Nastic *et al. Journal of Internet Services and Applications* (2015) 6:21
DOI 10.1186/s13174-015-0037-1

Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

SDG-Pro: a programming framework for software-defined IoT cloud gateways

Stefan Nastic*, Hong-Linh Truong and Schahram Dustdar

Abstract

Recently, emerging IoT cloud systems create numerous opportunities for a variety of stakeholders in terms of optimizing their existing business processes, as well as developing novel cross-organization and cross-domain applications. However, developers of such systems face a plethora of challenges, mainly due to complex dependencies between the application business logic and the underlying IoT cloud infrastructure, as well as difficulties to provision and govern vast, geographically distributed IoT cloud resources. In this paper, we introduce SDG-Pro – a novel programming framework for software-defined IoT cloud systems. The main features of our framework include programming abstractions: Software-Defined Gateways, Intents, Intent Scopes, and Data and Control Points, as well as provisioning and governance APIs that allow for programmatic management of software-defined gateways throughout their entire lifecycle. The SDG-Pro framework enables easier, efficient and more intuitive development of IoT cloud applications. It promotes the everything-as-code paradigm for IoT cloud applications in order to provide a uniform, programmatic view on the entire development process. To illustrate the feasibility of our framework to support development of IoT cloud applications, we evaluate it using a real-world case study on managing fleets of electric vehicles.

Keywords: IoT cloud applications programming; Software-defined gateways; IoT cloud systems

1 Introduction

Emerging IoT cloud systems extend the traditional cloud computing systems beyond the data centers and cloud services to include a variety of edge IoT devices such as sensors and sensory gateways. Such systems utilize the IoT infrastructure resources to deliver novel value-added services, which leverage data from different sensor devices or enable timely propagation of decisions, crucial for business operation, to the edge of the infrastructure. On the other side, IoT cloud systems utilize cloud's theoretically unlimited resources, e.g., compute and storage, to enhance traditionally resource constrained IoT devices.

In order to facilitate development of IoT cloud systems, existing research and industry have produced numerous infrastructure, platform and software services as well as frameworks and tools [1–6]. These advances set a cornerstone for proliferation of (unified) IoT cloud platforms and infrastructures, which offer a myriad of IoT cloud capabilities and resources. Lately, we have been exploring software defined approaches and introduced a design methodology and a set of software defined principles for IoT cloud [7] in order to facilitate utility-oriented delivery of the IoT cloud resources, provide elasticity support for the IoT cloud systems and enable automated and logically centralized provisioning of the geographically distributed IoT cloud infrastructure. Generally, the software-defined IoT cloud systems abstract from low-level resources (e.g., hardware) and enable their programmatic management through well-defined APIs. They allow for refactoring the underlying IoT cloud infrastructure into finer-grained resource components whose functionality can be (re)defined in software, e.g., applications, thus enabling more efficient resource utilization and simplifying management of the IoT cloud systems. However, most of the contemporary approaches dealing with IoT cloud are intended for platform/infrastructure providers and operations managers. Therefore, from the developer's perspective there is a lack of structured, holistic

*Correspondence: snastic@dsg.tuwien.ac.at
Distributed Systems Group, TU Wien, Argentinierstrasse 8/184-1, 1040 Vienna, Austria

approach to support development of the IoT cloud systems and applications. Concrete abstractions and mechanisms, which enable efficient, more intuitive and scalable application development still remain underdeveloped.

1.1 Contributions

In this paper we introduce SDG-Pro – a novel programing framework for software-defined IoT cloud systems. The main contribution of the paper is SDG-Pro's programming model. It provides a unified, programmatic view for the entire development process (*everything as code*) of IoT cloud applications, thus making it easily traceable and auditable. We demonstrate the main advantages of our programming model in terms of easier, efficient and more intuitive application development, by using a real-world case study on managing fleets of vehicles.

This paper substantially extends and refines our previous work presented in [7, 8]. In [8], we introduced a programming model for developing cloud-based IoT services. The SDG-Pro framework extends that approach, by introducing programming support for developing the (edge) device services, i.e., monitoring and control tasks (Section 4.2). In [7], we introduced a conceptual model and main design principles for software-defined IoT cloud systems. The SDG-Pro framework builds on these concepts and extends our previous work by introducing comprehensive programming support for unified development, provisioning (Section 4.3) and governance of (Section 4.4) IoT cloud applications.

1.2 Paper organization

The remainder of the paper is structured as follows. In Section 2, we describe a motivating scenario and main research challenges; Section 3 outlines the design of the SDG-Pro framework; Section 4 presents the framework's programming model; In Section 5, we outline SDP-Pro's runtime support; Section 6 presents our experiments; Section 7 discusses the related work; Finally, Section 8 concludes the paper and gives an outlook of our future research.

2 Motivation and research challenges

2.1 Scenario

Let us consider a realistic application scenario in the domain of vehicle management that we will refer to throughout the paper.

Fleet Management System (FMS) is a real-world IoT cloud system responsible for managing fleets of electric vehicles deployed worldwide, e.g., on different golf courses. The vehicles are equipped with an *on-board device*, acting as a gateway to vehicles' sensors and actuators, as well as offering resources to execute device services in the vehicles. The vehicles communicate with the cloud via 3G or Wi-Fi networks to exchange telematic and diagnostic data. On the cloud, FMS provides different applications and services to manage this data. Relevant services include realtime vehicle status, remote diagnostics, and remote control. In general, different stakeholders rely on the FMS applications to manage their portion of the fleet and optimize tasks specific to their business model.

The cloud plays a crucial role for the FMS due to several reasons. Besides utilizing the edge infrastructure, e.g., vehicle sensors and the on-board devices, the FMS heavily relies on cloud infrastructure to be able to process and reliably store vast amounts of sensory data as well as to connect large number of vehicles and provide centralized, simultaneous access to the geographically distributed fleet (needed by the services such as emergency remote fleet control). Further, since different stakeholders manage different portions of the fleet, i.e., are allowed to access specific vehicles and their data, FMS has to be able to support multiple tenants. In addition, since many of FMS's services can be elastically scaled down in off peak times, e.g., during the night, the elastic nature of cloud plays a significant role, especially in terms of costs control, since systems of such scale as our FMS incur very high costs in practice (e.g., of computation or networking).

The FMS runs atop a complex IoT cloud infrastructure, which includes a variety of IoT cloud resources. Figure 1 gives a high-level overview of the common elements in FMS's architecture and deployment. FMS deployment topologies span across the entire IoT cloud infrastructure, i.e., from large data centers to the edge of the network, resulting in complex dependencies among the business logic services, but also between such services and the underlying infrastructure. Therefore, developers need to consider numerous infrastructure resources and their properties such as availability of sensors, devices ownership and their location.

The FMS applications perform a variety of analytics and are mostly characterized by a reactive behavior. They receive (monitoring) data, e.g., a change in vehicles' operation and, as a response, perform (control) actions. Such monitoring and control tasks are executing in heterogeneous, dynamic FMS environment and interact with many geographically distributed vehicles and their low-level capabilities, e.g., engine control points. Further, FMS applications have different requirements regarding communication protocols. For example, the fault alarms need to be pushed to the services, e.g, via MQ Telemetry Transport (MQTT) and vehicle's diagnostics should be synchronously accessed via RESTful protocols such as Constrained Application Protocol (CoAP) or Simple Measurement and Actuation Profile (sMAP). Therefore, from the developers perspective such tasks and capabilities need to be decoupled from the underlying physical

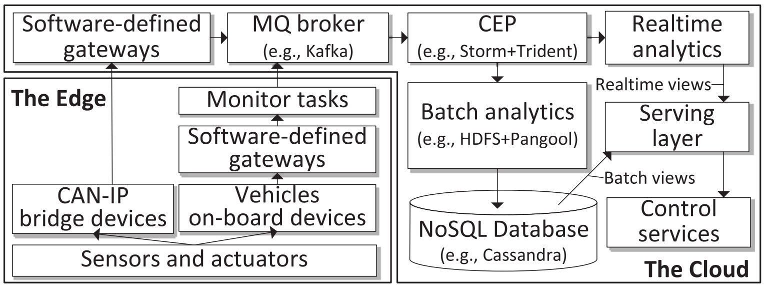
Software-defined gateways
MQ broker (e.g., Kafka)
CEP (e.g., Storm+Trident)
Realtime analytics
The Edge
Monitor tasks
Software-defined gateways
CAN-IP bridge devices
Vehicles on-board devices
Sensors and actuators
Batch analytics (e.g., HDFS+Pangool)
Realtime views
Serving layer
Batch views
NoSQL Database (e.g., Cassandra)
Control services
The Cloud

Fig. 1 Overview of FMS architecture and deployment

infrastructure, but also easily specified, provisioned and managed programmatically (as code), without worrying about the complexity of low-level device services, communication channels and raw sensory data streams.

The currently limited development support regarding the FMS requirements and features (as discussed in Section 7), renders the development of its applications a complex task. Consequently, system designers and application developers face numerous challenges to design and develop IoT cloud applications.

2.2 Research challenges

RC-1 – The development context of IoT cloud applications has grown beyond writing custom business logic (e.g., services) components to also considering the involved IoT devices (e.g., their capabilities) as well as the deployment and provisioning of such services across the IoT cloud infrastructure. The main reasons for this are complex and strong dependence of the business logic on the underlying devices (and their specific capabilities), novel (resource) features that need to be considered, such as device location and the heterogeneity of the utilized IoT cloud resources. Unfortunately, developers currently lack suitable programming abstractions to deal with such concerns in a unified manner, from early stages of development.

RC-2 – IoT cloud applications execute in very dynamic, heterogeneous environments and interact with hundreds or thousands of physical entities. Therefore, monitoring and controlling these entities in a scalable manner is another challenge for developers of IoT applications, mainly because they need to dynamically identify the scope of application's actions, depending on the task at hand, but also express its business logic independently of the low-level device capabilities.

RC-3 – The IoT cloud applications mostly rely on common physical infrastructure. However, IoT cloud infrastructure resources are mostly provided as coarse-grained, rigid packages. The infrastructure components and software libraries are specifically tailored for the problem at hand and do not allow for flexible customization and provisioning of individual resource components or runtime topologies. This inherently hinders self-service, utility-oriented delivery and consumption of IoT cloud resources at fine granularity levels.

RC-4 – Due to dynamicity, heterogeneity and geographical distribution of IoT cloud, traditional provisioning and governance approaches are hardly feasible in practice. This is mostly because they implicitly make assumptions, such as physical on-site presence, manual logging into devices, understanding device's specificities, etc., which are difficult, if not impossible, to achieve in IoT cloud systems. In spite of this, techniques and abstractions, which provide a programmatic, conceptually centralized view on system provisioning and runtime governance are largely missing.

In the rest of the paper, we introduce our SDG-Pro framework and focus on describing and evaluating its programming model for IoT cloud applications.

3 The SDG-Pro framework

The main aim of our SDG-Pro (*Software-Defined Gateways Programing framework*) is to provide programming support for IoT cloud application developers, which offers a set of adequate programming abstractions to facilitate overcoming the aforementioned challenges. To this end, SDG-Pro, enables expressing application's provisioning, governance and business logic programmatically, in a uniform manner. By raising the level of programming abstraction, SDG-Pro reduces the complexity of application development, makes the development process traceable and auditable and improves efficacy and scalability of application development.

SDG-Pro does not propose a novel software-defined approach for IoT cloud systems. It builds on the design principles that were elicited in our previous research [7, 9, 10] and adopts development methodology we proposed in [11], extending them to provide programming abstractions that facilitate development of the essential application artifacts. SDG-Pro's programming model is designed to enforce the main design principles of software-defined IoT cloud systems at application level, from the early development stages.

3.1 Main design principles and development methodology for software-defined IoT cloud systems

As we have shown in [7], software-defined IoT cloud systems comprise a set of resource components, provided by IoT cloud infrastructure, which can be provisioned and governed at runtime. Such resources (e.g., sensory data streams), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols and data point controllers) are described as software-defined IoT units.

The software-defined gateways (cf. Fig. 1) are a special type of such units and they are the main building blocks of IoT cloud infrastructure, e.g., similar to VMs in cloud computing. In our conceptual design of software-defined IoT cloud systems, such gateways abstract resource provisioning and governance through well-defined APIs and they can be composed at different levels, creating virtual runtime topologies for IoT cloud applications. This enables opening up the traditional infrastructure silos and moving one step higher in the abstraction, i.e., effectively making applications independent of the underlying rigid infrastructure. As we have extensively discussed in our previous work [7, 9, 10], the main design principles of software-defined IoT systems include:

Everything as code – All the concerns, i.e., application business logic, but also IoT cloud resources provisioning and runtime governance, should be expressed programmatically in a unified manner, as a part of the application's logic (code).
Scalable development – The programming abstractions exposed to the developers need to support scalable application development, i.e., shield the developers from dealing with the concerns such as manually referencing individual devices or managing the low-level data and control channels.
API Encapsulation – IoT cloud resources and capabilities are encapsulated in well-defined APIs, to provide a uniform view on accessing functionality and configurations of IoT cloud infrastructure.
Declarative provisioning – The units are specified declaratively and their functionality is defined programmatically in software, using well-defined API and available, familiar software libraries.
Central point of operation – Enable conceptually centralized (API) interaction with the software-defined IoT cloud system to allow for a unified view on the system's operations and governance capabilities (available at runtime), without worrying about low-level infrastructure details.
Automation – Main provisioning and governance processes need to be automated in order to enable dynamic, on-demand configuring and operating the software-defined IoT cloud systems, without manually interacting with IoT devices (e.g., logging in the gateways).

As proposed in [11], building IoT cloud systems includes creating and/or selecting suitable software-defined IoT units, provisioning and composing more complex units and building custom business logic components. This (iterative) development process is structured along four main phases (cf. Fig. 2): i) Initially, developers need to design and implement the software-defined IoT units or obtain them form a third-party, e.g., in a market-like fashion. Among other things the IoT units support execution of the light-weight device services (monitor and control tasks), i.e., from the software engineering perspective they encapsulate such tasks, comprising domain libraries; ii) Next, the developers need to design and provision the required application topologies. This process includes implementing the dependencies among the business logic services, but also between such services and the underlying infrastructure; iii) Building custom business logic components mainly involves developing device services and implementing reactive business logic (e.g., cloud services) around the device services; iv) The developers implement operational governance logic for managing the IoT units during application runtime.

3.2 SDG-Pro architecture overview

Generally, the SDG-Pro framework is distributed across the clouds, i.e., large data centers, and "small" IoT devices,

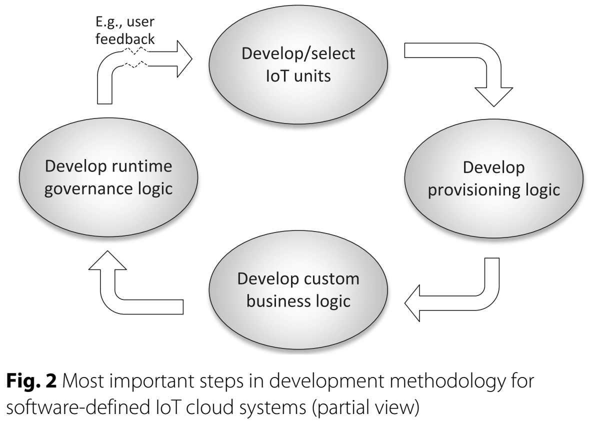
E.g., user feedback
Develop/select IoT units
Develop provisioning logic
Develop custom business logic
Develop runtime governance logic

Fig. 2 Most important steps in development methodology for software-defined IoT cloud systems (partial view)

e.g., physical gateways or cloudlets. It is designed based on the microservices architecture, which among other things enables evolvable and fault-tolerant system, while allowing for flexible management and scaling of individual components. Figure 3 gives a high-level overview of SDG-Pro's architecture and main IoT cloud application artifacts. These artifacts can be seen as executables produced by the aforementioned development methodology. To support the development of such artifacts our framework provides a set of programming abstractions (depicted as gray components in Fig. 3 and described later in Section 4) and runtime support mechanisms (Section 5).

The runtime mechanisms are part of the SDG-Pro's Runtime support (cf. Fig. 3), which underpins the programming abstractions exposed to the developers, i.e., provides an execution environment for IoT cloud applications. It takes over a set of responsibilities such as placement of the software-defined gateways, their runtime migration and elasticity control, infrastructure topology management and application scope coordination. By doing so, it does most of "heavy lifting" on behalf of the applications, thus supporting the developers to easier cope with the diversity and geographical distribution of the IoT cloud and enabling better utilization of the numerous edge devices.

Internally, our framework's runtime support comprises several microservices, which can be grouped into: API-Manager, IoT units management layer, Intents runtime container and Operational governance layer. The *API-Manager* exposes governance capabilities and the low-level data and control channels from the IoT cloud infrastructure to the applications via well-defined APIs and handles all API calls from such applications. The *IoT units management layer* provides mechanisms and agents to support instantiating, provisioning and deploying software-defined gateways programmatically and on-demand. The *Intents runtime container* is responsible to handle incoming application requests (Intents) and select, instantiate and invoke device services (tasks), based on the information provided in the intents. It also enables applications to dynamically delimit the scopes of their actions, by providing support for IntentScopes resolution. Finally, the *Operational governance layer* supports execution of the governance processes by enabling remote invocation of governance capabilities and mapping of API calls on underlying devices via governance agents.

4 SDG-Pro's programming model

4.1 Structure of IoT cloud applications

The main purpose of our programming model is to provide a programmatic view on the whole application ecosystem, i.e., the full stack from the infrastructure to software components and services. The main principle behind our programming model is *everything as code*. This includes providing support for writing IoT cloud applications' business logic, as well as representing the underlying infrastructure components (e.g., gateways) at the application level and enabling developers to programmatically determine their deployment and provisioning. Figure 4 shows a component diagram with the logical structure of IoT cloud applications. The main components of such application include: custom business

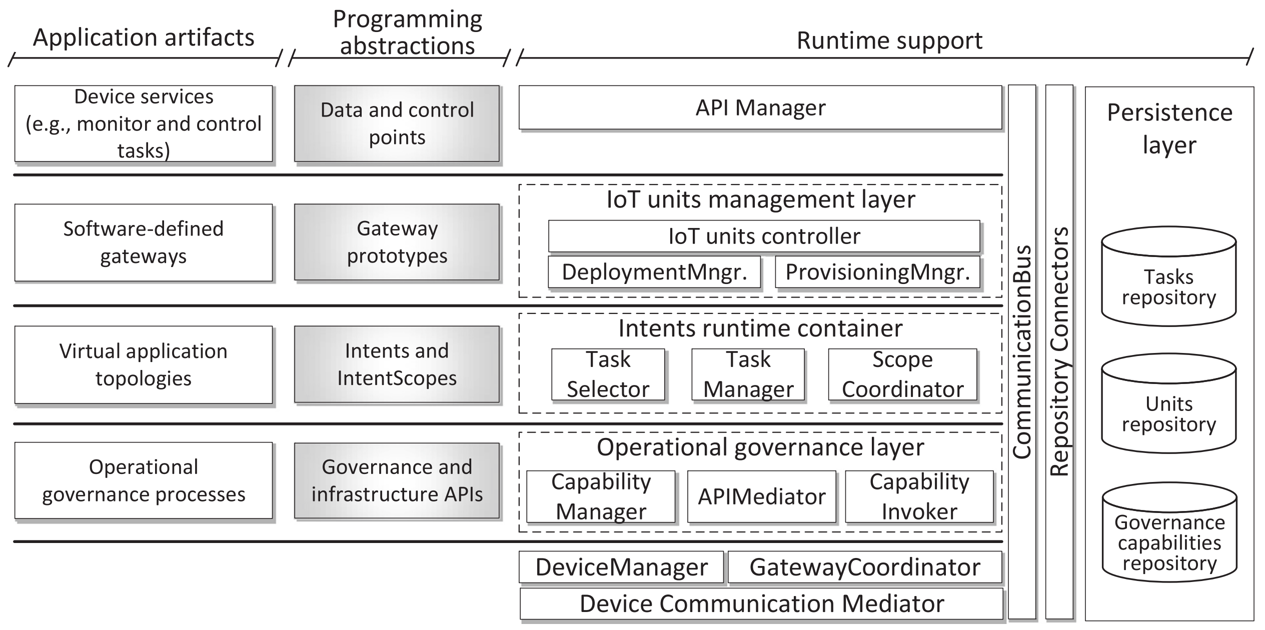

Fig. 3 Overview of SDG-Pro's architecture and main artifacts of IoT cloud applications

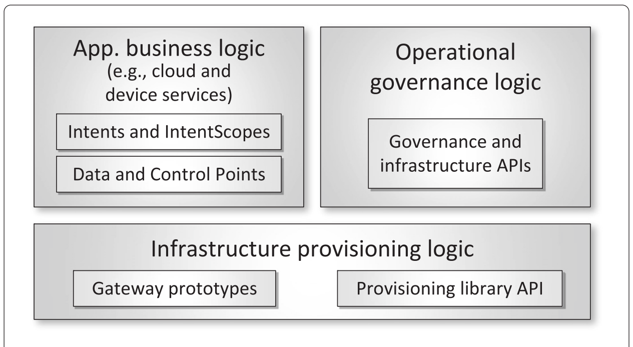
App. business logic
(e.g., cloud and device services)
Intents and IntentScopes
Data and Control Points
Operational governance logic
Governance and infrastructure APIs
Infrastructure provisioning logic
Gateway prototypes
Provisioning library API

Fig. 4 Overview of IoT cloud application structure

logic components (e.g., cloud services and device services); resource provisioning and deployment logic (custom or stock component provisioning); and operational governance logic.

4.2 Programming support for business logic services

In SDG-Pro we distinguish between two types of business logic services: device-level services and cloud services. Device-level services are executed in IoT devices and implement control and monitor tasks. For example, a monitoring task includes processing, correlation and analysis of sensory data streams. To support task development, the SDG-Pro framework provides *data and control points*, which are described later in this section.

The cloud services usually define virtual service topologies by referencing the tasks. At the application level, we provide explicit representation of these tasks via *Intents*, i.e., developers write *Intents* to dynamically configure and invoke the tasks. Further, developers use *IntentScope*s to delimit the range of an *Intent*. For example, a developer might want to code the expression: “stop all vehicles on golf course X”. In this case, “stop” is the desired *Intent*, which needs to be applied on an *IntentScope* that encompasses all vehicles with the location property “golf course X”.

Intents Intent is a data structure describing a specific task which can be performed in a physical environment. In reality, *Intent*s are processed and executed on the cloud platform, but enable monitoring and controlling of the physical environments. Based on the information contained in an *Intent*, a suitable task is dynamically selected, instantiated and executed.

Depending on the task’s nature, we distinguish between two different types of *Intent*s: *ControlIntent* and *MonitorIntent*. *ControlIntent*s enable applications to operate and invoke the low-level components, i.e., provide a high-level representation of their functionality. *MonitorIntent*s are used by applications to subscribe for events from the sensors and to obtain devices’ context.

Figure 5 shows the *Intent* structure and its most relevant parts. Each *Intent* contains an ID, used to correlate invocation response with it or apply additional actions on it. Additionally, it contains a set of headers, which specify meta information needed to process the *Intent* and bind it with a suitable task during the runtime. Among other things, headers carry *Intent’s* name and a reference to an *IntentScope*. Further, an *Intent* can contain a set of attributes, which are used by the runtime to select the best matching task instance in case there are multiple Intent implementations available. Finally, *Intent* can contain data, which is used to configure the tasks or supply additional payload. Generally, *Intents* allow developers to communicate to the system what needs to be done instead of worrying how the underlying devices will perform the specific task.

IntentScopes IntentScope is an abstraction, which represents a group of physical entities (e.g., vehicles) that share some common properties, i.e., a set of software entities in the cloud, which represent corresponding physical entities. The IntentScopes enable developers to dynamically delimit physical entities on which an *Intent* will have an effect. The SDG-Pro framework provides mechanisms to dynamically define and work with *IntentScope*s on the application level.

To define an *IntentScope* developers specify properties, which need to be satisfied by the physical entities to be included in the scope. To enable *IntentScope* bootstrapping, we provide a special type of *IntentScope*, which is called *GlobalScope*. It defines the maximal scope for an application and usually contains all physical entities administered by a stakeholder at the given time. Therefore, it is reasonable to assume that the *GlobalScope* is slow-changing over time and it can be configured by a user, e.g., a golf course manager. Our programming model allows *IntentScopes* to be defined explicitly and implicitly, i.e., developers can explicitelly add entities to the scope by specifying their IDs or recursively prune the *GobalScope*.

Formally, we use the well-known set theory to define *IntentScope* as a finite, countable set of entities (set elements). The *GlobalScope* represents the universal set, denoted as S^{max}, therefore, $\forall S(S \subseteq S^{max})$, where S is an *IntentScope*, must hold. Further, for each entity E in the system general membership relation $\forall E(E \in S | S \subseteq S^{max})$, must hold. Therefore, an entity is the unit set, denoted as S_{min}. Empty set $\emptyset$ is not defined, thus, applying an *Intent* on it results with an error. Finally, a necessary condition for an *IntentScope* to be valid is: *IntentScope* is valid iff it is a set S, such that $S \subseteq S^{max} \wedge S \not\equiv \emptyset$ holds. Equation 1 shows operations used to define or refine an *IntentScope*. The most interesting operation is $\subseteq_{cond}\ S$. It is used to find

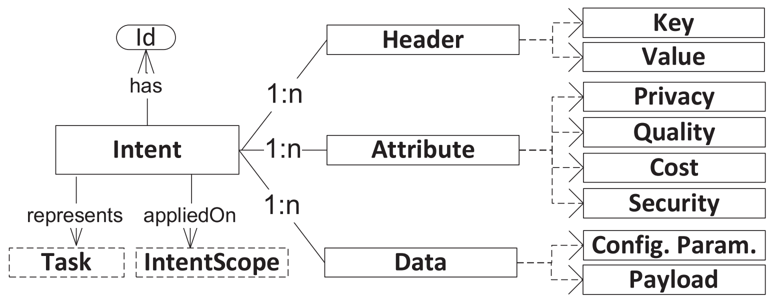

Fig. 5 Intent structure

a subset ($\hat{S}$) of a set S, which satisfies some condition, i.e., $E \in \hat{S} \mid E \in S \wedge cond(E) = True$.

$$S = S_{min}|S^{max}| \subseteq_{cond} S|S \cup S|S \cap S|S \setminus S \quad (1)$$

Listings 1 and 2 show example usage of Intents and IntentScopes.

Listing 1 Example usage of MonitorIntent and GlobalScope

```
Intent eFault = Intent.newMIntent("EnergyFault");
//monitor whole fleet
eFault.setScope(IntentScope.getGlobal());
notify(eFault,this);//invoke task
//callback function called on event arrival
public void onEvent(Event e){//perform some action}
```

Listing 2 Example usage of ControlIntent and IntentScope

```
IntentScope cs = delimit(IntentScope.getGlobal(),
 Cond.isTrue(eFault)); //eFault defined in Listing1
Intent eCons = Intent.newCIntent("ReduceEnergy");
eCons.setScope(cs);//set intent scope
eCons.set("speed").value("5");
eCons.set("RPM").value("1100");
send(eCons); //invoke task
```

Data and Control points The main purpose of the data and control points is to support development of the light-weight device services. Generally, they represent and enable management of data and control channels (e.g., device drivers) to the low-level sensors/actuators in an abstract manner. Data and control points mediate the communication with the connected devices (e.g., digital, serial or IP-based) and also implement communication protocols for the connected devices, e.g., Modbus, CAN or SOX/DASP.

The data and control points enable developers to interact with sensory data streams and actuating functionality in a unified manner, independent of communication type, e.g., protocol. The most important concept behind data and control points are the *virtual buffers*, which are provided and managed by our framework. In general, such buffers enable virtualized access to and custom configurations of underlying sensors and actuators. They act as multiplexers of the data and control channels, thus enabling the device services to have their own view of and define custom configurations for such channels. For example, an application can configure sensor poll rates, activate a low-pass filter for an analog sensory input or configure unit and type of data instances in the stream.

Listing 3 Example usage of data point

```
DataPoint dataPoint = new DataPoint();
// Query available buffers
Collection<BufferDescription> availableBuffers
 = dataPoint.queryBuffers(new SensorProps(...));
// Assign the buffers to the data point
dataPoint.assign(availableBuffers.get(0));
dataPoint.setPollRate(5);
```

Listing 3, gives a general example of how to define a data point. It shows a data point with one stream of simple data instances that represent, e.g., vehicle's tire speed, based on the required sensor properties. By default the data points are configured to asynchronously push the data to the applications at a specific rate, which can be configured as shown in the example. The application defines a callback handler, which contains some data processing logic, e.g., based of complex event processing techniques. Additionally, the data and control points offer a `read` operator that can be used to sequentially (or in batch) read a set of instances from a stream, e.g., in order to perform more complex stream processing operations.

4.3 Programmatic infrastructure provisioning with software-defined gateways

The most important abstraction for provisioning IoT cloud infrastructure is the software-defined gateway. In our programming model software-defined gateways are treated as first-class citizens. This allows the developers to specify, manipulate and manage the IoT cloud infrastructure resources programmatically from within the application logic.

Generally, provisioning part of the application logic is used to programmatically specify the infrastructure dependencies, i.e., the state of the infrastructure required by the business logic services to execute correctly. To this end, our framework supports the developers to perform two main tasks. First, the developers can programmatically define the software-defined gateways and specify their internal structure. Second, our framework supports the developers to deploy such gateways atop IoT cloud (e.g., data centers or physical IoT gateways) form within the application logic. Therefore, provisioning logic is specified in software enabling the infrastructure dependencies and requirements to be defined dynamically and on-demand.

Figure 6 shows the typical structure of a software-defined gateway. We notice two important properties of software-defined gateways. First, to technically realize software-defined gateways SDG-Pro offers gateway prototypes. These are resource containers, used to bootstrap more complex, higher-level gateway functionality. Generally, they are hosted atop IoT cloud and enriched with functional, provisioning and governance capabilities, which are exposed via well-defined APIs. Currently, our framework supports software-defined gateways based on kernel supported virtualization, but virtualization choices do not pose any limitations, because, by utilizing the well-defined API, our gateway prototypes can be dynamically configured, provisioned, interconnected, deployed, and controlled.

Second, developers use the software-defined gateways to programmatically provision and deploy required application services, but also to configure an execution environment for such services. Therefore, by utilizing the provisioning APIs, developers can customize the software-defined gateways to exactly meet the application's functional requirements. For example (Fig. 6), they can dynamically configure a specific cloud communication protocol, e.g., CoAP or MQTT, select services runtime, e.g., Sedona VM or configure data and control points, e.g., based on Modbus.

In order to provision a software-defined gateway, initially the developers need to specify the software defined gateway prototypes. Listing 4 illustrates how the gateway prototypes are programmatically defined with our framework. In this example, a software-defined gateway is created from a gateway prototype, based on BusyBox. In the background the framework creates a Linux container and installs the provisioning and governance agents on it (more details about this process are given in Section 5). In general, the agents expose the provisioning APIs, which are activated and available at that point. Afterwards, a developer provides a configuration model for the gateway. In this example the gateway is configured to be deployed on a specific host by setting the "host address". In case it is not set the framework uses the deployment class to determine the gateway placement. Finally, the developer specifies the class that contains the internal provisioning logic.

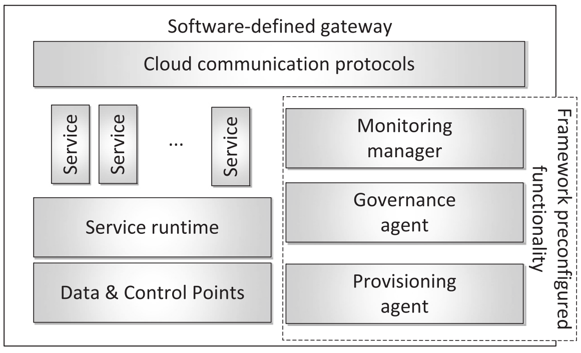

Fig. 6 Overview of software-defined gateway structure

Listing 4 Example of software-defined gateway prototype

```
//Define and configure a gateway
SDGateway gateway
= UnitsController.create(GType.BUSYBOX);
gateway.setId("gateway-X");
gateway.setHost("http://host_address");
gateway.setMetaData(Deployment.class);
gateway.addConfigClass(Provisioning.class);
```

Listing 5 illustrates our framework's support for dynamic provisioning of such gateways. The gateway provisioning logic contains the directives to internally provision the gateway, e.g., to install and configure device services, cloud communication libraries and data and control points. To this end, developers can use the framework's provisioning support, which contains the APIs, provided by the provisioning agent, and a provisioning library comprising a number of functions that facilitate provisioning of the software-defined gateways. In this example, we show how to provision a gateway with Sedona runtime.

Listing 5 Example of gateway provisioning API

```
String dest = ".../G2021/svm";
provisioner.CreateDirIfMissing(dest);
provisioner.CopyToDir("sedona-vm-1.2.28/svm", dest);
provisioner.setPermissions(dest, "a+x");
```

4.4 Programmatic governance and infrastructure APIs

4.4.1 Programmatic application governance

After an application is provisioned and deployed, a new set of runtime concerns emerges, e.g., dynamically reconfiguring sensor update rates or elastically scaling software-defined gateways. In order to address such concerns, application developers implement operational governance processes (cf. Fig. 4).

In our previous work [9, 10] we introduced a general approach for runtime operational governance in software-defined IoT cloud systems, as well as the concepts of operational governance processes that manipulate the states of software-defined gateways at runtime. Such processes can be seen as a sequence of operations, which perform runtime state transitions from a current state to some desired target state (e.g., that satisfies some non-functional properties, enforces compliance, or exactly meets custom requirements).

The core abstraction behind the operational governance business logic are *governance capabilities*. Generally, the governance capabilities represent the main building blocks of operational governance processes and they are usually executed in IoT devices. The governance capabilities encapsulate governance operations which can be applied on deployed software-defined gateways, e.g., to query the current version of a service, change a communication protocol, or spin up a virtual gateway. The framework enables such capabilities to be dynamically added to the system and supports managing their APIs. Generally, we do not make any assumptions about concrete capability implementations. However, the framework requires them to be packaged as shown in Fig. 7.

To enable programmatic operational governance our framework offers governance APIs that are used by application developers to install, deploy, manage and invoke the governance capabilities. Listing 6, shows examples of operational governance APIs exposed by our framework. In general, the operational governance processes are defined as a sequence of such API calls, performed by the IoT cloud applications.

Listing 6 Examples of operational governance APIs

```
/* General case of capability invocation. */
/deviceId/{capabilityId}/{methodName}/{arguments}?
arg1={first-argument}&arg2={second-argument}&...

/* Data points capability invocation example. */
deviceId/DPcapa/setPollRate/arguments&rate=5s
/deviceId/DPcapa/list

/* Capabilities manager examples. */
/deviceId/cManager/capabilities/list
/deviceId/cManager/{capabilityId}/stop
```

4.4.2 Intents API operators

Intent is a passive data structure. Therefore, we need to provide developers with operators to work with the *Intent*s. These operators encapsulate mechanisms to *select, instantiate and execute* tasks, based on the input *Intent*. Consequently, instead of dealing with the individual tasks, a developer is presented with a unified interface to communicate with the runtime systems.

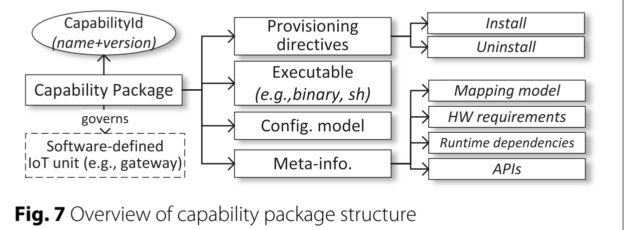

Fig. 7 Overview of capability package structure

Listing 7 shows the core API operators that support working with Intents and IntentScopes.

Listing 7 Core Intent API operators

```
send(in ci:ControlIntent,out r:Result)
notify(in mi:MonitorIntent,in o:CallBackObj)
poll(in mi:MonitorIntent,out el:List<Event>)
delimit(in s:IntentScope,in c:Cond,out so:IntentScope)
```

The `send` primitive is used to communicate and execute a ControlIntent. When the `send` operator is invoked the container first selects suitable tasks to execute the ControlIntent by using Intent's headers. The task list is further filtered, based on Intent's attributes, e.g., quality requirements. Here, we use best-effort to find the best matching task implementation. Further, the selected task is configured with Intent's configuration parameters and a payload, and finally executed.

The core operators `notify` and `poll` are used to support working with the MonitorIntents. The operator `notify` is used by an application to subscribe for events, which are asynchronously delivered to the application. The `poll` operator is used to synchronously check the status of the environment, i.e., it will block application's main thread if the required event is currently unavailable.

The `delimit` operator is API equivalent of $\subseteq_{cond}$, defined in Section 4.2. It is used to define an IntentScope with entities, which satisfy a certain condition. Usually, when an application wants to determine the IntentScope, it will start by invoking `delimit` on the GlobalScope and further refine it by recursively applying this operator and/or using other scope operators.

5 SDG-Pro runtime mechanisms

In the current implementation of the SDG-Pro framework we provide a set of runtime mechanisms that underpin the programming abstractions (Section 4) and support application execution atop the IoT cloud. Generally, application execution includes instantiating, provisioning and deploying software-defined gateways; dynamically loading device services atop the gateways; instantiating virtual application topologies (with Intents and IntentScopes); and triggering execution of operational governance processes (on-demand, depending on the business logic). Next, we discuss the design and implementation of the most important SDG-Pro's runtime mechanisms in more detail.

5.1 Instantiating, provisioning and deploying software-defined gateways

Currently SDG-Pro supporta a version of software-defined gateways (prototypes), which is based on Linux Containers (LXC). When a developer instantiates a gateway prototype (e.g., as shown in Listing 4), the *IoT units controller* (cf. Fig. 3) performs three main tasks. First, it creates an instance of LXC and installs the *provisioning* *and governance agents* in the container. Second, the provisioning agent[1] executes the provisioning directives, supplied in a provisioning script (e.g., Listing 5). Finally, the *IoT units controller* deploys the gateway instance in IoT cloud.

Firstly, to instantiate a software-defined gateway our framework relies on Docker[2], i.e., more specifically on Docker deamon that offers a remote API for programmatic container creation. To bootstrap the instantiation, SDG-Pro provides a custom base image, which we developed atop a BusyBox user land on a stripped-down Linux distribution. In SDG-Pro, the *DeviceManager* is based on the Docker remote API, but it provides additional support for configuring and managing containers such as specifying the custom meta information (e.g., location) to provide more control over the software-defined gateways at the application level. As the last part of gateway instantiation, SDG-Pro installs its provisioning and governance agents that support execution of the subsequent phases.

Provisioning a software-defined gateway includes configuring, deploying and installing different artifacts such as device services, libraries (e.g., cloud communication protocols) and other binaries atop the newly created gateway instance. In the first step of the provisioning process, the *ProvisioningManager* creates artifacts image. In essence, it is a (compressed) set of component binaries and provisioning scripts. Next, the *DeploymentManager* places the image in the update queue. The provisioning agent periodically inspects the queue for new updates and when it is available the agent polls the image in the gateway (container) in a lazy manner. Additionally, SDG-Pro allows the components to be asynchronously pushed to the gateways, similarly to eager object initialization. Finally, the agent interprets provisioning scripts, i.e., performs a local installation of the binaries and executes any custom configuration directives.

Lastly, the SDG-Pro framework selects an IoT cloud node, i.e., an edge device or a cloud VM, and deploys the gateway instance on it. The main component responsible for gateways (containers) allocation and deployment is the *GatewayCoordinator*. In the current prototype, the GatewayCoordinator is built based on `fleet` and `etcd`. The `fleet` is a distributed init system that provides the functionality to manage a cluster of host devices, e.g., the IoT cloud nodes. The `etcd` is a distributed key/value store that supports managing shared configurations among such nodes. In order to allocate a gateway, i.e., select the best matching node in the IoT cloud, the GatewayCoordinator compares the available gateway attributes (e.g., location, ownership, node type, etc.) with the meta data of the available IoT cloud nodes. The gateway's meta data is obtained from a developer-specified configuration model. The nodes' meta data is provided by the *DeviceManager* and it is mostly maintained manually, e.g., by system administrators. At the moment, we only provide a rudimentary support for gateway allocation, i.e., SDG-Pro only considers static node properties. In the future, we plan to address this issue by including support for dynamic properties such as available bandwidth and providing support for runtime migration (reallocation) of software-defined gateways. Finally, after a node is selected, the GatewayCoordinator invokes the `fleet` to deploy the gateway on that node.

5.2 Intent-based invocation and IntentScope resolution

In the SDG-Pro framework, the communication among the main application components is performed via Intents. Generally, it follows a partial content-based publish/subscribe model and in the current prototype it is based on the Apache ActiveMQ JMS broker.

When an application submits a new Intent to SDG-Pro's *RuntimeContainer*, it first routes the Intent to the *TaskSelector*, which matches intent headers with device services (task) filters to find suitable services that match the Intent. Afterwards, the TaskSelector reads the Intent attributes and compares them with the task filters to find the best matching task. The attributes are represented as feature vectors and a multi-dimensional utility function, based on the Hamming distance, is used to perform the matching. Afterwards, the TaskSelector requests a service instance, by providing its description to the the TaskManager. It checks the validity of the mapping and, if it is valid, invokes the corresponding service. If no service is available the Intent is marked as failed and the invoker is notified.

In a more general case, when an Intent gets invoked on an IntentScope, the aforementioned invocation process remains the same, with the only difference that our framework performs all steps on a complete IntentScope, in parallel, instead on an individual gateway. To this end, the *ScopeCoordinator* provides dynamic resolution of the IntentScopes.

The IntentSope specifications are implemented as composite predicates which reference device meta information and profile attributes. The predicates are applied on the GlobalScope (Section 4.2), filtering out all resources that do not match the provided attribute conditions. The ScopeCoordinator uses the resulting set of resources to initiate the Intent mapping and invocation. The ScopeCoordinator is also responsible to provide support for gathering results delivered by the invoked device services. This is needed since the scopes are resolved in parallel and the results are asynchronously delivered by the software-defined gateways.

5.3 Invocation of runtime governance capabilities

As shown in Section 4.4, application developers define operational governance logic as a sequence of API calls to the governance capabilities. The *APIManager* is responsible to mediate (map) these invocations to the underlying infrastructure, i.e., the software-defined gateways. To this end it relies on the *CapabilityManager*, which is a cloud-based service and the governance agent, which is a lightweight HTTP deamon, preinstalled in software-defined gateway prototypes.

When an API request is submitted by an application, SDG-Pro performs following steps: it registers the capability, maps the API call, executes the capability, and returns the result. First, the APIManager registers the API call with the corresponding capability. This involves querying the capability repository to obtain its meta-information (such as expected arguments), as well as building a dynamic mapping model, which includes capability ID, a reference to a runtime environment (e.g., Linux shell), input parameters, the result type, and further configuration directives. The CapabilityManager forwards the model to the gateways (i.e. the governance agent) and caches this information for subsequent invocations. During future interactions, the framework acts as transparent proxy, since subsequent steps are handled by the underlying gateways. In the next step, the governance agent needs to perform a mapping between the API call and the underlying capability. By default, it assumes that capabilities follow the traditional Unix interaction model, i.e., that all arguments and configurations (e.g., flags) are provided via the standard input stream (stdin) and output is produced to standard output (stdout) or standard error (stderr) streams. This means, if not specified otherwise in the mapping model, the framework will try to invoke the capability by its ID and will forward the provided arguments to its stdin. For capabilities that require custom invocation, e.g., property files, policies, or specific environment settings, the framework requires a custom mapping model. This model is used in the subsequent steps to correctly perform the API call. Finally, the governance agent invokes the governance capability and as soon as the capability completes it collects and wraps the result. Currently, the framework provides means to wrap results as JSON objects for standard data types and it relies on the mapping model to determine the appropriate return type. However, this can be extended to support generic behavior, e.g., with Google Protocol Buffers.

6 Evaluation

6.1 Evaluation methodology

In this section we present a functional evaluation of the paper's main contribution – the SDG-Pro's programming model for IoT cloud applications. To validate SDG-Pro's programming model we follow evaluation design guidelines provided in [12]. The main objective of our qualitative analysis is twofold. First, to show that SDG-pro facilitates dealing with the challenges of designing and developing IoT cloud applications (RC1-RC4), we demonstrate how our programming model enforces the main design principles of IoT cloud systems, as justified in Section 3.1. Second, in order to show that SDG-Pro enables easier, efficient and more intuitive development of IoT cloud applications, we compare it against traditional programming model evaluation criteria that include: *readability, code simplicity, reusability, expressiveness* and *functional extensibility*.

Although analysis of non-functional properties of the runtime mechanisms (e.g., regarding system's scale) is not the main focus of this paper, we refer interested reader to our previous work (e.g., [10, 13]), where we partly show their performance.

6.2 Examples of FMS applications and services

To demonstrate the most important concepts and features of SDG-Pro's programming model, we present a set of real-life applications from our FMS system (Section 2). This example suite is designed to cover typical interactions and requirements of IoT cloud applications, such as realtime monitoring and data analytics, remote actuation and control, autonomous device tasks and offline data analytics, in order to show the *completeness* of SDG-Pro's programming model regarding its support w.r.t. the real-life requirements. The example applications are developed and deployed atop a virtualized IoT cloud testbed, based on CoreOS. In our testbed we simulate and mimic physical gateways in the cloud. The gateways are based on a snapshot of a real-world gateway, developed by our industry partners. The testbed is deployed on our local OpenStack cloud and it consists of 7 CoreOS 444.4.0 VMs, each running 150 LXCs, thus simulating approximately 1000 vehicles.

6.2.1 Example 1 – Energy consumption tracking

The FMS needs to monitor high-value vehicles' energy consumption in (near) real-time. In case any energy fault is detected, it must notify a golf course manager and put the vehicles in a reduced energy mode.

Listing 8 Remote monitoring of fleet's energy consumption

```
//select high-value vehicles
IntentScope s =
cont.delimit(IntentScope.getGlobal(),
Cond.greaterThan("price", "5000"));
Intent eFault = Intent.newMIntent("EnergyFault");
eFault.setScope(s);
cont.notify(eFault, this);//sub. to event
...
public void onEvent(Event e){
IntentScope ts = IntentScope.create(e.getEntityId();
Intent eCons = Intent.newCIntent("ReduceEnergy");
eCons.setScope(ts);//set task scope
cont.send(eCons); //send to all vehicles in ts
}
```

The most important part of this application in shown in Listing 8. To implement the monitoring behavior, developers only need to define an IntentScope (lines 2–4), in which they declare properties (e.g., metadata) that need to be satisfied by monitored vehicles, define a MonitorIntent and assign the desired scope to it (lines 5–7). Similarly, to implement a remote control behavior developers only need to define a ControlIntent (lines 12–14). In this example, it is natural to use asynchronous communication (of sensory data), thus a developer uses SDG-Pro's `notify` directive (line 8), to subscribe for the state changes in the environment.

This example demonstrates how easy it is to implement a real-time remote monitoring behavior. By introducing IntentSopes at the application level, SDG-Pro shields the developers from directly referencing the vast number of diverse physical entities and enables them to delimit the range of their actions on a higher abstraction level. Similarly, to perform an IoT control action or to subscribe for relevant events, developers only need to define and configure the corresponding Intents. This allows them to communicate to the system what needs to be done, instead of worrying how the underlying devices will perform the specific task.

6.2.2 *Example 2 – Scheduled maintenance check*

The FMS performs daily checks of the fleet's health. This is done mainly during the night, when the vehicles reside dormant in the club house, within the Wi-Fi range. The application reads the diagnostic data, gathered during the day, and analyzes them offline.

Listing 9 Scheduled maintenance check

```
Intent localCon = Intent.newMIntent("ConnType");
 localCon.setScope(IntentScope.getGlobal());
 IntentScope ds = container.
 delimit(Cond.eq("WLAN", localCon));
 ds.addObserver(this);

public void update(Observable obs, Object arg){
 Intent di = Intent.newMIntent("DiagnosticsLogger");
 di.setScope((IntentScope)obs);
 List<Event> data = container.poll(diagnostics);
 // send data to an analytics framework
}
```

To implement such behavior, application first needs to determine that a vehicle is connected to a local network. This is achieved by defining an active IntentScope, as shown in Listing 9, lines 1–4. Second, the application needs to gather vehicles' diagnostic data and store them, e.g., in a local database. To synchronously poll the vehicle data, a developer simply defines a MonotorIntent and uses the `poll` directive (lines 8–10).

This example, demonstrates several important points. First, since MonitorIntents can be used to define an IntentScope, SDG-Pro enables developers to dynamically (e.g., based on environment or context changes) determine application behavior. Second, since IntentScopes are observable, developers can specify complex conditions that will trigger an execution of the business logic, without having to write complicated queries and event processing schemes.

Finally, it is worth noticing that SDG-Pro does not provide support for the data analytics. However, we have shown that with little effort, by using intuitive concepts, an offline analytics application can obtain the required data, which can then be analyzed with data analytics frameworks, e.g., MapReduce.

6.2.3 *Example 3 – Diagnostics data logging*

This application periodically pools the data from the variety of vehicle's sensors e.g., engine status, battery status, transmission, etc. and stores them locally for later analysis (e.g., see Example 2).

Listing 10 Logging diagnostics data locally

```
//Create custom sensor from physical channel
BufferConf bc = new BufferConf("voltage_{i}n");
bc.setClass(BufferClass.SENSOR);
bc.getAdapterChain().add(
 new ScalingAdapter(0.0,100.0,10.0));
 bc.getAdapterChain().add(new LowpassFilter(0.30));
 BufferManager.create("lowpass-scaled", bc);
//Define diagnostics model
DataPoint diagnostics =
new ComplexDataPoint("lowpass-scaled","voltage_{i}n");
 DataInstance di = diagnostics.read();
//log the diagnostics data di
```

Listing 10 shows a partial diagnostics data model. The diagnostic data contains raw engine voltage readings and scaled voltage readings with low-pass filter, e.g., possibly indicating that something is taking the power away from the motor. To develop a custom sensor, developers only need to create a virtual buffer (referencing the base channel, e.g., raw voltage readings) and configure its adapter chain, as shown in lines 2–6. After creating a custom virtual sensor (line 7) application can treat this sensor as any other sensor. Consequently, a data model can then be easily defined with Data Points, as sown in lines 9–11. Storing the data is omitted for readability purposes.

Essentially, this example shows how our framework transparently virtualizes access to the same voltage sensor. This demonstrates two important features of the data and control points. First, since the SDG-Pro provides (virtually) exclusive access to the sensors (i.e., buffers act as multiplexers), developers can define custom configurations for the data streams, effectively creating an application-specific view of the sensors. An important consequence is that multiple applications can easily share the infrastructure, retaining a custom view of it. Second, since Data and

Control points support developers to interact with underlying devices in a unified manner, i.e., independent of the communication protocols or the input channel types, applications can define their (arbitrarily complex) data models by only specifying the required data points. These can be seen as volatile fields in traditional data model entities.

6.2.4 Example 4 – Energy fault detection

To detect vehicles over consuming battery an FMS service relies on powermeter, odometer and temperature sensors that are available in the vehicles and uses a custom algorithm to detect potential energy faults.

Listing 11 Device service for energy fault detection

```
DataPoint dp1 = DataPoint.
 create( "battery", "odometer" );
 DataPoint dp2 = new DataPoint();
//Since we have multiple temperature sensors
//we query them via the meta data
Collection<BufferDescription> tempBuffers =
 dataPoint.queryBuffers(
 new SensorProps("*temperature*"));
 dp2.assign(tempBuffers);
...
 //invoke energy fault detection algorithm
```

In Listing 11 we show a code snippet from the corresponding FMS service. Developers create two data points. The `dp1` combines the battery status and odometer readings and it asynchronously delivers the sensory readings to the service. The `dp2` queries the available temperature data channels, based on their meta data and aggregates the temperature readings from the available thermometers (lines 6–9). Among other things, the energy fault detection algorithm uses these data points and Complex Event Processing (CEP) techniques to determine potential energy faults, but its implementation is omitted in accordance with our nondisclosure agreement.

We notice that application obtains the temperature readings without directly referencing any physical sensor. Instead it generically queries the sensors' meta data. Further, since SDG-Pro takes care of synchronizing the sensors' readings, e.g., among the temperature sensors, developers can focus on custom data processing steps (algorithm). This is a crucial requirement to be able to develop portable applications, which do not directly depend on the physical infrastructure.

6.2.5 Example 5 – Provisioning and deploying application runtime environment

In order to execute an application/service (see Examples 1–4), developers need to provision a software-defined gateway and deploy it atop IoT cloud.

Listing 12 Creating a software-defined gateway

```
/* Snippet from Provisioning.java*/
//install JVM Compact Profile 1
String dest = ".../G2021/jvm";
provisioner.CreateDirIfMissing(dest);
provisioner.CopyToDir("jvm-profile1-1.8.0/*",dest);
provisioner.setPermissions(dest, "a+x");
...
/* Snippet from Gateways.java*/
SDGateway gateway
 = UnitsController.create(GType.BUSYBOX);
 gateway.addConfigClass(Provisioning.class);
 UnitsController.startParallel(gateway,
 IntentScope.getGlobal().asResource());
```

Listing 12, shows how to programmatically add Java Compact Profile runtime to a gateway and how to deploy instances of that gateway atop the vehicles' on-board devices. In lines 3–6 we show how developers can use the provisioning API to specify which custom resources are required in the gateway prototype. Further, this example show the most important parts related to gateways deployment, i.e., gateway instantiation from Docker-based Busybox prototype (lines 9,10), associating the configuration model with the prototype (line 11) and multiple deployment (lines 12,13).

This example shows a part of general SDG-Pro's provisioning API. We notice that our framework provides a generic API which can be used to declaratively configure different types of resources. This essentially enables developers to programmatically deal with complex IoT cloud infrastructure and its dependencies, i.e., the desired configuration baseline is specified locally and once for multiple application instances. SDG-Pro provides a unified view on defining and manipulating the infrastructure through software-defined gateways, but also offers a fine-grained access and control of the gateways configuration (e.g., container's base image).

6.2.6 Example 6 – Configuring application dependencies programmatically

The FMS applications have different dependencies and requirements e.g., regarding communication protocols. To guarantee correct application behavior, developers (or operations managers) need to correctly configure such infrastructure dependencies.

Listing 13 Configuring application dependencies

```
//install Modbus
provisioner.addDCPointResource("modbus/Modbus.sab");
provisioner.addDCPointResource("modbus/Modbus.sax");
provisioner.addDCPointResource("modbus/Kits.scode");
provisioner.addDCPointResource("modbus/Kits.xml");
//install MQTT client
RemoteLibrary mqttClient
 = provisioner.getFromURL(
 "http://..../mqtt-client-0.0.1.jar");
provisioner.installComProto(mqttClient.getBinary());
...
```

Listing 13 shows excerpt of typical FMS protocols configuration. Lines 2–6 show how developers can to configure Modbus device protocol (used by Data and Control Points) and MQTT cloud connectivity protocol (lines 7–10), e.g., used by MonitorIntents.

The most important thing to notice here is that SDG-Pro provides software-defined gateway specific provisioning APIs. This shows that our abstractions are designed in such manner to inherently support programmatic provisioning, by exposing well-defined API and providing runtime mechanisms which transparently enable inversion of control and late (re)binding of the dependencies. Also standard provisioning operations such as fetching a remote resource can be combined with specific provisioning APIs, as shown in lines 7–9. The most important consequence is that developers can design generic application business logic and transparently declare the desired infrastructure dependencies programmatically, e.g., in a separate application module.

6.2.7 Example 7 – Emergency governance process

In case of an emergency situation the FMS needs to increase the monitoring frequency of vehicles' sensors.

Listing 14 Example emergency operational governance process

```
Iterator<Vehicle> vehicles.iterator();
//for each vehicle on the golf course
List<DataPoint> dPoints = HTTPClient
 .invoke(".../APIManager/mapper/"
 +vehicles.next().getId()+"/DPcapa/list");
 for (DataPoint dp : dPoints) {
 HTTPClient.invoke(".../DPcapa/"
 +"setPollRate/args?rate=10s&id="+dp.getId()");
}
```

To satisfy this cross-cutting compliance requirement, developers need to develop an operational governance process [9, 10]. Listing 14 shows a code snippet form such emergency governance process. The most important part of the process is shown in lines 7–8, which show how a developer can use governance API to dynamically manipulate the edge of the infrastructure, in this case change the sensor update rate.

SDG-Pro takes over the responsibility of invoking individual governance capabilities (e.g., per vehicle), effectively shielding the developers from low-level infrastructure details. The most important consequence of having such governance API is that the governance logic can be specified programmatically and maintained locally. Also governance processes are completely separated from the business logic, thus the core business logic is not polluted with cross-cutting governance concerns.

In addition, since at the application level the infrastructure is perceived as a set of capabilities exposed through the governance API, the developers do not have to worry about geographical distribution, heterogeneity or scale of the IoT cloud infrastructure nor directly deal with individual devices.

6.3 Discussion

As shown on a set of real-life examples, our SDG-Pro framework enables addressing most of development concerns at application code level (*everything as code*). This provides advantages such as having a uniform view on the entire development process, which makes it easily traceable and auditable, but also enables exploiting proven and well-known technologies, e.g., source control or configuration management systems, during the entire application lifecycle. Moreover, it gives full control to developers and makes IoT cloud applications less infrastructure-dependent.

We have shown how SDG-Pro provides *API encapsulation* of the most important aspects related to gateway provisioning and governance. A key advantage of this approach is that developers do not need to explicitly worry about the underlying infrastructure. Rather, they perceive the complex and heterogeneous IoT cloud infrastructure as a set of uniform APIs that enable programmatic management of such infrastructure. Our SDG-Pro framework supports the developers to *declaratively provision* IoT cloud systems and to *automate* most of the provisioning process. This improves general readability and maintainability of the provisioning logic and simplifies the provisioning process. Additionally, by encoding the provisioning directives as part of application's source code, our framework makes the provisioning process easily repeatable. This reduces the potential errors, but more importantly enables continuous, automated enforcement of the configuration base line.

Regarding the governance processes, by providing a logically *centralized point of operation* of IoT cloud infrastructure, SDG-Pro supports developers to easily define desired states and runtime behavior of IoT cloud systems, but also enables automated enforcement of governance processes, which is crucial to realize (time) consistent governance strategies across the entire IoT cloud system.

We also notice a number of limitations of our approach. From the technical perspective, at the moment SDG-Pro offers a rudimentary mechanism for gateway allocation, which only considers static properties when deploying the software-defined gateways. Additionally, although IoT cloud systems include many mobile and unstable devices, the current prototype provides a limited support regarding the dependability concerns. However, optimization of gateway allocation and addressing the dependability issues related to device mobility are subject of our future work.

Furthermore, the set of proposed programming concepts is not exhaustive and especially the provisioning and governance APIs are in an active state of development and refinement. However, as we have shown on a set of real-life examples, SDG-Pro offers programming support sufficient to express many common behaviors of IoT cloud applications. Although our programming model has many important traits such as readability and simplicity, as well as facilitates writing reusable and portable application logic, in SDG-Pro's programming model, we trade flexibility and expressiveness for more intuitive and efficient programming of the IoT cloud applications. Finally, although developers utilize the well-known Java programming language, SDG-Pro introduces a number of new concepts that require an initial learning effort. However, by explicitly enforcing main design principles of software-defined IoT cloud systems, we believe that in the long-run our framework can reduce development time, potential errors and eventually the costs of application development.

7 Related work

Developing and managing IoT cloud systems and applications have been receiving a lot of attention lately. In [1, 14, 15] the authors mostly deal with IoT infrastructure virtualization and its management on cloud platforms. A number of different approaches (e.g., [2, 16]) employ semantics aspects to enable discovering, linking and orchestrating heterogeneous IoT devices. In [3, 17] the authors propose utilizing cloud for additional computation resources and approaches presented in [4, 18] focus on utilizing cloud's storage resources for sensory data. Approaches presented in [5, 19] deal with integrating IoT devices and services with enterprise applications based on SOA paradigm.

In [14] the authors focus on developing a virtualized infrastructure to enable sensing and actuating as a service on the cloud. They propose a software stack that includes support for management of device identification and device services aggregation. Although, this approach facilitates development of IoT cloud applications to a certain extent, contrary to SDG-Pro it does not define a structured programming model for developing such applications. In [1] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors on the cloud and provides management and monitoring mechanisms for the virtual sensors. However, the support for sensor provisioning is based on static templates that, contrary to our approach, do not support dynamic provisioning of IoT cloud resources.

SenaaS [16] mostly focuses on providing a cloud semantic overlay atop physical infrastructure. It defines an IoT ontology to mediate interaction with heterogeneous devices and data formats, exposing them as event streams to the upper layer cloud services. Similarly, the OpenIoT framework [2] focuses on supporting IoT service composition by following cloud/utility based paradigm. It mainly relies on semantic web technologies and CoAP to enable web of things and linked sensory data. Such approaches address very important issues such as discovering, linking and orchestrating internet connected objects and IoT services, thus conceptually complementing our approach. Although, SDG-Pro relies on semantic concepts, e.g., it implements hierarchical namespaces and a proprietary taxonomy for sensor interoperability to support Data and Control Points, the semantic aspects are not the main focus this work. The aforementioned approaches mainly focus on providing different virtualization, management and (semantic-based) interoperability techniques for IoT devices. Therefore, such approaches can be seen as complementary to our own, as device virtualization sets the cornerstone for achieving IoT cloud systems. The SDG-Pro framework relies on the contemporary advances in IoT cloud and extends them with novel programming abstractions which enable everything-as-code paradigm, facilitating development of IoT cloud applications and making the entire development process traceable and auditable (e.g., with source control systems), thus improving maintainability and reducing development costs.

Putting more focus on the edge devices, i.e., IoT gateways, network devices, cloudlets and small clouds, different approaches have emerged recently. For example, in [20] the authors present a concept of fog computing and define its main characteristics, such as location awareness, reduced latency and general QoS improvements. They focus on defining a virtualized platform that includes the edge devices and enables running custom application logic atop different resources throughout the network. Similarly, Cloudlets [6] and small clouds [21] are introduced as intermediary infrastructure nodes (between the edge devices and data centers), which can be used to reduce network delays, processing time and costs. The SDG-Pro framework also aims at better utilization of the edge infrastructure, but we focus on providing a systematic approach, supporting application developers to address most of the infrastructure provisioning and governance issues programmatically, in a logically centralized fashion, by offering the software-defined gateways and well-defined provisioning and governance APIs.

Different approaches have exploited and extend software defined concepts to facilitate utilization and management of the pooled sets of shared IoT cloud resources, e.g., software-defined storage [22] and software-defined data center [23]. Also recent advances in more traditional software-defined networking (SDN) [24] have enabled easier management and programming of the intermediate network resources, e.g., routers. However, SDN mostly focuses on defining the networking logic, e.g., injecting routing rules into network elements. Conversely, our SDG-Pro addresses the more general problem of providing programming support for a general business logic of IoT cloud applications. It builds on our previous concepts to provide programming abstractions, which enforce earlier identified design principles of software-defined IoT cloud, in order to enable scalable, efficient and more intuitive application development.

Another related field is macroprogramming of sensor networks [25–28]. For example, in [25] the authors provide an SQL-like interface where the entire network is abstracted as a relational database (table). Contrary to their approach, we utilize more general set theory to define operations on our IntentScopes. This gives more flexibility to developers, since SDG-Pro also allows dynamic, custom properties to be included in scope definitions, but comes at the cost of additional performance overhead.

Similarly, in [27], the authors deal with enabling dynamic scopes in WSN, mainly addressing the important issues of task placement and data exchange (among the WSN nodes), in order to account for the heterogeneity of the nodes and enable logically localized interactions. Their approach can be seen as conceptually complementing the SDG-Pro, since task allocation and such interaction types are not the main focus of our framework. In [26], the authors propose the notion of logical neighborhood. Their approach is based on logical nodes (templates), which enable instantiating and grouping the nodes, based on their exported attributes. To facilitate communication within the neighborhoods, which is of greater importance in WSN, they also provide an efficient routing mechanism. In [28] the authors introduce an extensible programming framework that unifies the WSN programming abstractions in order to facilitate business processes orchestration with WSN. Despite the relevant efforts to integrate provisioning and business logic (e.g., template-based customizations [26]), the main focus of the aforementioned approaches is application business logic, while we address a more general problem of enabling everything-as-code paradigm, in order to also allow for capturing provisioning and governance logic for IoT cloud resources programmatically.

8 Conclusion and future work

In this paper we introduced the SDG-Pro framework for software-defined IoT cloud systems. We presented SDG-Pro's programming model for IoT cloud applications, which is designed to enforce the main principles of software-defined IoT cloud systems that were elicited in our previous research in this area [7, 9, 10]. By enforcing such principles on the application level, our framework enables easier, efficient and more intuitive application development. It provides a unified programmatic view on the entire development process (*everything as code*) making it easily traceable and auditable, thus reducing development time, errors and costs of application development.

In the future, we will continue the development of the SDG-Pro framework to address its current limitations, i.e., improve the gateways allocation mechanism to include support for dynamic infrastructure properties. We also plan to address the current limitations regarding the mobility aspects, especially the dependability issues related to the device mobility and mobility of software components, i.e., runtime migration of software-defined gateways. Finally, we plan to extend the current prototype to support elastic, on-demand scaling of the software-defined gateways.

Endnotes

[1]The provisioning agent is implemented as a light-weight service, based on Oracle Compact Profile1 JVM.

[2]https://docker.com/.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
SN is the main contributor of this work, which was undertaken as part of his Ph.D. studies. SN drafted most of the manuscript and carried out most of the framework design and development. HLT is a co-supervisor of SN, with contributions to conceptualizing software-defined gateways and programming in IoT cloud systems. SD is the supervisor of SN, with main contributions to conceptualization of software-defined IoT cloud systems. All authors read and approved the final manuscript.

Acknowledgements
This work is partially supported by the EU H2020 U-Test project, under grant No. 645463.

Received: 30 January 2015 Accepted: 1 October 2015
Published online: 30 October 2015

References
1. Yuriyama M, Kushida T (2010) Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing. In: NBiS'10. pp 1–8. doi:10.1109/NBiS.2010.32
2. Soldatos J, Serrano M, Hauswirth M (2012) Convergence of utility computing with the internet-of-things. In: IMIS. pp 874–9. doi:10.1109/IMIS.2012.135
3. Chun BG, Ihm S, Maniatis P, Naik M, Patti A (2011) Clonecloud: elastic execution between mobile device and cloud. In: Proceedings of the Sixth Conference on Computer Systems. ACM, New York, NY, USA. pp 301–314. http://doi.acm.org/10.1145/1966445.1966473
4. Stuedi P, Mohomed I, Terry D (2010) Wherestore: Location-based data storage for mobile devices interacting with the cloud. In: Proceedings of the 1st ACM Workshop on Mobile Cloud Computing &#38; Services: Social Networks and Beyond. ACM, New York, NY, USA. pp 1:1–1:8. http://doi.acm.org/10.1145/1810931.1810932
5. De Souza LMS, Spiess P, Guinard D, Köhler M, Karnouskos S, Savio D (2008) Socrades: A web service based shop floor integration infrastructure. In: The Internet of Things. pp 50–67. http://link.springer.com/chapter/10.1007%2F978-3-540-78731-0_4
6. Satyanarayanan M, Bahl P, Caceres R, Davies N (2009) The case for vm-based cloudlets in mobile computing. Pervasive Comput 8(4):14–23. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5280678

7. Nastic S, Sehic S, Le DH, Truong HL, Dustdar S (2014) Provisioning Software-defined IoT Cloud Systems. In: FiCloud'14. IEEE, Barcelona, Spain. pp 288–295. doi:10.1109/FiCloud.2014.52
8. Nastic S, Sehic S, Voegler M, Truong HL, Dustdar S (2013) PatRICIA - A novel programing model for IoT applications on cloud platforms. In: SOCA. IEEE Computer Society, Koloa, HI, USA. doi:10.1109/SOCA.2013.48
9. Nastic S, Voegler M, Inziger C, Truong HL, Dustdar S (2015) rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems. In: Mobile Cloud 2015. IEEE, San Francisco, CA, USA. pp 24–33. doi:10.1109/MobileCloud.2015.38
10. Nastic S, Inziger C, Truong HL, Dustdar S (2014) GovOps: The Missing Link for Governance in Software-defined IoT Cloud Systems. In: WESOA14. Springer, Paris, France Vol. 8954. pp 20–31. doi:10.1007/978-3-319-22885-3_3
11. Inzinger C, Nastic S, Sehic S, Vögler M, Li F, Dustdar S (2014) MADCAT - A methodology for architecture and deployment of cloud application topologies. In: Service-Oriented System Engineering. IEEE Computer Society, Oxford, United Kingdom. pp 13–22. doi:10.1109/SOSE.2014.9
12. Mohagheghi P, Haugen Ø (2010) Evaluating domain-specific modelling solutions. In: Advances in Conceptual Modeling - Applications and Challenges. pp 212–21. http://link.springer.com/chapter/10.1007%2F978-3-642-16385-2_27
13. Voegler M, Schleicher JM, Inziger C, Nastic S, Sehic S, Dustdar S (2015) LEONORE – Large-scale provisioning of resource constrained IoT deployments. In: SOSE. IEEE, San Francisco Bay, CA, USA. pp 78–87. doi:10.1109/SOSE.2015.23
14. Distefano S, Merlino G, Puliafito A (2012) Sensing and actuation as a service: a new development for clouds. In: NCA. pp 272–275. doi:10.1109/NCA.2012.38
15. Hassan MM, Song B, Huh EN (2009) A framework of sensor-cloud integration opportunities and challenges. In: ICUIMC. ACM, New York, NY, USA. pp 618–626. doi:10.1145/1516241.1516350
16. Alam S, Chowdhury M, Noll J (2010) Senaas: An event-driven sensor virtualization approach for internet of things cloud. In: NESEA. pp 1–6. doi:10.1109/NESEA.2010.5678060
17. Kumar K, Lu YH (2010) Cloud computing for mobile users: Can offloading computation save energy? Computer 43(4):51–6
18. Zaslavsky A, Perera C, Georgakopoulos D (2013) Sensing as a service and big data. arXiv preprint arXiv:1301.0159. http://arxiv.org/abs/1301.0159
19. Kovatsch M, Lanter M, Duquennoy S (2012) Actinium: A restful runtime container for scriptable internet of things applications. In: Internet of Things. pp 135–142. doi:10.1109/IOT.2012.6402315
20. Bonomi F, Milito R, Zhu J, Addepalli S (2012) Fog computing and its role in the Internet of Things. In: MCC Workshop on Mobile Cloud Computing. ACM, New York, NY, USA. pp 13–16. http://doi.acm.org/10.1145/2342509.2342513
21. Sixsq NuvlaBox. http://sixsq.com/products/nuvlabox.html. [Online; accessed Jan-'15]
22. Thereska E, Ballani H, O'Shea G, Karagiannis T, Rowstron A, Talpey T, et al. (2013) IoTFlow: A software-defined storage architecture. In: SOSP. ACM, Farmington, PA, USA. pp 182–96. http://doi.acm.org/10.1145/2517349.2522723
23. Davidson, Emily A (Softchoice Advisor): The Software-Defined-Data-Center (SDDC): Concept Or Reality? http://tinyurl.com/omhmbfv. [Online; accessed Jan-'15]
24. Koldehofe B, Dürr F, Tariq MA, Rothermel K (2012) The power of software-defined networking: line-rate content-based routing using OpenFlow. In: Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing. ACM, New York, NY, USA. pp 3:1–3:6. http://doi.acm.org/10.1145/2405178.2405181
25. Madden SR, Franklin MJ, Hellerstein JM, Hong W (2005) TinyDB: an acquisitional query processing system for sensor networks. ACM Trans Database Syst (TODS) 30(1):122–73
26. Ciciriello P, Mottola L, Picco GP (2006) Building virtual sensors and actuators over logical neighborhoods. In: Proceedings of the International Workshop on Middleware for Sensor Networks. ACM, New York, NY, USA. pp 19–24. http://doi.acm.org/10.1145/1176866.1176870
27. Mottola L, Pathak A, Bakshi A, Prasanna VK, Picco GP (2007) Enabling scope-based interactions in sensor network macroprogramming. In: MASS 2007. IEEE Computer Society, Pisa, Italy. pp 1–9. http://dx.doi.org/10.1109/MOBHOC.2007.4428655
28. Casati F, Daniel F, Dantchev G, Eriksson J, Finne N, Karnouskos S, et al. (2012) Towards business processes orchestrating the physical enterprise with wireless sensor networks. In: ICSE'12. IEEE, Zurich, Switzerland. pp 1357–1360. doi:10.1109/ICSE.2012.6227080