSB introduced the concept of complex script devices.

2. Complex Script Devices: The term *complex* denotes the possibility to configure a device, which contains multiple sub-devices (*complex device members*) of different base types. Thats why this approach ideally fulfills the requirement of the mixed device scenario. The term *script* denotes the fact, that a complex device is created by the script engine as explained below. Nevertheless, complex script devices, and their individual device members, have the same representation on external application interface as platform devices, i.e. the external interface definition will not change upon creation of a new complex script device.

In summary: complex script devices have two major goals. First, complex script devices allow the merging of different devices (platform and virtual devices) so that the composition remains on the external interface (i.e., mixed device representation). Second, the definition of complex script devices also includes the description of their rules and behavior. This essentially differentiates the complex device script approach from the solutions of other systems where the representation and its behavior are defined in different places (e.g. [12], [14]). Simple configuration at a single location makes this approach simple and robust and avoids

```
r dev=new VirtualDevice("Dev1","string","abc");
mplexDev.addDevice(dev);
mplexDev.addDevice({deviceName: "Dev2",
  deviceType: "numeric",
  unit: "Watt",
  decimalPower: 0,
  minValue: 0,
  maxValue:100,
  currentValue: 0
 });
```

Fig. 9 Definition of two new complex device members

```
mplexDev.addDevice({deviceName: "Dev3",
  deviceType: "string",
  currentValue: "xyz",
  onTargetChange: function(newVal) {
    this.targetValue = newVal;
    complexDev.setCurrentValueOfDevice(this.newVal);
  }
;
```

Fig. 10 Definition of a new string actuator device member

any redundancy. Complex script devices can be configured, started and stopped via the external application interface by users or administrators during operation of the controller.

The embedded OpenSB script engine has been extended by a*ComplexDev* API library that provides appropriate for creating complex device members and add those to the corresponding complex device. Complex device members can be created from scratch, i.e. a new virtual device will be created as illustrated in Fig. 9. It defines new sensor complex device members: Dev1 represents a string sensor device with default current value `abc` and Dev2 represents a numeric sensor device with default value `0`.

To define an actuator device member, an appropriate `onTargetChange()` callback method must be defined as shown in Fig. 10.

Additionally, complex device members can be created based on existing platform devices or virtual devices as shown in Fig. 11.

The corresponding rules for state retrieval and for state changes of complex member devices are defined within the complex device script code as well. This can be achieved either by implementing the code into the appropriate `onTagetChange()` callbacks or by means of appropriate event callback listeners, that are to be registered during the `start()` method initialization as explained already in the virtual device section and Fig. 8.

A final example scenario should clarify the possibilities that arise through the use of complex script devices.
Building operators would like to have a device representation on the external application interface that includes several

```
r dev = $("#101");      // exist. virtual device
mplexDev.addDevice(dev);
v = dev = $("#774939"); // exist. platform device
mplexDev.addDevice(dev);
```

Fig. 11 Definition based on an existing platform devices

sub-devices.

1) An aggregated device that allows setting the target temperature of all heater thermostat devices on second floor in building A.
   The corresponding virtual device exists already in the data model (see Fig. 6).
2) A real sensor device, that shows to outside temperature of building A.
   We assume that a corresponding platform sensor device exists already in the data model (device id=774939).
3) A device that shows the current minimum temperature of all thermostat devices on the floor.
   This virtual numeric sensor device needs to be created by the complex device script.
4) A device, that allows operators to specify a phone number which receives a short message if the minimum temperature is below a given threshold.
   This virtual string actuator device needs to be created by the complex device script as well.

The resulting device representation at the external interface allows client applications to render the new complex device as it was intended by the operator. Fig. 13 shows an exemplary device view based on the complex device script from Fig. 12. Fig. 12 presents the script code that defines a new complex device according to the requirements given above. From the script code it becomes obvious that users or operators can define complex scenarios by means of complex device scripts they need to write. We are aware of the fact, that defining complex device scenarios requires appropriate programming skills from users or operators. This skill requirement can be minimized by applying appropriate graphical user interface techniques that offer graphical input elements for device setup. The input data needs to be converted into the corresponding complex device script by these tools. The development of such tools is beyond the scope of this paper. Similar techniques have been applied by other systems that aim to offer graphical front ends for creation of DSL-based rule scripts, e.g. [22].

```
complexDev.start = function () {
// register value change event handler for all
// thermostat temperature sensors in group 5003
$("#5003 > device").on("currentValue","cvChanged");
};
complexDev.stop = function () {
// unregister all change event handlers
$("#5003 > device").off("currentValue");
};
var minTemp = 100;
var minThreshold = 17;
complexDev.name = "Floor2 Thermostat Control",

// add virtual heater device
var floor2HeaterDev = $("#101);
complexDev.addDevice(floor2HeaterDev);

// add real temp sensor device
var outdoorTempSensor = $("#774939);
complexDev.addDevice(outdoorTempSensor);

// add virtual thermostat minimum temp. sensor
complexDev.addDevice({deviceName: "Min Temp",
deviceType: "numeric",
unit: "Celsius",
decimalPower: -1,
minValue: 0,
maxValue:1000,
currentValue: 0,
registerEventHandler: [
newMinValue: function(newVal) {
this.currentValue = newVal;
}
]
});

// add virtual short message phone number actuator
complexDev.addDevice(
{deviceName: "Short Message Alert Phone Number",
deviceType: "string",
currentValue: "",
onTargetChange: function(newVal) {
this.targetVlaue = newTarget;
this.currentValue = newTarget;
}
});
script.cvChanged = function(e,id,oldVal, newVal) {
if(newVal < minTemp) {
minTemp = newVal;
// inform other devices about a new min temp value
this.triggerEvent("newMinValue", newVal);
}
if(minTemp < minThreshold) {
// set alert phone number to real SMS modem dev.
$("#240001.setTargetValue(???);
// set alert text to real SMS modem device
$("#240002.setTargetValue("min temperature on
floor 2 is below " + minThreshold);
// send short message using real SMS modem dev.
$("#240003.setTargetValue(1);
}
};
});     // end of complex device definition
```

Fig. 12 Exemplary complex script device definitions

V. Conclusion

In the course of this paper we described mechanisms for device interoperability as applied in the OpenSB smart building system. We focused on device interoperability at platform layer and application layer.

At platform layer, device interoperability has been achieved is by means of a device model based on so called base types that allows any real device to be connected to the platform as the appropriate device driver on the OpenSB gateway translates real device functionality into a set of base-type platform devices. Furthermore, these platform devices can interact with each other by means of rule scripts that can be added, altered or removed at controller runtime by platform users or operators.

At application layer, the OpenSB controller provides an external REST-based interface that allows external applications and services to access the OpenSB data model, i.e. device representations and groups that devices might be assigned to. To allow external applications to control devices, the virtual

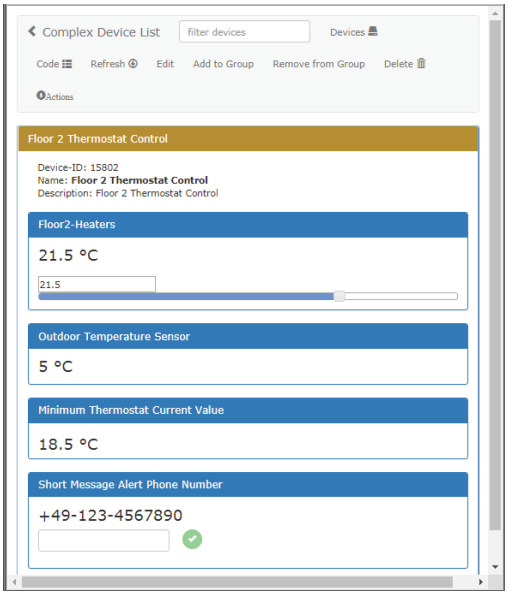

Fig. 13 Application view of complex device representation

device concept has been introduced. This allows users or operators to create virtual devices whose behavior can be defined by appropriate script rules. In this way, for external applications, for example, aggregated devices can be created, the use of which drives any number of real devices.

To further improve interoperability at the application level, complex script devices have been introduced that allow a combination of different platform device and virtual devices on the external interface. Thus, device representations can be created that meet the needs of users and operators at application level. In contrast to other systems, these complex devices scenarios can be created by users or operators by means of appropriate script code that can be added, changed or removed at controller runtime. Configuration at one place, i.e. in script code only, prevents scattered configuration and any kind of redundancy.

In the near future, further investigation in scalability and distributed rule processing are planned. Due to the distributed nature of the IoT gateways, we believe that there is significant potential for performance speed up in larger scale setups, e.g. in buildings with many IoT gateways. Currently, OpenSB implementation changes from UPnP to MQTT as application protocol between OpenSB controller and OpenSB gateways. This leads to new possibilities for aggregated device communication, i.e. summarizing the target value change operation of many devices connected to an IoT gateway. It is also planned to apply the concept of edge computing by allowing IoT gateways to participate in rule processing.

References

[1] I. T. Union, "Overview of the Internet of Things," I. T. Union, Recommendation ITU-T Y.2060, 2012.
[2] A. Gnther and T. Meier, "A modular system for building automation," in *Proceedings 55. International Scientific Colloquium*, TU Ilmenau, 2010.
[3] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things," in *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, Dec 2010, pp. 347–352.
[4] A. Mynzhasova, C. Radojicic, C. Heinz, J. Klsch, C. Grimm, J. Rico, K. Dickerson, R. Garca-Castro, and V. Oravec, "Drivers, standards and platforms for the iot: Towards a digital vicinity," in *2017 Intelligent Systems Conference (IntelliSys)*, Sept 2017, pp. 170–176.
[5] S. K. Lee, M. Bae, and H. Kim, "Future of iot networks: A survey," *Applied Sciences*, vol. 7, no. 10, 2017. (Online). Available: http://www.mdpi.com/2076-3417/7/10/1072.
[6] P. Masek, J. Hosek, K. Zeman, M. Stusek, D. Kovac, P. Cika, J. Masek, S. Andreev, and F. Kröpfl, "Implementation of true iot vision," *Int. J. Distrib. Sen. Netw.*, vol. 2016, Apr. 2016. (Online). Available: http://dx.doi.org/10.1155/2016/8160282.
[7] R. T. Fielding, "REST: architectural styles and the design of network-based software architectures," Doctoral dissertation, University of California, Irvine, 2000. (Online). Available: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm.
[8] "Zigbee alliance," http://www.zigbee.org/, accessed: 2018-04-18.
[9] "Mq telemetry transport," http://mqtt.org/, accessed: 2018-04-18.
[10] D. Thangavel, X. Ma, A. Valera, H. X. Tan, and C. K. Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, April 2014, pp. 1–6.
[11] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, and A. Riegg, "Internet of things patterns," in *Proceedings of the 21st European Conference on Pattern Languages of Programs*, ser. EuroPlop '16. New York, NY, USA: ACM, 2016, pp. 5:1–5:21. (Online). Available: http://doi.acm.org/10.1145/3011784.3011789.
[12] "Eclipse smarthome - a flexible framework for the smart home," https://www.eclipse.org/smarthome/, accessed: 2018-03-21.
[13] F. Heimgaertner, S. Hettich, O. Kohlbacher, and M. Menth, "Scaling home automation to public buildings: A distributed multiuser setup for openhab 2," in *2017 Global Internet of Things Summit (GIoTS)*, June 2017, pp. 1–6.
[14] "openhab - empowering the smart home," https://www.openhab.org/, accessed: 2018-03-24.
[15] "Edgex foundry a linux foundation project," https://www.edgexfoundry.org/, accessed: 2018-04-4.
[16] "Iotivity a linux foundation project," https://www.iotivity.org/, accessed: 2018-02-24.
[17] "relayr iot middleware platfrom," https://relayr.io/en/iot-middleware-platform/, accessed: 2018-04-17.
[18] Y. Sun, T. Y. Wu, G. Zhao, and M. Guizani, "Efficient rule engine for smart building systems," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1658–1669, June 2015.
[19] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and ide for internet of things applications," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 996–1001.
[20] "jquery write less, do more," http://jquery.com/, accessed: 2018-04-21.
[21] "Sizzle a javascript selector engine," https://sizzlejs.com/, accessed: 2018-04-21.
[22] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and ide for internet of things applications," in *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, May 2015, pp. 996–1001.