FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Contract Definition and Governance for IoT

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Peter Klein, BSc
Registration Number 8251105

to the Faculty of Informatics
at the TU Wien
Advisor: Privatdoz. Dr.techn. Hong-Linh Truong

Vienna, 23rd January, 2018

Peter Klein	Hong-Linh Truong

Erklärung zur Verfassung der Arbeit

Peter Klein, BSc
Kuhngasse 8, Haus 5, 2201 Gerasdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Jänner 2018

Peter Klein

Acknowledgements

First of all I would like to express my sincere appreciation and gratitude to my advisor, Mr. Prof. Hong-Linh Truong. He always gave me very valuable feedback, guidance, support and I really enjoyed working on this thesis.

I would also like to thank my family, my wife Gabriele, my sons Martin and Markus, and my friends and colleagues for their support, patience and encouragement.

Abstract

In large scale deployment of the Internet of Things a high number of devices such as sports wearables, home heating and air conditioning systems, manufacturing machines and in-vehicle components are connected to each other and to cloud applications providing complex services for users. IoT units provide a software based abstraction of services, sensors and actors of devices. Contractual relationship of the Stakeholders IoT unit provider, IoT service provider and IoT service consumer have to be captured in a well-defined, machine processable and automatically enforceable manner. A challenge is the dynamic runtime environment where unit providers run IoT units for different services on the same shared platform, and services require input from several units of different providers to be able to fulfill their requirements.

In this thesis an extensible framework is described that covers contract creation with a flexible model based on JSON (Java Script Object Notation), monitoring of contract term related constraints on IoT units via Aspect-oriented programming as well as contract enforcement by storing and retrieval of contract violations and linking them as hash values to smart contracts on the Ethereum blockchain.

A prototype is implemented and evaluated with respect to solving use cases based on real world scenarios and with respect to handle workloads defined in performance testing.

The results gained from evaluation demonstrate that the introduced framework for IoT contract definition and governance is able to cover real world scenarios and provides performance and scalability to handle the workloads related to the scenarios.

Kurzfassung

Im Internet der Dinge wird eine große Zahl unterschiedlicher Geräte wie tragbare Sportgeräte, Heizungs- und Klimaanlagensteuerungen, Produktionsmaschinen oder Fahrzeugkomponenten miteinander und mit der Cloud verbunden, so dass komplexe Dienste für die Nutzer zur Verfügung gestellt werden können. IoT Einheiten stellen eine Abstraktion der Sensoren, Aktoren und Dienste von IoT Geräten dar. Dabei sollen Vertragsbedingungen zwischen den unterschiedlichen Partnern, wie Anbietern von IoT Einheiten und IoT Diensten sowie IoT Dienstnutzern, erfasst und in exakter, maschinenlesbarer und automatisch verfolgbarer Art und Weise umgesetzt werden können. Eine Herausforderung dabei ist eine dynamische Laufzeitumgebung in der IoT Geräte für unterschiedliche Dienste auf einer gemeinsam genutzten Plattform laufen und Dienste auf Geräte unterschiedlicher Art zugreifen müssen um ihre Funktion erfüllen zu können.

In der Diplomarbeit wird ein erweiterbares Framework beschrieben das den Bereich des Vertragsentwurfs über ein flexibles Modell auf Basis von JSON (JavaScript Object Notation) abbildet. Die Überwachung der mit den Vertragsbestandteilen verbundenen Einschränkungen erfolgt über Aspekt-orientierte Programmierung. Die Vertragserfüllung wird durch eine Komponente zur Speicherung und Abfrage von Vertragsverletzungen sowie deren Verlinkung mit Smart Contracts in der Ethereum Blockchain unterstützt.

Es wird ein Prototyp implementiert und in Hinblick auf die Realisierbarkeit echter Szenarien und der damit verbundenen Rechnerbelastung evaluiert.

Die Ergebnisse der Evaluierung zeigen dass das neu entwickelte Framework für IoT Contract Definition und Governance in der Lage ist die Szenarien abzubilden und die notwendige Performanz und Skalierbarkeit gewährleistet werden kann.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Introduction

The Internet of Things (IoT) refers to the growing range of Internet-connected devices that capture or generate an enormous amount of information every day. For consumers, these devices include mobile phones, sports wearables, home heating and air conditioning systems, and more. In an industrial setting, these devices and sensors can be found in manufacturing equipment, the supply chain, and in-vehicle components [1], [2], [3], [4]. Many existing IoT applications have been built in a silo approach for a single purpose and use a strong coupling between devices, frameworks and application logic. Contracts for these applications work like contracts in traditional software applications being defined in a software end user license agreement (EULA) that is created by legal people and intended to be read and accepted by customers. Examples of such license agreements are given in [5] and [6]. Monitoring of Quality of Service (QoS) or Quality of Data (QoD) related contract terms is done by application specific means (e.g. analysis of log files provided or measurements taken on client side). Disputes are settled in a manual way by a legal suit and legal decision [7], [8]. In state of the art designs, applications are composed of independent IoT units instead of the strong coupling [9] and deployed in cloud data centers and edge gateways [10]. IoT units provide a software based abstraction of services, sensors and actors of IoT devices [11]. Services by one unit are consumed by other units to provide aggregated services that are then again consumed by applications thus building a complex web of provider and consumer relationships which makes contracts hard to define and enforce in the existing approach.

1.2 Problem Statement

In scenarios such as the Santander Smart City project [12] or the Array of Things [13] a large number of different sensors are deployed in a city and services are provided based

on the sensor data to inhabitants and the government of the city. Similar cases are large scale environment monitoring, vehicle monitoring or health tracking [3]. IoT services are composed of IoT units. IoT contracts are formally defined, machine readable and executable definitions for IoT services that have to cover access rights (e.g. giving certain IoT units access to read sensor data and use actors), communication rights (amount of data to be allowed in transmission, allowance to send data to certain receivers) as well as resource utilization rights (CPU usage, frequency of sensor readings) and service level agreements (response time, throughput, payment) [14].

The thesis aims to answer following questions:

- What is a usable modeling approach for IoT contracts that is flexible enough to cover different kinds of services and is able to be interpreted and processed by software running on IoT units themselves thus providing a machine readable and executable contract.
- How can the monitoring requirements be derived from the contract and monitoring functions being instantiated on the IoT units so that data for evaluation of contract terms is collected and made available for enforcement.
- How can enforcement of contract terms be provided in an automatized way so that contract violations are identified and information about them is provided to contract parties in a trusted manner.

The goal of this thesis is to develop a framework that supports IoT contract definition, monitoring and enforcement. Contracts will be defined as objects according to a contract model and assigned to IoT units. Monitoring of IoT units is derived from contracts and provides information to contract enforcement. Figure 1.1 gives an overview of the framework and its embedding into IoT units and services:

- The *Contract Manager* is responsible to build contracts from contract models and assign them to IoT units.
- The *Contract Models* contains building blocks for creation of contracts.
- The *Contract Governance* is responsible for recording of contract violations identified by monitoring and enforcement and making them available to contract parties.

The framework will provide the base to ensure that IoT units and services can fulfill their contracts as well as work together deployed on an IoT edge gateway or cloud in a coordinated manner with proper resources available for each service.

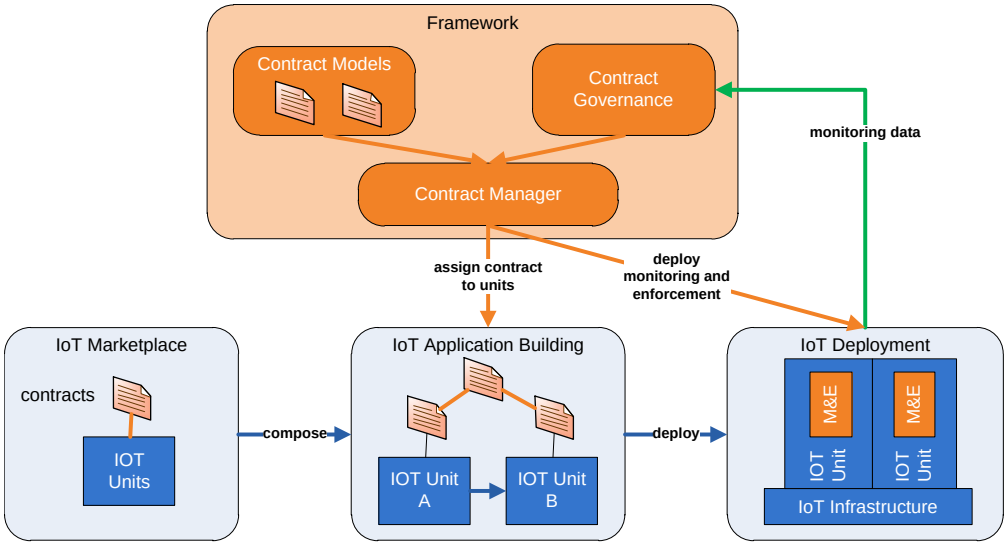

Figure 1.1: High Level Overview of IoT Contract Framework

1.3 Methodological Approach

It is divided into three main parts: analysis, design and development and evaluation.

- In the *Analysis* part literature research on relevant topics for IoT contract definition and governance is conducted, leading to selection of scenarios and definition of requirements based on the scenarios.
- In the *Design and Development* part, a framework is designed and a prototype is implemented, based on the previously defined requirements. The prototype serves as implementation to solve the problems defined above.
- In the *Evaluation* part the prototype is evaluated with respect to functionality provided to solve use cases based on real world scenarios (management of Heating, Ventilation, Air Conditioning system in transceiver stations for mobile communications) and with respect to handle workloads defined in performance testing.

1.4 Thesis Contribution

In this thesis a framework for IoT contract definition and governance including monitoring of constraints imposed by the contract and enforcement of contract violations by linking them to the blockchain is introduced. Contracts are built from contract templates which are composed of a set of contract terms. Each contract term includes a set of constraints where for each constraint the parameters and a piece of Javascript code for monitoring and enforcement is defined. Contract terms are defined to cover access rights, quality of service and data such as throughput, completeness or accuracy as well as payment and

pricing conditions.

The prototype implementation of the framework is available as open source software on https://github.com/rdsea/IoTContract.

1.5 Thesis Structure

The rest of this thesis is structured as follows:

- Chapter 2.1 presents relevant background topics and the state of the art regarding the Internet of Things (IoT), contracts, governance, enforcement and smart contracts.
- In Chapter 3 scenarios and requirements derived from the scenarios are presented.
- In Chapter 4 the contract specification and composition part of the new framework is described.
- In Chapter 5 the governance part of the new framework is described.
- In Chapter 6 the design and evaluation of a prototype implementation is presented.
- Finally the findings of this thesis are summarized and future work is outlined in Chapter 7.

CHAPTER 2

State of the Art and Background

2.1 Background

In this section we discuss further key elements related to IoT contracts and governance.

2.1.1 Internet of Things

The Internet of Things connects physical devices (e.g. machines, vehicles, home appliances) that are equipped with sensors, actuators, processing capability and network connectivity to applications running in the cloud, on gateways or routers. Each device is uniquely identifiable and is able to communicate on top of the existing Internet infrastructure. Device and applications together build cyber-physical systems such as smart grid, smart homes, intelligent transportation and smart cities.

IoT Architecture

A traditional way to build IoT applications is to connect IoT devices running on embedded platforms to applications performing data storage and analytics in the cloud via standard IP communications such as web services. In [15] an overview and a critique on this approach is presented. Main concerns seen are privacy and security, scalability, latency, bandwidth and QoS (Quality of Service) guarantees. A Global Data Plane that stores time series data in a location independent way together with the publish / subscribe pattern to access the data is proposed as a solution to overcome the problems described for traditional cloud based IoT applications. The idea to move processing from the cloud to processing elements close to the IoT devices or onto the devices themselves is further developed in Edge Centric Computing where computing, data and services are moved from central nodes to the edge of the network running on gateways, routers or powerful end devices [16]. Satyanarayanan, et al. in [17] introduce the concept of 'Cloudlets' that augment resource poor IoT device elements with cloud based, resource rich capabilities

on the edge level. Autonomous decision making, storage of personal and sensitive data as well as coordination and management of applications is performed on these edge computing elements.

IoT Applications

In this section we look at IoT applications, especially in the smart city domain where many different IoT services are provided and open access for the city's inhabitants and government is required. A very important is that access has to be controlled in a way that lets sensor owners decide which parts of their data they will provide and which conditions have to be guaranteed. In [3] an overview of context aware IoT solutions is given. According to the survey Context-Awareness is defined as *"the ability of a system to provide relevant information or services to users using context information (such as location, time or environment data) where relevance depends on the user's task"*. In the survey solutions are grouped into the categories smart wearable, smart home, smart city, smart environment and smart enterprise. Context-Aware features are categorized into presentation (showing relevant information to the user), execution (automatic processes depending on the context) and tagging (fusing data collected by multiple sensors together). A set of 58 solutions is grouped into the categories and analyzed with respect to the context-aware features. Among the lessons learned topics from the survey especially interoperability of products and services as well as resource management, large scale deployment, privacy and data analytics have to be considered for IoT contracts and governance.

The Array of Things (AoT) is an urban scale approach to deploy a large amount of identical sensors in an urban environment to provide the foundation for smart city applications [13]. Focus is on measurement, edge computing and reliability, scalability and replication of hardware devices. Measurements comprise meteorological data (e.g. temperature, humidity or light), chemical data (e.g. carbon monoxide, ozone) and environment data (e.g. camera, sound level). The set of sensors can be extended to include additional guest sensors. Edge computing (processing data already on the device) is applied to reduce bandwidth and resolve privacy issues for camera and sound sensors. Privacy is addressed by running the camera and sound sensors either in common mode (where predefined operations are applied locally and only extracted features like the number of people seen by the camera are transmitted) or sampling mode (where full images or sound data are transmitted to a secure storage but access is only granted through a confidentiality agreement) [13].

The SmartSantander smart city testbed in the city of Santander, Spain is presented in [12]. In the testbed wireless sensor networks are deployed together with NFC tags and smart-phone applications using them as sensors but also as interaction point with the users. A common information model is applied throughout the SmartSantander platform. It includes a resource model (static data), an observation model (handling measurements) and a taxonomy supporting physical parameters and units of measurement. Common infrastructure monitoring provides data quality monitoring (detecting non communicating

sensors, stuck sensor and outliers) and device status monitoring generating alarms when a device is not behaving properly.

2.1.2 Service Composition

In IoT applications the services provided by several IoT units are combined together to provide composite services. Linked to composition of services also a composition of IoT contracts is required combined with composition of QoS and QoD. In this section we will look into composition of services and quality measurements.

Valenzuela et al. describe a SOA (Service Oriented Architecture) based data fusion mechanism for the Internet of Things [18]. In the DOHA (Dynamic Open Home-Automation) platform services are self-contained components that receive and deliver data through well-defined interfaces. Operations performed by services are either simple ones, that work without interaction with other services, or composite ones, that require interaction with other services. Service interactions are modeled by a service composition map. As an example, the temperatures service managing the room temperature makes use of the temperature sensor service and the temperature regulator service.

In [19] service orchestration is modeled as a graph consisting of task nodes that perform some function and gateway nodes that are routing the orchestration logic. XOR nodes represent conditional branching, AND nodes represent parallel forking and OR nodes represent multiple choice. Depending on the orchestration graph, composition functions for QoS parameters can be defined, e.g. for latency as QoS parameter, the maximum of the paths in the AND block has to be taken. A Multi-Layer QoS Model for Service-Oriented Systems is presented in [20]. QoS parameters are grouped into performance, dependability, security and trust, and cost and payment. Parameters are aggregated depending on the composition patterns of sequence, loop, AND and XOR. Depending on the pattern aggregation formulas for parameters such as throughput, response time or availability are defined. An extensive overview of data quality parameters in streaming environments and rules for combining them when several data streams are merged is given by Klein et al. in [21].

2.1.3 Monitoring and Enforcement

Task of monitoring is to observe parameters of the IoT unit at run-time that are relevant for checking constraints attached to contract terms. Monitoring should work without code modification of the monitored IoT units and should have as little impact as possible on performance and run-time behavior of the IoT unit. Enforcement deals with execution of appropriate actions if a contract violation is detected by comparing the constraints defined in the contract with the measurement taken by monitoring. Such an action could be simply to report the contract violation in a secure way, but also more complicated actions such as to reconfigure the IoT application to meet quality of service constraints or to automatically initiate a payment if the conditions defined in the contract are met.

In [22] a comparison of instrumentation techniques for monitoring of Java based programs is given. Low level byte code instrumentation provides high flexibility but is difficult and error-prone to apply. AOP (Aspect-oriented Programming) allows to add monitoring on Java source code level. Details on AOP and its implementation AspectJ are given in [23] and [24]. Crosscutting concerns such as monitoring, logging or access control are defined in separate code parts and woven into the original code at defined join points by the AspectJ compiler. Join point definitions allow to select methods (e.g. all methods where the method name is messageHandler that have 2 arguments) and attach code parts (called advices) to them, that are executed before, instead of, or after the original method.

In [25] a framework for management of sensing resources in IoT cloud systems is presented. Resources (sensors, gateways, analyzers) are described by their properties (states and meta-data) and capabilities (actions that can be performed on them). A unified model of sensing resources and cloud services holds properties and capabilities. Management services such as deployment, elasticity control or IoT governance then manage cloud resources as well as start and stop of services or reconfiguration of sensors.

2.1.4 Smart Contracts

The term was "Smart Contract" was introduced by Szabo [26] and is defined as "*a computerized transaction protocol that executes the terms of a contract*". As an example let's look at a scenario where usage of a car will be shared and offered to others. In the contract it is offered by the seller that the car is available between 8 pm and 4 pm for usage by anyone who agrees to pay a rate of X coins / hour as fee + Y coins per kilometer traveled. As soon as the contract is agreed by a buyer, access to the car is granted. When the car is returned the proper fee is calculated and automatically transferred from the buyer's account to the seller's account. Smart Contracts rely heavily on capabilities of the blockchain to perform secure transactions between untrusted parties without need for a trusted intermediary. A blockchain constitutes a distributed ledger holding transactions in a way that all participants have full access to the ledger and will provably reach consensus about the transactions and their state. Each transaction is encapsulated in a block, cryptographically secured and connected to its predecessor block. The longest chain of blocks is considered as consensus among all parties. When a transaction is embedded in a block that is already linked to several other blocks in the chain, it would require rewriting all following blocks to invalidate the transaction. Since creating a new block is difficult (depending on the consensus algorithm it is defined what is "difficulty") it would require ownership of the majority of the blockchain network resources to create a long chain of fake blocks in this case. Therefore transactions in the blockchain can be considered as immutable and can be inspected by all participants. Messages sent to a smart contract that change the state of the contract (in our example signing the contract and paying the final fee) are handled as transactions in the blockchain. Since

smart contracts themselves are distributed on the blockchain and creation of a contract is a transaction they are also immutable and can be inspected by every participant.

In [27] an overview of blockchain, consensus algorithms and smart contracts is given. Applications of smart contracts to IoT cover firmware update of devices, IoT devices that sell their own data and services via micro transactions, and application to the supply chain where moving of goods is tracked among several parties. Deployment considerations for smart contracts include lower throughput and higher latency compared to a central solution, privacy issues since all transactions are visible on the network, and legal enforceability which is still limited.

Usage of blockchain in IoT is presented in [28] with focus on a blockchain based smart home platform where local immutable ledgers are connected via a blockchain overlay network to cloud storage. In [29] experiments to use the IBM Adept blockchain and Multichain for distribution of code and the storing of configuration data are performed.

2.2 State of the Art

In this section we discuss related work on contract definition and contract enforcement from the IoT domain and related domains such as data markets, cloud systems and security.

2.2.1 Execution Policy Framework for IoT Services in the Edge

P4SINC (Policy for Servicing IoT, Network Functions, and Clouds) is an execution policy framework for IoT SDM's(Software Defined Machines) [30]. An SDM provides an abstract representation of IoT devices and edge components. Policy templates define allowed data amount, allowed number of accesses, a white-list of allowed users, execution lifetime and SDM capabilities. Policy instances then assign concrete values to a template, e.g. limit the data read amount to 100 KB per day. Policy instances are then used by the policy enforcement to instrument code of SDM's which supervises and enforces policies at run time.

Compared to our work the data model is focused on policy definitions (such as data amounts, white lists or lifetime) and does not allow to define arbitrary contract terms. Monitoring and enforcement is based on program instrumentation via Aspect-oriented programming. This conceptual approach we reuse for monitoring and enforcement in our framework but allow to execute arbitrary enforcement logic via Javascript.

2.2.2 Enforcement of Security Policy Rules for the Internet of Things

In [31] a model for the definition of security policies for MQTT [32] is presented. It includes entities such as data, time, identity, role, behavior, trust and risk, rule templates and rules. Rule templates follow the event-condition-action paradigm and contain variables that get concrete values assigned from the system configuration when the policy is

applied. As an example a policy could be defined that allows access only if the requester of the access is within a certain range of the IoT device (e.g. 100 meters). Actions defined in policies can also log messages or notify users. The policy engine is directly integrated with the Mosquitto MQTT broker so that enforcement is performed in the broker and not in the IoT device.

The model used is focused on security policies and does not allow to the define other contract terms such as data quality or service availability. In our work we reuse the concept of rules (logic) where variables get assigned concrete values when the policy is applied.

2.2.3 Policy-Driven Security Management for Fog Computing

Dzousa et al. in [33] present a framework for security management in fog computing devices. The framework contains a policy repository holding policy rules, a policy decision engine connecting to a policy resolver and a policy enforcement point in the IoT device. As use case a smart transportation system is described where connected vehicles and smart traffic lights communicate with each other, e.g. to inform a connected vehicle about an approaching emergency vehicle so that it can give way. The data schema defines a set of attributes associated with a physical or virtual component. The policy schema defines a set of conditions associated with a requested action (e.g. to read data) that are required to be fulfilled for the request.

Our approach follows a similar architectural pattern with a contract repository and a governance controller but extends it to general contract terms and their enforcement instead of focusing on security policies.

2.2.4 IoT Access Control Issues: a Capability Based Approach

In [34] a capability based approach to access control is presented. A service user has to present a capability (in form of a token) to the service provider to get access to an object. The concept supports the principle of least authority (only the minimum required set of permissions is granted) and fine granular access control. In an example presented a car owner defines access capabilities to information about the car (such as location or engine status) to his wife (location), the city traffic management service (location) and the car maintenance company (engine status). The car control unit then uses the capabilities defined to grant or deny access by users.

A similar approach can be taken to manage access rights for IoT devices in our IoT contract and governance framework.

2.2.5 Portable Architecture for QoS Monitoring in the Cloud

Adinolfi et al. in [35] describe the QoS-MONaaS (Quality of Service - MONitoring as a Service) architecture for monitoring of QoS in IoT and cloud applications. It implements a monitoring functionality that is available to all applications running on top of the underlying cloud platform. Terms defined in the service level agreement are monitored on the platform. If a breach is detected a violation is notified and a violation record is created. As an experiment the architecture was applied to a smart metering application where measurements of smart meters are taken by an energy supplier and sent to the energy consumer.

The overall architecture resembles our work but focuses on QoS and does not handle access control or payment. Monitoring is performed on the platform level whereas in our framework monitoring is performed directly on the end devices or gateways.

2.2.6 Contract and Rights Management Framework Design

Guth, et al. in [36] describe a contract and rights management framework using standard XML based modeling. Contracts in the educational domain (support of exchange of learning resources) are mapped to the domain independent XML based ODRL language (Open Digital Rights Language) [37]. In ODRL a contract agreement holds assets (the items being handled by the contract), parties (the contract partners) and permissions (constraints and requirements). The framework supports offer creation, interpretation and access control. First the provider places an offer on the framework that is stored in a repository. In the next step the consumer checks available offers, selects the appropriate one and agrees on a contract with the provider. Finally the contract is interpreted so that the consumer can access the required resource.

Compared to our work the framework and ODRL focus on description of assets and the rights associated with these assets (such as playing or copying) but lacks constructs to define enforcement and monitoring terms. What we take over to our work is the concept that ODRL defines a meta-model (for parties, assets, policies, permissions) but does not define specific policies or assets.

2.2.7 Contract-aware IoT Dataspace Services

In [38] a platform for enabling contract-aware IoT dataspace services is presented. An IoT dataspace makes data from multiple IoT devices (Things) and from multiple providers available for data consumers. IoT devices send data to the IoT dataspace, data packages can then be downloaded by customers governed by data contracts assigned to the data packages. Terms in data contracts include data rights, quality of data (QoD), quality of service (QoS), pricing model, purchasing policy and control relationship. The platform supports contract negotiation where contract partners agree on contract terms. Data contract monitoring distributes incoming data according to the agreed contracts and

measures QoD (e.g. completeness) and QoS (e.g. availability) parameters.

The platform focuses on data contracts including monitoring of quality of data and quality of service. It includes advanced features such as contract negotiation and contract recommendation. Compared to our work the contract model as well as monitoring and enforcement is focused on data and does not allow to define other terms, e.g. access control of sensors and actors for IoT units.

2.2.8 Data contracts for cloud-based data marketplaces

They are presented by Truong, et al. [14]. An abstract model for data contracts is developed that can be used to build different types of data contracts for specific types of data. Main terms in such data contracts are data rights (what the consumer is allowed to do with the data, e.g. reproduction or derivation), quality of data, regulatory compliance (privacy and confidentiality), a pricing model and control relationship (such as limitation of liability and auditing of contractual compliance). An abstract model consisting of term categories, term names, term values and term units is proposed to represent the terms described above. Based on the model, applications can be built like data contract compatibility evaluation if several data sets and their associated contracts are combined.

The concept of generic contract modeling is also applied in our work. Evaluation of contract compatibility is not considered in our work yet but would be useful as a future extension.

2.2.9 Governance Platform for Cloud Service Delivery

A governance platform for cloud service delivery is presented in [39]. The platform adapts resources of cloud services and routing with respect to fulfillment of SLA's (Service Level Agreements) for throughput and availability. A governance manager holds a repository of SLA's, templates for elastic strategies and governance parameters. An SLA Manager consisting of an SLA monitor and an SLA controller monitors SLA parameters and issues commands to route incoming messages appropriately depending on commands received from the governance manager. An elasticity manager monitors elasticity parameters and applies elasticity strategies on behalf of the governance manager. Resource allocation is adapted to changing demand but still SLA's have to be fulfilled.

The system architecture consisting of SLA repository, governance controller and monitor is a pattern that will be reused and applied to the the IoT contract and governance framework. It will be extended beyond service level agreements to include also access control, data quality as well as pricing and payment. Controlling in our framework is restricted to simple access control and payment and does not provide elaborate elasticity mechanism as described in the work above.

CHAPTER 3

Scenarios and Requirements

3.1 Scenarios

The first step to derive requirements for contract definition and governance is to look at a set of real world scenarios. We selected three scenarios from the domains of industrial control, smart cities and crowd sensing. Based on the scenarios a set of common requirements is identified and described. The details of the scenarios are elaborated in the following three sections.

3.1.1 Monitoring and Controlling HVAC Systems in Base Transceiver Station (BTS)

A telco operator is running an IoT infrastructure for operation and maintenance of BTS support equipment such as HVAC (Heating, Ventilation and Air Conditioning), power generators or electricity backup systems. In each of the thousands of geographically distributed BTS an IoT gateway connects to sensors and actuators that interface with the BTS equipment. Each IoT gateway runs a set of software defined IoT units that encapsulate sensors, actors and IoT capabilities [40]. The gateway communicates via an MQTT broker with a complex cloud based monitoring and control system providing analytics, optimization, storage via Hadoop and query capabilities . Services in the central application analyze the sensor data and, if necessary, send configuration and control commands via MQTT to the IoT gateway. IoT units in the gateway will then execute those commands. Maintenance of HVAC equipment is outsourced to one or more companies that deploy and run their own IoT units for HVAC monitoring and control in parallel on the same IoT gateway. Since the HVAC is a critical component for operation of a BTS the third party IoT units have to operate properly but may not affect other IoT units (e.g. monitoring other components of the BTS such as radio access or network connectivity) running on the same gateway.

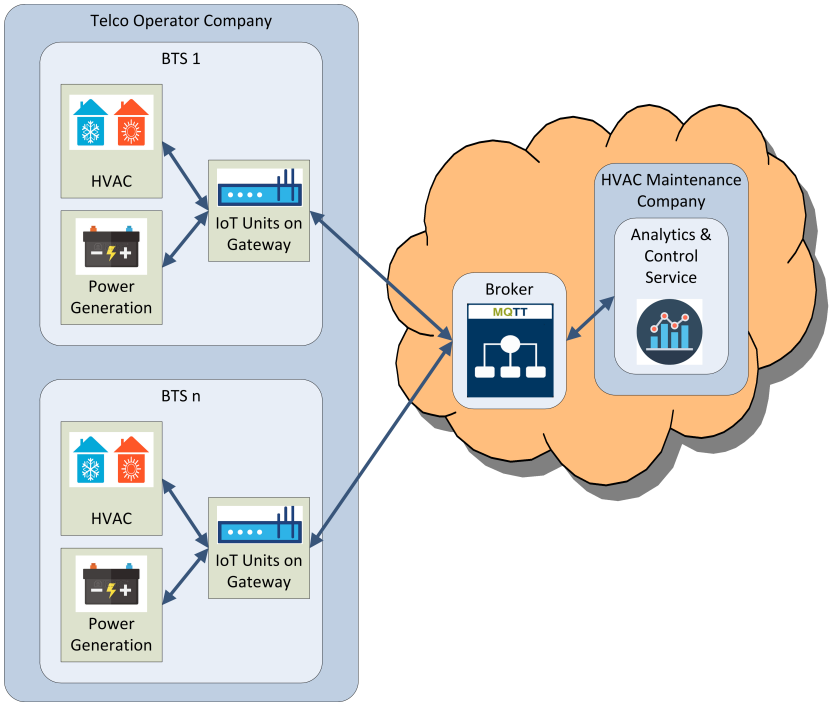

Figure 3.1: BTS Scenario

From point of view of the HVAC maintenance company following contract terms should be defined in the contract:

- access to the sensors and actuators required for operation of the HVAC maintenance analytics and control service.
- required quality of data delivered by the sensors (accuracy, completeness).
- required availability and responsiveness of the actuators.
- required processing power and storage capacity for the HVAC management IoT units running on the IoT gateway.
- required availability of the IoT gateway for running the HVAC management IoT units.
- price model and pricing requested for the HVAC maintenance analytics and control service, e.g. a price per month for every BTS managed.

From point of view of the telco operator following terms should be defined in the contract:

- restriction of access to sensors and actuators required by the HVAC IoT units.

- provided guaranteed processing power and storage capacity for the HVAC IoT units running on the IoT gateway.
- QoS criteria for the HVAC maintenance analytics and control service, e.g. maintaining a constant temperature of 20 degrees Celsius for 99 % of the time.
- availability of the HVAC maintenance analytics and control service, e.g. minimal 99.9 % with a maximum unplanned downtime of 2 hours/year and a maximum planned downtime of 5 hours/year.
- price model and pricing requested for running the IoT maintenance units on the IoT gateway, e.g. usage based fees depending on CPU power and storage consumed.

Major contract relevant points of the scenario are access control, data quality of sensors, availability of actors, resource management on the IoT gateway and pricing.

3.1.2 Analysis of Images Capturing from city-wide IoT Gateways

In a smart city IoT gateways, including sensors (e.g. for air quality, temperature and noise) and cameras, are deployed in public places. An example of such a system is the Array of Things project in Chicago [13] where IoT gateways are deployed on street light poles. Data from the sensors is made available for customers in a data marketplace and they may deploy their own IoT units on the IoT gateways. An IoT unit might access the camera pictures to provide traffic counting for urban planning or measuring traffic flow for an intelligent control of traffic lights. Many IoT units for different services will run on the IoT gateway in parallel and IoT units might also collaborate with other IoT units on the gateway.

Similar to the scenario of the HVAC application the company running the camera based analysis service has to make sure in the contract that it gets access to the required resources and the provider of the IoT gateways has to make sure that operation of the IoT units on the gateway is compliant with operation of other IoT units for different services.

From point of view of the camera analytics company following contract terms should be defined in the contract:

- access to the cameras as required for the IoT service, it could be, e.g. constrained to defined time intervals.
- required quality of pictures delivered by the camera, e.g. frame rate, resolution.
- required processing power and storage capacity on the IoT gateway.
- required availability of the IoT gateway.

From point of view of the smart city provider following contract terms should be defined in the contract:

- access rights restricted to usage of the camera.
- provided guaranteed processing power and storage capacity for the camera analytics IoT service.
- price model and pricing requested for running the IoT units on the IoT gateway.
- ensure that data provided by the camera is used only locally for the defined purpose and not sent to the Internet or stored in databases.

Major contract relevant points of the scenario are access control, data quality of sensors and resource management on the IoT gateway.

3.1.3 Crowdsensing using Mobile Signals

Location based applications (LBA's) analyze crowds (flow of people) [41] through signals emitted from people's mobile phones. In most cases data from sensors will be sent and processed on the cloud but there might be also local IoT edge gateways that provide access to sensor raw data [16], [17], [42]. In the scenario a company has deployed an IoT service composed of a set of IoT units that monitor LBA data to ensure security of an event happening in the public place but the IoT service is not allowed to send data to the cloud.

From point of view of the security company following contract terms have to be defined in the contract:

- access to LBA data as required by the IoT service.
- required processing power and storage capacity on the IoT gateway.
- required availability of the IoT gateway.

From point of view of the smart city provider following contract terms have to be defined in the contract:

- access restricted to LBA data.
- provided guaranteed processing power and storage capacity for the security IoT service.
- price model and pricing requested for running the IoT units on the IoT gateway.

Major contract relevant points of the scenario are access control and resource management on the IoT gateway.

3.1.4 Common Requirements from the Scenarios

Analyzing the three scenarios together we can see common goals to achieve for IoT contracts and their included contract terms. On one hand ensure that an IoT unit deployed on an IoT gateway does not conflict with IoT units for other IoT services running on the same IoT gateway, on the other hand ensure that the IoT units for a certain IoT service get the necessary resources to perform their tasks. Required topics for IoT contracts are:

- controlling access to sensors, actuators and data provided by the IoT gateway and used by IoT services
- defining QoS for sensors and actuators provided by the IoT gateway and used by IoT services.
- defining QoD for data provided by the IoT gateway and used by IoT services.
- defining QoS for IoT services.
- controlling usage of processing resources provided by the IoT gateway and used by IoT services.
- pricing of resources provided by IoT gateways.
- pricing of IoT services.

In the next section stakeholders and their concerns will be further elaborated to lead finally to a model for contract definition and governance.

3.1.5 Stakeholders and Concerns

Analysis of the stakeholders and their roles involved in operation of an IoT system is performed in this section. Roles are those of IoT service provider, IoT platform provider (edge gateways or cloud platform), IoT unit provider and IoT service user. In case of the HVAC monitoring scenario the HVAC maintenance company is IoT service provider and IoT unit provider, the telco running the IoT gateways is platform provider and also service user of the HVAC monitoring service. In case of the image analysis scenario the company providing the image analysis service is service provider and IoT unit provider, the smart city is platform provider and IoT unit provider and the customers of the image analysis company are the service users. In case of the crowd sensing scenario the company providing the crowd analysis service is service provider and IoT unit provider, the IoT gateway provider is platform provider and the customers of the crowd analysis company are the service users.

Each of the stakeholders has different concerns with respect to the services they provide or use. From these concerns contract terms and constraints will be derived that provide the base for contract definition.

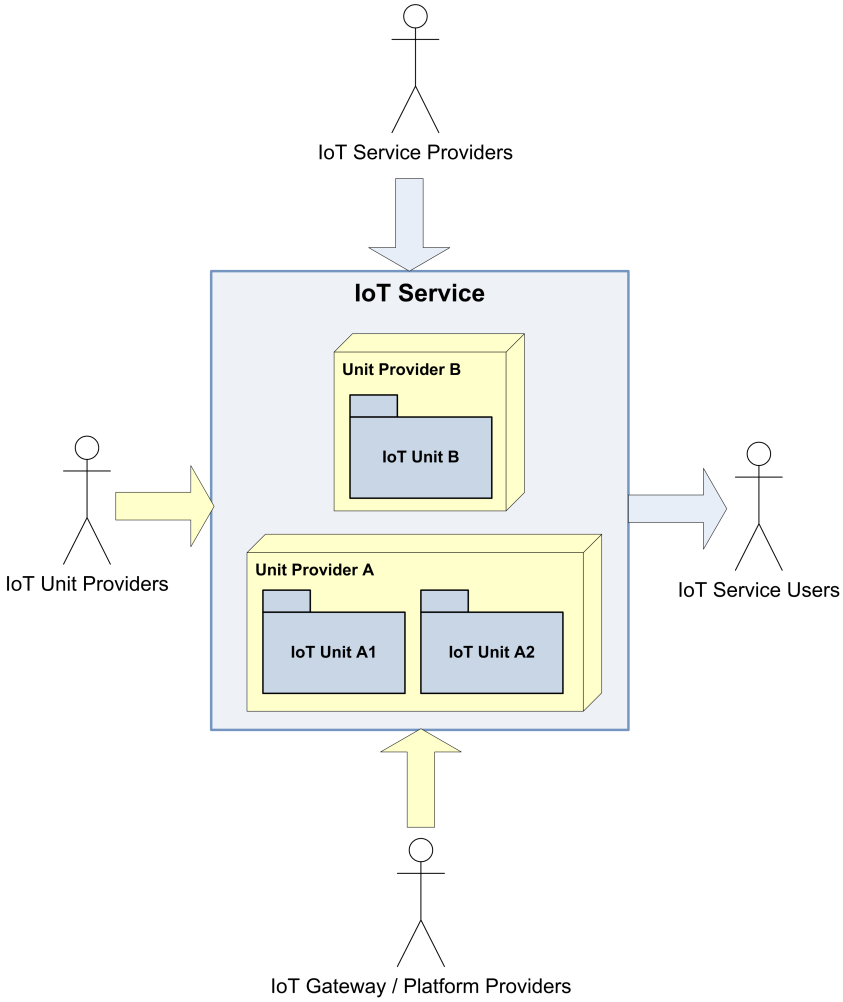

Figure 3.2: Stakeholders

IoT Service Users

They use services from one or more service providers either for their own needs or to provide own enhanced composite services, e.g. access to the sensors and actors of the HVAC equipment in the BTS scenario. The major concerns for IoT service users are:

- *Service availability* which defines basic information such as "service available from 9 am to 6 pm" and quality attributes such as "service available 99.9 % of the time". Availability concerns will have to be reflected in contract terms as part of the Quality of Service (QoS) definitions.
- *Service data quality* as part of QoS covers the requirements of IoT users on data produced by services such as completeness, accuracy or consistency. They will be

used by IoT users to select from different available IoT service providers and to ensure the QoS for their own composite services.

- *Trust relationships.* The IoT service users must be sure that the service is actually provided by an authenticated provider and not faked or altered. Providers and users will work together to provide trust, e.g. by sharing secrets, using certificates in public key infrastructure or relying on a general reputation building mechanism.

IoT Service Providers

Provide services based on a set of IoT units, e.g. the HVAC monitoring and maintenance service in the BTS scenario. The major concerns for an IoT service provider are:

- *Control usage of services* by IoT service users to balance usage requests by different users and manage integrity of the service with respect to constraints imposed by the IoT units where the service is built upon.
- *Authorize usage of services.* Only authorized users identified by the IoT service provider, e.g. by password, API key or certificate have to be allowed use the service. Authorization has to be coupled with fine grained access control to allow users only access to strictly defined services provided by the IoT service provider.
- *Get paid* for providing the service. The IoT service provider wants to define a payment scheme (e.g. pay per use of the service or pay a flat rate per month) that has to be agreed with the service users and get written in a contract. In IoT systems it will be often the case that the payments for a single service are rather small so we have to deal with micro-payments and put them together in larger packets.

IoT Unit Providers

They provide IoT units with a set of defined functions, e.g. the camera units in the Images Capturing from city-wide IoT Gateways scenario. Major concerns for an IoT unit provider are:

- *Control usage of IoT units* by IoT service providers to balance usage requests by different service providers. IoT units often run on small devices / gateways with limited bandwidth and CPU power so that e.g. the number of read requests per minute has to be controlled carefully. Typical constraints related to usage control are frequency of service usage (e.g. max of 10 service calls / hour), amount of data provided by the service (e.g. max 100 KByte / day) or maximum transfer rate (e.g. not more than 1 KB / sec).
- *Authorize usage of IoT units.* Only authorized service providers must be allowed use the IoT unit. This is especially important if IoT units provide services that

allow users to control elements of the IoT unit to operate actors on a real world process, e.g. to switch off an air condition in a building when there is a shortage of electricity supply.

- *Get paid* for providing the IoT unit. Similar to service providers also the IoT unit provider wants to define a payment scheme that has to be agreed with the service providers and get written in a contract.

IoT Gateway / Platform Providers

They run IoT units on cloud platform or on network edge elements such as gateways. The platform provider does not care about the individual functions provided by IoT units but runs them utilizing platform resources such as sensors, actuators, CPU power and network resources. Major concerns for platform providers are:

- *Control usage of resources* by IoT units to ensure that computing and storage resources are allocated as required by the IoT units and overall available resources of the platform are not over committed.
- *Control access to resources* by IoT units so that only authorized IoT units get access and resource allocation request conflicts between different IoT units are resolved, e.g. access to an actuator such as a valve should be granted only for one IoT unit at a time exclusively.
- *Get Paid* for usage of the resources. The IoT platform provider wants to define a payment scheme (e.g. pay per use of the resource) that has to be agreed with the resource users and get fixed in a contract.

3.2 Requirements

Based on the abstraction of scenarios and analysis of stakeholders we have identified different requirements. These are control of access to services and resources, management of resources, service and data quality, authorization and payment. We need a flexible way to define them in our contracts. In the following section we will detail these requirements.

3.2.1 Contract Model

Contract terms should model the basic building blocks of a contract. They cover definitions like access control, authorization, resource usage control and payment but it should be possible to define also arbitrary other terms in a contract. In addition the definitions in contract terms should be reusable in different contracts. Contract templates should provide the base for modeling of concrete contracts. They should contain all the elements that are the same for a certain class of contracts. The basic contract model is abstract in the sense that it is independent of a concrete IoT unit, platform or service. But enforcement of a contract requires inter-working with the concrete implementation

of an IoT unit, platform or service. Therefore it should be possible to define custom logic in the contract model to handle implementation dependent aspects of contract enforcement. Composition is performed when a contract for an IoT service or unit depends on other defined contracts. The logic how the contract terms are combined depends on the functionality of the service or units combined in the contract and therefore should be flexibly definable in the contract model. A concrete contract for an IoT unit or service should be based on templates and allow the definition of arbitrary additional contract information such as contract partners, validity date or descriptions.

The requirements are detailed in Tables 3.1, 3.2, 3.3, 3.4 and 3.5.

Table 3.1: Flexible Contract Terms

ID	CM-001
Title	Flexible Contract Terms
Description	The contract model has to support flexible definition of contract terms since those will differ a lot between different kinds of IoT contracts. Definition of contract terms should be independent of a concrete contract so that reuse of contract terms is possible.
Stakeholders	Service Provider, Unit Provider, Platform Provider

Table 3.2: Flexible Contract Templates

ID	CM-002
Title	Flexible Contract Templates
Description	The contract model has to support flexible definition of contract templates. They are the base for definition of contracts for specific IoT services. A concrete contract should be built by instantiation of a contract template.
Stakeholders	Service Provider, Unit Provider, Platform Provider

Table 3.3: Custom Logic for Contracts

ID	CM-003
Title	Custom Logic for Contracts
Description	The contract model has to support flexible definition of enforcement logic executed in a contract.
Stakeholders	Service Provider, Unit Provider, Platform Provider

Table 3.4: Custom Logic for Composition

ID	CM-004
Title	Custom Logic for Composition
Description	The contract model has to support flexible definition of composition logic executed in a contract.
Stakeholders	Service Provider

Table 3.5: Flexible Contract Definition

ID	CM-005
Title	Flexible Contract Definition
Description	The contract model has to support definition of concrete contracts based on contract templates. Contracts should contain additional information such as contract validity and information about contract partners in a flexible way .
Stakeholders	Service Provider, Unit Provider, Platform Provider, Service User

To fulfill the requirements models will be defined in an object oriented approach using JSON (JavaScript Object Notation) [43], a flexible, widely used and lightweight model. Storage will be provided by the Node4J graph database, reusing the already existing implementation in SALSA [44].

3.2.2 Contract Enforcement and Governance

Enforcement of contracts should be based on monitoring the actual execution of IoT units and services. It is not required to include monitoring capabilities in the contract definition but it should be possible to integrate the contract definition and enforcement with existing monitoring functionalities. It should be possible to execute enforcement logic on a concrete IoT unit or service without requiring modification of the existing IoT unit or service. If contract terms are violated it should be possible to inform all contract partners about the violation. If possible the enforcement logic should be able to execute preventive actions, e.g. denying access if an access violation is detected. If enforcement is combined with external actions, e.g. when a payment has to be done, the enforcement logic should trigger these external actions.

The requirements are detailed in Tables 3.6, 3.7, 3.8, and 3.9.

Table 3.6: Monitoring Support

ID	ENF-001
Title	Monitoring Support
Description	The contract management system has to support integration with existing monitoring facilities as a base for contract enforcement.
Stakeholders	Service Provider, Unit Provider, Platform Provider

Table 3.7: Enforcement Logic

ID	ENF-002
Title	Enforcement Logic
Description	The contract management system has to support provisioning and execution of enforcement logic on IoT units.
Stakeholders	Service Provider, Service User

Table 3.8: Execution of Governance Actions

ID	GOV-001
Title	Execution of Governance Actions
Description	The contract management system has to support execution of governance actions. If a contract term is violated contract partners have to be informed about it and, if available, actions to remedy the contract violation have to taken.
Stakeholders	Service Provider, Service User

Table 3.9: Payment Actions

ID	GOV-002
Title	Payment Actions
Description	The contract management system has to support payment of IoT services in a flexible way. This should include simple payment schemes such as flat fee but also advanced scenarios such as pay by use of services.
Stakeholders	Service Provider, Service User

To fulfill the requirements a contract management system has to be developed. Enforcement logics applied to IoT units will reuse existing script languages such as Javascript [45] and will be injected into IoT units using Aspect-oriented programming via AspectJ [46]. Java and AspectJ were chosen because of its wide spread availability on platforms (e.g. the Raspberry PI used in the performance evaluation) and support for IoT by major development organizations such as Eclipse [47], [48].

3.2.3 Performance

Contract enforcement is an add-on to the normal operation of the IoT unit or service. Therefore it should have as less influence on the operation of the IoT unit or service as possible. Especially we have to consider cases where a high number of events is processed locally on the IoT unit. In this case also the enforcement has to be provided locally to minimize impact on performance of the IoT unit.

Table 3.10: Performance of Enforcement

ID	PERF-001
Title	Performance of Enforcement
Description	Enforcement of contract terms should have as minimal impact on IoT unit performance as possible. Especially it is desired to execute enforcement locally on the IoT unit.
Stakeholders	Service Provider, Unit Provider

3.2.4 Integration

The contract management and enforcement is only a component in a possibly large deployment of existing IoT units and services. Therefore it should be possible to integrate the contract management and enforcement with standard technologies into existing IoT units and services.

Table 3.11: Integration to IoT Unit Management Platforms

ID	INT-001
Title	Integration to IoT Unit Management Platforms
Description	The contract management system should support integration into existing IoT unit management platforms (e.g. SALSA) via standard web service interfaces.
Stakeholders	Service Provider, Service User

3.2.5 Security

Results of the contract enforcement and governance have to be stored in a tamper proof way and made available to all authorized contract partners but access by anyone not authorized should be denied. Results of governance should be stored immutable so that no one can dispute the application of a certain enforcement action. In addition the whole system should work in an environment where there is no trust between contract partners and there is no central authority of trust available.

Table 3.12: Enforcement Logging

ID	SEC-001
Title	Enforcement Logging
Description	The contract management system should support persisting logs of enforcement and governance actions in way that make them immutable and provable by all contract partners.
Stakeholders	Service Provider, Service User

3.3 Use Cases

Based on the high level requirements we go into more detail by describing the system use cases. *Contracts* are built from *Contract Templates* where each contract template holds a set of *Contract Terms* that hold the actual conditions defined for the contract (e.g. allow access to a certain unit or promise to deliver a certain QoS) [36], [14], [49]. Contract terms are linked to *Scripts* that contain executable logic defined in Javascript [45]. Scripts solve the problem that executable enforcement logic that depends on the actual implementation of the concrete IoT unit and on the requirements defined in the contract terms can be handled in a uniform way by the framework. Scripts are injected to IoT units and executed on the unit to enforce contract terms. As IoT service provider

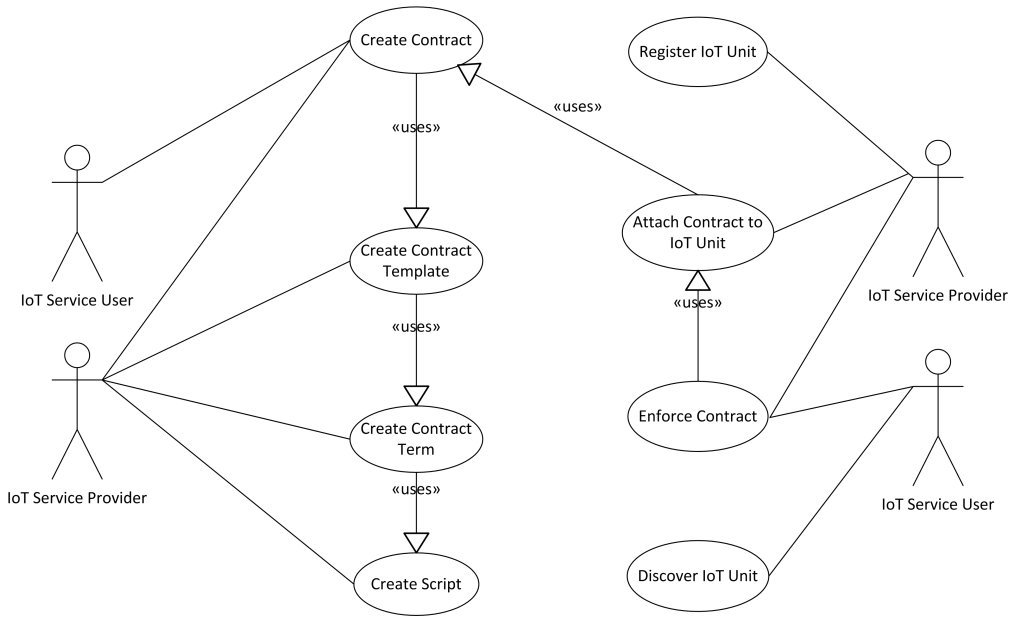

Figure 3.3: Use Cases

we subsume all entities providing a service comprising IoT unit providers, IoT platform providers and IoT application providers. Similar as IoT service users we subsume IoT unit, platform and applications users.

- IoT service providers create the basic entities in the contract management system related to their services, such as contract terms, contract templates and enforcement logic.
- IoT service providers together with IoT service users create a specific contract based on a contract template.
- IoT service providers register their IoT units.
- IoT service users discover IoT units suitable for their needs.
- IoT service providers attach agreed contracts to the IoT units.
- Both IoT service providers and IoT service users enforce the contract.

3.3.1 Create Script

ID	UC-001
Title	Create Script
Stakeholder (s)	IoT Service Provider
Preconditions	none
Scenario	1. Write script 2. Call web service to insert script to contract management system
Postcondition	script persisted in database
Exceptions	Script with same name already existing Script code not valid In both cases an error is returned to the caller

3.3.2 Create Contract Term

ID	UC-002
Title	Create Contract Term
Stakeholder(s)	IoT Service Provider
Preconditions	Scripts created, if they are referenced in the contract term
Scenario	1. Write contract term as JSON object 2. Call web service to insert contract term to contract management system
Postcondition	contract term persisted in database
Exceptions	Contract term with same name already existing In this case an error is returned to the caller

3.3.3 Create Contract Template

ID	UC-003
Title	Create Contract Template
Stakeholder(s)	IoT Service Provider
Preconditions	Set of contract terms available
Scenario	1. Write contract template as JSON object 2. Call web service to insert contract template to contract management system
Postcondition	contract template persisted in database
Exceptions	Contract template with same name already existing In this case an error is returned to the caller

3.3.4 Create Contract

ID	UC-004
Title	Create Contract
Stakeholder(s)	IoT Service Provider, IoT Service User
Preconditions	Contract template available
Scenario	1. Service user and service provider agree on contract terms 2. Write contract as JSON object 3. Call web service to insert contract to contract management system
Postcondition	contract persisted in database
Exceptions	Contract with same name already existing In this case an error is returned to the caller

3.3.5 Register IoT Unit

ID	UC-005
Title	Register IoT Unit
Stakeholder(s)	IoT Service Provider
Preconditions	none
Scenario	1. Write unit registration as JSON object 2. Call web service to insert unit registration to contract management system
Postcondition	IoT unit registered in database
Exceptions	IoT unit with same name already existing In this case an error is returned to the caller

3.3.6 Discover IoT Unit

ID	UC-006
Title	Discover IoT Unit
Stakeholder(s)	IoT Service User
Preconditions	IoT units registered
Scenario	1. Write unit discovery command as JSON object 2. Call web service to query unit registration
Postcondition	IoT unit returned
Exceptions	No IoT found for query In this case an error is returned to the caller

3.3.7 Attach Contract to IoT Unit

ID	UC-007
Title	Attach Contract to IoT Unit
Stakeholder(s)	IoT Service Provider
Preconditions	IoT unit registered
Scenario	1. Write contract attachment as JSON object 2. Call web service to insert contract attachment to contract management system
Postcondition	Contract attached to IoT unit
Exceptions	Contract already attached to IoT unit In this case an error is returned to the caller

3.3.8 Enforce Contract

ID	UC-008
Title	Enforce Contract
Stakeholder(s)	IoT Service User, IoT Service Provider
Preconditions	IoT unit with Contract attached
Scenario	1. Receive contract enforcement message 2. Query contract management system for enforcement logs 3. Execute appropriate actions to enforce contract
Postcondition	Contract enforced
Exceptions	No enforcement data available In this case an error is returned to the caller

CHAPTER 4

Contract Specification and Composition

4.1 Contract Specification

Based on the generic contract data model described in Chapter 3 which is based on existing work in electronic contracts [49], data contracts [38] and policy control [30] we will define now a set of concrete contract terms used in IoT contracts.

4.1.1 Contract Elements for IoT

A contract requires definition of the items that are contracted, the contract parties and the related terms of the contract and is supported by a set of meta-data. IoT specific contract terms such as access rights or service quality are derived from the analysis of scenarios, stakeholders and their concerns. In addition general contract elements such as contract partner definition and meta-data are added for a complete contract definition. In the following the main focus is on IoT specific contract elements.

Service Description

As an IoT service we denote the functionality of an IoT unit or a set of IoT units provided to other IoT services or IoT users. IoT Services are the contracted items. IoT units are connected to sensors and actuators. Sensors read values from a real world process (e.g. temperature, energy consumption and control settings in heating and air conditioning system) as a set of time-stamped data. Actuators interact with a real world process by changing the values of physical elements (e.g setting the heating or cooling controls of an air conditioning system) by application of commands.

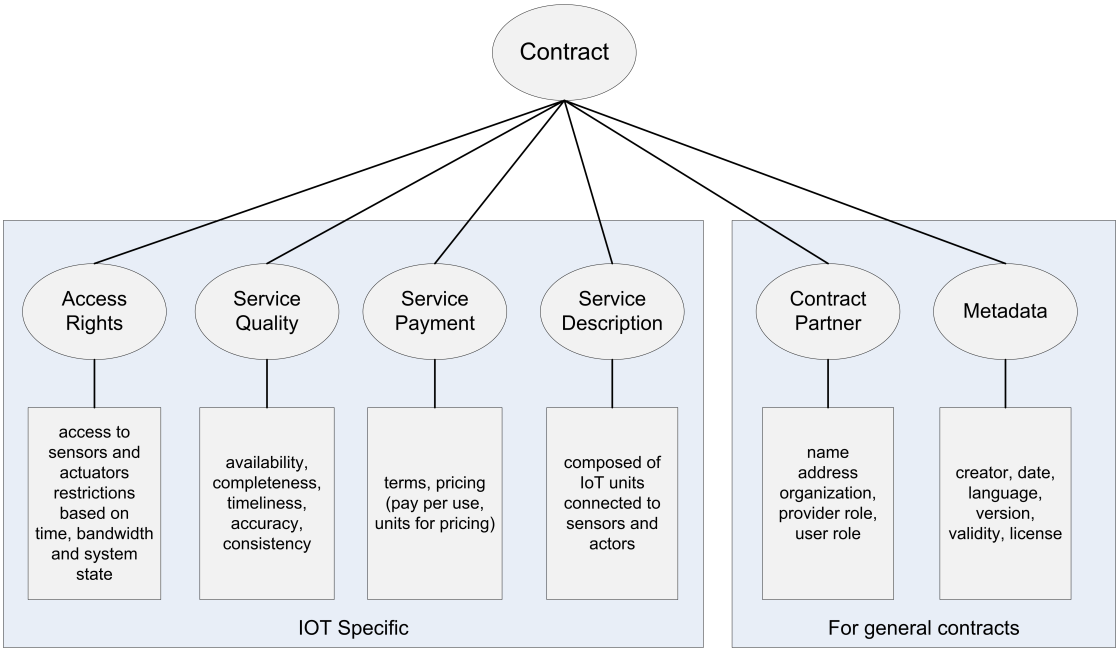

Figure 4.1: Contract Elements

Access Rights

They allow or deny to access a certain part of a service. Access rights can be further restricted with respect to time (access is only allowed in defined time intervals and not always), bandwidth and frequency (amount of services provided per time unit, e.g. max. 1 GB/month or 100 commands/day) or dependent on system state (e.g. not in an emergency condition).

- *Data Access:* allows to get a set of time-stamped data from a service.
- *Control Access:* allows to send a command to a service and modify its state or behavior.

Service Quality

It is bound to provided services, typical measures of service quality [20], [21], [38] are:

- *Availability:* required uptime for the service (e.g. 99.9 % of the time with maximum unplanned downtime of 2 hours / year).
- *Completeness:* ratio of missing values to the number of delivered values, e.g. a smart metering service has to deliver 96 meter reads per day, one each 15 minutes. If one read is missing then completeness is 98.9 %.

- *Timeliness*: age of values, e.g. a monitoring service for a machine has to deliver a condition report at least every day.
- *Accuracy:* indicates the accuracy of the data, e.g., a voltage measured by a sensor is required to have a measurement error below 0.1 %
- *Consistency:* indicates the degree to which a value of an attribute adheres to defined constraints, e.g., if a value lies in certain bounds

Payment

Determines how much a contract partner using a service has to pay for service usage to the provider of the service. Payment properties include payment conditions such as postpaid or prepaid, time allowed to pay a bill or penalties in case of non-payment. Pricing defines a mapping of services to priced items (e.g. pay per individual use, pay a flat fee per month) and the price applied.

Contract Partners

They are (legal) entities participating in a contract. There is no IoT specific definition required to describe contract partners and existing models can be applied. Contract partners might be users or organizations. Both share properties such as name, address or email.

Role

Describes who owns the services and is able to grant rights on them. Services will be provided by a contract partner and used by a set of other contract partners. A contract partner might also be on one hand user of a service and on the other hand (when he has the right to resell the service) provider of a service.

Meta-data

They cover information about the contract itself such as creator, date, language, version, validity and license. Meta-data are not specific for the IoT domain and reuse of existing meta-data models is possible and suggested.

4.1.2 Specification of Contracts

Contracts terms and constraints associated with them are dependent on the actual IoT service and units targeted by the contract, e.g. a service such as the maintenance of heating and air conditioning will require different contract terms than a service for crowd surveillance in a smart city. Therefore the set of contract terms cannot be determined in advance in the contract definition framework and a layered approach that allows to build contracts in a flexible way is applied.

- A *generic model* defines the entities required to build contract templates and instances. It holds contracts terms, constraints attached to the terms and parameters required to instantiate the terms and constraints.

- A *contract template* is built using the generic model and defines common terms of a contract, e.g. that access rights are used in the contract or that a throughput limit is to be applied. Enforcement of constraints is implemented by attaching scripts that are evaluated at runtime when constraint enforcement is performed.

- A *contract* is built on top of the contract template defining concrete values of the contract, e.g. throughput max. 100 KB / hour as well as contract partners and contract items which bind the contract to a specific set of IoT units and their services.

- A *script* is a template written in Javascript for code to be executed on the IoT unit to check and enforce contract terms. Variables in the template referring to constraints in contract terms are replaced with the value defined in the contract when it is attached to an IoT unit. The script code defines how a certain constraint is enforced. As an example a script to enforce access rights checks the entity to be accessed against a list of allowed entities. The list of allowed entities is not defined directly in the script but as a reference to a constraint. Therefore it is set to the concrete value defined for this constraint when an IoT unit is assigned to the contract. In this way scripts can be reused for different contracts terms and contracts.

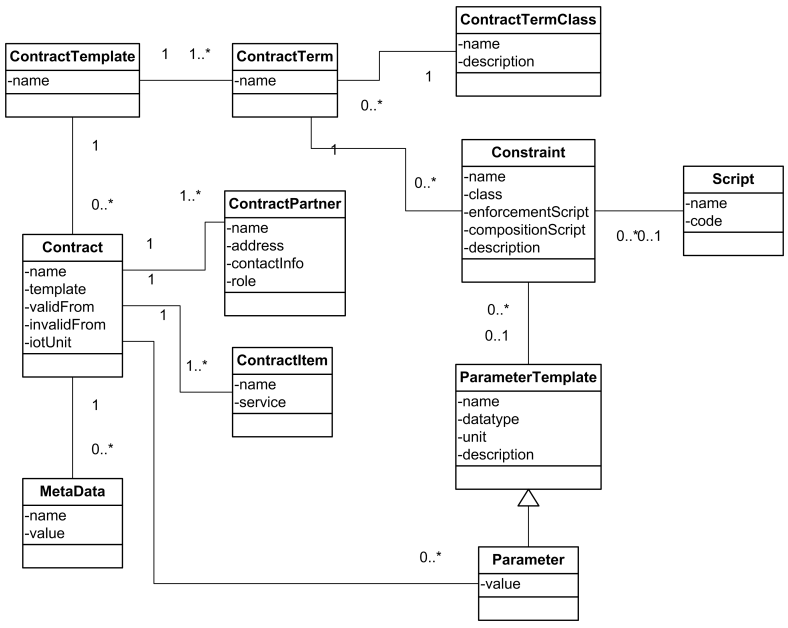

Figure 4.2: Data Model

IoT specific contract elements such as service description, access rights, service quality and payments are mapped to enforceable contract terms. General contract elements are mapped to contract, contract partner and meta-data entities. It is based on similar data models for IoT policies [30] or data contracts [38] but uses an arbitrary user definable set of contract terms instead of a fixed set. In addition it allows to define the contract term constraints and the logic required to enforce the constraints.

4.2 Composition of Contracts for IoT Services

Services are described in an abstract model where each service running on an IoT unit is denoted by a black box receiving inputs from units or other services on input ports and delivering results to users or other services on output ports. Services are composed by connecting output ports of one service to input ports of another service. They can be combined in a pipeline, a tree or, in general a directed acyclic graph.

We assume that contracts are assigned to each service. When the same constraint is applied to a higher level service as to the connected lower level services, the concrete value assigned to the constraint can be derived by composition of the concrete values of the connected lower level services. This procedure can be applied recursively until a base service is reached. Composition of in real world cases requires knowledge of the service function. A service B that combines several input services A1 to An may use the input services either in a disjunctive way (only one of the input services is required so that the combined service can work) or in a conjunctive way (all input services to the combined service have to be available). In the following we look at the different IoT specific contract terms with respect to composition.

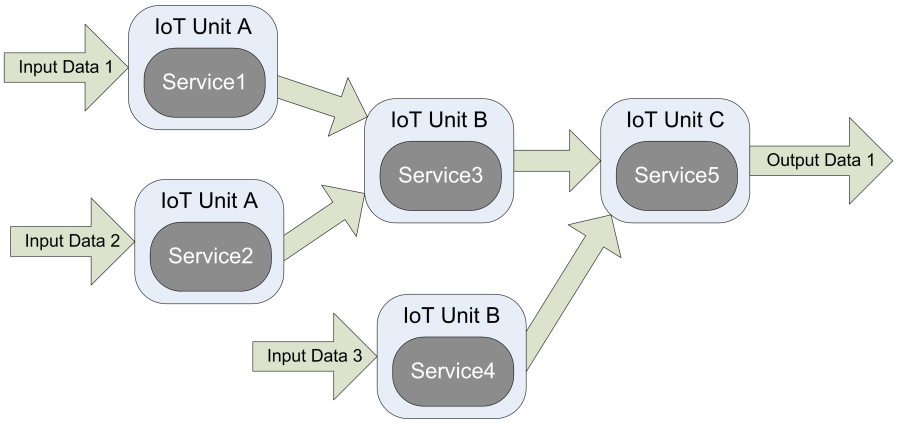

Figure 4.3: IoT Service Composition Abstract Model

4.2.1 Composition of Access Rights

Access rights are related to components of a single service (e.g. access to a sensor connected to an IoT unit providing a base service) and are not applicable to composition since each contract is only responsible to control access rights for its own components.

4.2.2 Composition of Service Quality

We follow the ontologies described in [50] and [51] to define elements of service quality. Based on the concepts in these works we propose following simple approach for the composition of service quality contract terms. The combination is dependent of the type of service quality (e.g. availability or timeliness) and the type of combination (conjunctive or disjunctive). Availability as an example is either is calculated as $avail(B) = min(avail(A1), avail(An))$ in the disjunctive case or as $avail(B) = avail(A1) * avail(An)$ in the conjunctive case. Similar formulas for other types of service qualities are given in "QoS-Aware Composition of Adaptive Service-Oriented Systems" [52] .

4.2.3 Composition of Payment

Payment contract terms of composed services have their own payment and pricing model independent of payment contract terms for services used in the composition (e.g. a temperature reading service might charge 10c for 100 reads but the combined temperature control service might simple charge 10 EUR / month).

4.2.4 Composed Service Example

Given the HVAC example in the scenarios we can have:

- a temperature sensing IoT unit that provides a service to read the BTS temperatures.
- an heating control IoT unit that provides a service to turn the heating on and off.
- an air condition control IoT unit that provides a service to turn the air condition on and off.
- an HVAC control IoT unit that provides a service to switch on heating resp. air conditioning to maintain a desired temperature.

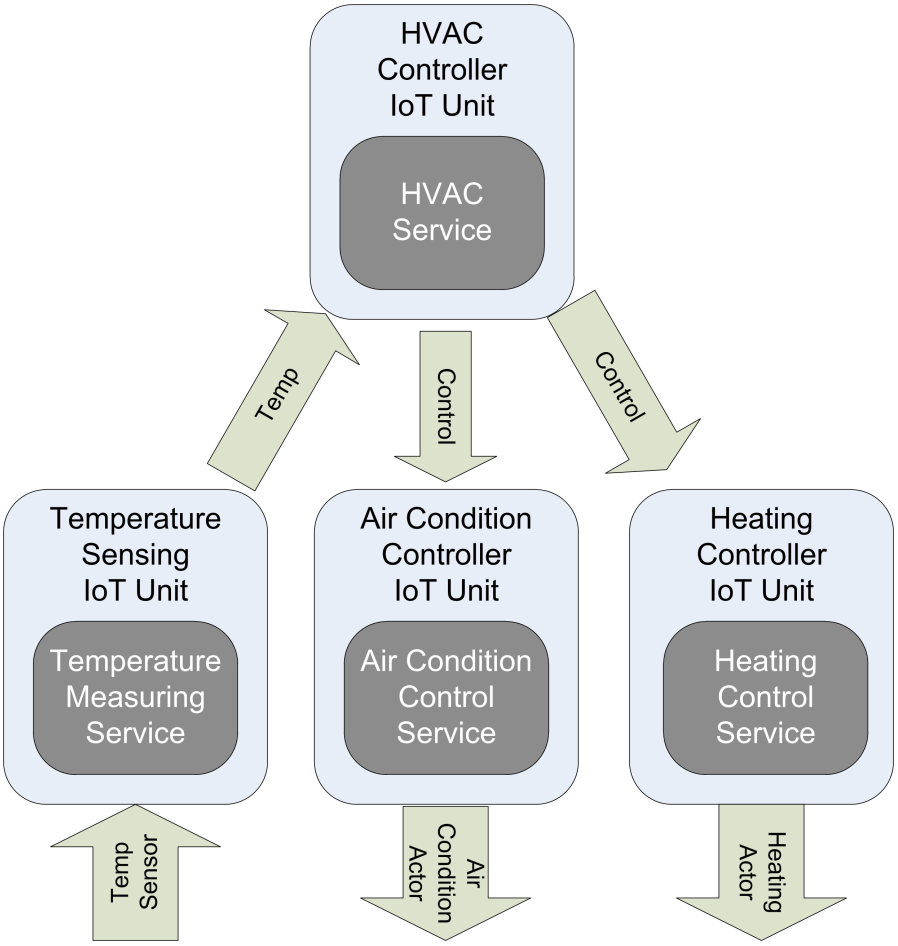

Figure 4.4: Example IoT Service using Composition

We use JSON (Javascript Object Notation) format to represent the contracts and store them in the contract database. A sample contract for the temperature sensing IoT unit in JSON looks like:

Listing 4.1: Temperature Sensing IoT Contract

```
{
  "name":"TemperatureSensingContract",
  "ContractItem":"TemperatureSensingService",
  "ContractPartners":{"Provider":"telco","User": HVAC"},
  "ContractTerms":[
    "AccessRights":
      {"name":"TempSensor",
       "constraint":{"name":"NrOfReads","description":"<1/min"}
      },
    "ServiceQuality":
```

```
        {"name":"Availability",
         "constraint":{"name":"Average","description":"99.9%"}
        }
      "Payment":
        {"name":"UsageFee",
         "constraint":{"name":"Price","description":"1c/read"}
        }
  ]
}
```

For the composed service of the HVAC controller the contract in pseudo code looks like.

Listing 4.2: HVAC Controller IoT Contract

```
{
  "name":"HVACControllerContract",
  "ContractItem":"HVACService",
  "ContractPartners":{"Provider":"telco","User":"HVAC"},
  "ContractTerms":[
  "ServiceQuality":
    {"name" : "Availability",
     "constraint" :
       {"name":"Average",
        "description":"@ComposedBy
            (TemperatureSensingContract,
                AirConditionControllerContract)"
       }
  "Payment" :
    {"name":"UsageFee",
     "constraint":{"name":"Price","description":"5EUR/month"}
    }
  ]
}
```

CHAPTER 5

Governance

5.1 Principles of Governance

According to a definition given in [53] "*IT Governance is to provide leadership, organizational structures and processes to ensure that an organization's IT sustains and extends the organization's strategies and objectives*". Within our framework for contract definition and governance for IoT this means to ensure that contract terms defined for IoT units in a contract and agreed between the contract parties are fulfilled when the units are running and performing their task. Enforcement is the process that supports contract fulfillment by monitoring and checking contract terms for violations. It includes following parts:

- Real-time monitoring of observable attributes of the IoT unit that correspond to contract terms and constraints defined in these terms.
- Supervision to detect violations of constraints.
- Information of all involved contract parties on the constraint violation and record it in a way agreed by the contract partners.
- Simple constraint violations can be treated locally on the IoT unit (e.g. in case of an not allowed access it can be denied), more complex ones will require intervention by a contract partner (e.g. reconfiguration of the IoT units in case performance or availability terms are violated).

5.2 Enforcement of Contracts

In order to enforce a contract term we have to identify observable and measurable service or resource attributes. These attributes are monitored continuously to identify if a

contract term is fulfilled or violated. In case of violation actions to enforce the contract term are performed.

5.2.1 Enforcement of Access Rights

Basic Access Rights

Enforcement of basic access rights, see definition in Chapter 4.1.1, requires to identify the service being accessed as well as the rights of the accessing entity. A service is identified by a unique name, and each service has a provider defined in the contract. The provider grants access rights (read, write, execute) to service users. Observable attributes are the identifier of the requested service and the identity of the contract partner requesting the service. Monitoring can be performed on message level via a proxy intercepting the message and analyzing the content depending on the message protocol (e.g. via XSLT for XML messages) or on service level by instrumenting the service (e.g. via AOP for Java based services). Enforcement then, based on the access rights defined in the contract, either allows or denies access. Listing 5.1 from the BTS maintenance scenario, see Chapter 3.1.1 shows part of a contract defined in JSON [43] that gives the maintenance service access to the temperate sensor with ID=114 to measure the temperature in the server room.

Listing 5.1: BTS Contract with Basic Access Rights

```
{
    "name": "BTSMaintenanceService",
    "contract": {
    "name": "BTSMaintenanceContract",
    "partners": [ {
        "name": "TelcoOpCo",
        "role": "provider"
     },{
        "name": "HVACMaintCo",
        "role": "user"
    } ],
    "items": [ {
         "name": "BTSServerRoomTemperatureService"
    } ],
    "parameters": [ {
        "name": "SensorID",
        "datatype": "String",
        "description": "allowed sensor access",
        "value": "114"
    }, {
        "name": "AccessRight",
        "datatype": "String",
```

```
      "description": "allowed access right",
      "value": "read"
   }]
}
```

Time Based Access Rights

In addition to basic access rights a set of time constraints allows to control the access. They are based on a set of time windows together with definition of repetition (daily, weekly or monthly, weekend or workday). It could be allowed e.g. to access a sensor only between 8 pm and 6 am or on weekends. Monitoring of time is only requiring a trigger when a service is requested. An example of for such a contract term from the BTS scenario is shown in listing 5.2.

Listing 5.2: BTS Contract Term for Time based Access Rights

```
{
   "name": "BTSServerRoomTemperatureAccess",
   "type": "AccessRight",
    "constraints": [ {
        "name": "SensorAccessAtTime",
        "enforcementScript": "AccessRightCheckWithTime",
        "description": "check access to the sensor at time",
        "parameters": [ {
            "name": "SensorID",
            "datatype": "String"
        }, {
            "name": "From",
            "datatype": "Date"
        }, {
            "name": "To",
            "datatype": "Date"
        }]
    }]
}
```

An example for the core logic of a script to check access rights with time constraints is given in listing 5.3. The contract terms and constraints are independent of a concrete IoT unit but the enforcement scripts that work by instrumentation of the IoT unit program code are dependent on the concrete unit and have to be developed specific for each IoT unit code. The values starting with @ are replaced with concrete values of the constraint parameters when the script is loaded to the IoT unit. The variables starting with underscore transport information to the governance controller when a contract violation is detected.

Listing 5.3: Script for Time based Access Rights Enforcement

```
var sensorID = @SensorID;
var from = @From;
var to = @To;
var ts = now();
if ((!Boolean(dataPoint.getName() == SensorID)) ||
    (ts < from) ||
    (ts > to)) {
    _reason='ABORT';
    _log='access not allowed';
}
```

Volume or Bandwidth Based Access Rights

In addition to basic access rights only a certain volume of requests with a defined time frame or consumption of a certain bandwidth are allowed (e.g. not more than 10 requests per hour or maximum bandwidth of 10 KB / sec). If a constraint is violated further access to the service is denied for the defined time. Monitoring of volume and bandwidth is performed by counting the number of service requests and the volume of data transferred. An example of for such a contract term from the BTS scenario is shown in listing 5.4.

Listing 5.4: BTS Contract Term for Bandwidth based Access Rights

```
{
    "name": "BTSServerRoomTemperatureAccess",
    "type": "AccessRight",
    "constraints": [ {
        "name": "SensorAccessWithBandwidth",
        "enforcementScript": "AccessRightCheckWithBandwidth",
        "parameters": [ {
            "name": "SensorID",
            "datatype": "String"
        }, {
            "name": "MaxDataAmount",
            "datatype": "Integer",
            "unit: "KB"
        }, {
            "name": "DataAmountInterval",
            "datatype": "Integer",
            "unit: "minutes"
        }]
    }]
}
```

An example for the core logic of a script to check access rights with bandwidth constraints is given in listing 5.5. The scratchpad is a map that allows to store data from previous invocations of the enforcement script so that decisions based on historical data can be made as in the example where a the amount of data sent with a time interval is checked.

Listing 5.5: Script for Bandwidth based Access Rights Enforcement

```
var sensorID = @SensorID;
var maxDataAmount = @MaxDataAmount;
var dataAmountInterval = @DataAmountInterval;
var ts = now();
var size = dataPoint.getMessageLength();

function calcDataAmount(scratchpad, dataAmountInterval) {
    // calculates the data amount per interval based on
       scratchpad data
  var actualDataAmount = 0;
    for (var [key, value] of scratchpad) {
      if (key > ts - dataAmountInterval) {
        actualDataAmount += value
        } else {
          scratchpad.remove(ts);
        }
    }
    return actualDataAmount;
}

scratchpad.put(ts, size);
if ((!Boolean(dataPoint.getName() == SensorID)) ||
    (calcDataAmount(scratchpad, dataAmountInterval) >
       maxDataAmount)) {
    _reason='ABORT';
    _log='access not allowed';
}
```

System Utilization based Access Rights

Instead of constraining the number of service requests sent or the data volume used it is also possible to use constraints based on global parameters of utilization, e.g. that access is only allowed if CPU or network load is below a certain threshold. Monitoring of such global parameters can be realized by interaction with the operating system or the network management system and is independent of specific services. An example of for such a contract term from the BTS scenario is shown in listing 5.6.

Listing 5.6: BTS Contract Term for System Utilization based Access Rights

```
{
  "name": "BTSServerRoomTemperatureAccess",
  "type": "AccessRight",
  "constraints": [ {
    "name": "SensorAccessWithSystemUtilization",
    "enforcementScript":"AccessRightCheckWithSystemUtilization
       ",
    "description": "check access to the sensor with system
       utilization constraints",
    "parameters": [ {
      "name": "SensorID",
      "datatype": "String"
    }, {
      "name": "CPULoad",
      "datatype": "Integer",
      "unit: "percent"
    }]
  }]
}
```

An example for the core logic of a script to check access rights with system utilization constraints is given in listing 5.7.

Listing 5.7: Script for System Utilization based Access Rights Enforcement

```
var sensorID = @SensorID;
var maxCPULoad = @MaxCPULoad;

function getCPULoad() {
    // gets the CPU loads from the operating system
    ...
}


if ((!Boolean(dataPoint.getName() == SensorID)) ||
    (getCPULoad() > maxCPULoad)) {
    _reason='ABORT';
    _log='access not allowed';
}
```

System State based Access Rights

In this case access constraints are dependent on global parameters but those parameters are depending on state variables of the service, e.g. access is only allowed in normal operating mode but not in an emergency mode. Access to the operating mode has to be provided by an API to enable such a constraint check. An example of for such a contract term from the BTS scenario is shown in listing 5.8.

Listing 5.8: BTS Contract Term for System State based Access Rights

```
{
  "name": "BTSServerRoomTemperatureAccess",
  "type": "AccessRight",
  "constraints": [ {
    "name": "SensorAccessWithSystemState",
    "enforcementScript": "AccessRightCheckWithSystemState",
    "description": "check access to the sensor with system
       state constraints",
    "parameters": [ {
      "name": "SensorID",
      "datatype": "String"
    }, {
      "name": "SystemState",
      "datatype": "enumeration"
    }]
  }]
}
```

An example for the core logic of a script to check access rights with system state constraints is given in listing 5.9.

Listing 5.9: Script for System State based Access Rights Enforcement

```
var sensorID = @SensorID;
var systemState = @SystemState;

function getSystemState() {
  // reads the system state from the IoT unit
}


if ((!Boolean(dataPoint.getName() == SensorID)) ||
  (getSystemState() != NORMAL_OPERATION)) {
  _reason='ABORT';
  _log='access not allowed';
}
```

5.2.2 Enforcement of Quality of Service

Depending on the quality of service term a comparison based on data returned by the service and the constraints defined in the contract is performed. If a constraint is violated, enforcement writes a log entry and raises an alarm. Execution of corrective actions, e.g. to assign more processing resources or switch to backup system is not in scope of this work.

Availability

Monitoring requires access to the services used and calculating the same criteria as defined in the contract term. E.g. if in the contract it is stated that a service availability of 99% for the last hours is guaranteed, all successful and unsuccessful service accesses are traced and the resulting availability is calculated and compared to the constraints defined in the contract. Results of successful and failed reads are stored in the scratchpad and then the actual availability is calculated for the desired interval. If the actual availability is less than the required availability a message is sent to the governance controller to notify the contract violation.

Listing 5.10: Script for Availability Check

```
var requiredAvailability = @RequiredAvailability;
var availabilityInterval = @AvailabilityInterval;
var ts = now();

function calcAvailability() {
  // iterate over all timestamps in scratchpad within the
     availabilityInterval
  // sum the number of successes and failures
  // return the actual availability
  var successes = 0;
  var failures = 0;
  for (var [key, value] of scratchpad) {
    if (key > ts - availabilityInterval) {
      if (value == "success") {
        successes += 1;
      } else {
        failures += 1
      }
    } else {
      scratchpad.remove(ts);
    }
  }
  if (successes == 0) return 0;
  if (failures == 0) return 100;
```

```
  return (successes / failures) * 100;
}

if (datapoint.getStatus() == OK) {
  scratchpad.put(ts, "success");
) else {
  scratchpad.put(ts, "failure");
}

if (calcAvailability() < requiredAvailability) {
  _reason='NOTIFY';
  _log='availability violated';
}
```

Completeness

The ratio of the number of received values to the number of expected values is calculated and compared to the constraints defined in the contract. The script works similar to the one for availability but counts the number of successful read values and compares them with the required number of read values.

Listing 5.11: Script for Completeness Check

```
var requiredCompleteness = @RequiredCompleteness;
var completenessInterval = @CompletenessInterval;
var ts = now();

function calcCompleteness() {
  // iterate over all timestamps in scratchpad within the
     completenessInterval
  // sum the number of successes
  // return the actual completeness
  var successes = 0;
  for (var [key, value] of scratchpad) {
    if (key > ts - availabilityInterval) {
      if (value == "success") {
        successes += 1;
      }
    } else {
      scratchpad.remove(ts);
    }
  }
  return successes;
}
```

```
if (datapoint.getStatus() == OK) {
  scratchpad.put(ts, "success");
)

if (calcCompleteness() < requiredCompleteness) {
  _reason='NOTIFY';
  _log='completeness violated';
}
```

Timeliness

Age of the data is computed from timestamps received in the data and the result is then compared to the contract terms as shown in the following listing.

Listing 5.12: Script for Timeliness Check

```
var requiredTimeliness = @RequiredTimeliness;
var ts = now();

if (now() - datapoint.getTimestamp() > requiredTimeliness)
  _reason='NOTIFY';
  _log='timeliness violated';
}
```

Accuracy

It has to be delivered by the data source as meta-data and is then compared to the contract terms as shown in the following listing.

Listing 5.13: Script for Accuracy Check

```
var requiredAccuracy = @RequiredAccuracy;
var ts = now();

if (datapoint.getAccuracy() < requiredAccuracy)
_reason='NOTIFY';
_log='accuracy violated';
}
```

Consistency

Data returned by the service is checked according to consistency checking terms in the contract, e.g. values must be between a min and max value as shown in the following listing.

Listing 5.14: Script for Consistency Check

```
var min = @RequiredMin;
var max = @RequiredMax;

if (datapoint.getValue() < min || datapoint.getValue() > max) {
  _reason='NOTIFY';
  _log='consistency violated';
}
```

5.2.3 Enforcement of Payment

Contract terms related to payment enable pay-per-use functionality. Usage of a certain service as defined in the contract requires the user to pay a certain amount, either in advance or automatically deducted from an account. The actual calculation of the price for a certain service is out of scope of this work (and normally done by external charging and billing systems) but enforcement of the payment is an interesting topic turning the IoT contract into a smart contract that automatically executes the payment transaction when the payment enforcement constraint is fulfilled, e.g. pay a certain amount after 100 reads from a sensor.

5.2.4 Script Language

Scripts, see Chapter 4.1.2, execute contract enforcement logic. Such a logic is not only dependent on the type of constraint but also on the concrete implementation of the corresponding functionality in the IoT unit, e.g. to check access rights in an IoT unit we have to know how accesses in the IoT unit are handled. We considered following alternatives:

- Direct implementation of enforcement logic in the IoT unit. This would imply that our framework is only applicable to IoT units containing the enforcement logic and it is restricted to the type of constraints handled by the predefined logic.
- Provide a library of enforcement logics in the contract framework. This would also imply that the type of constraints that can be used in the contract is limited and it is not possible to add contract terms with arbitrary constraints.
- Define and implement an own language for definition of enforcement logic. This implies that each IoT unit has to implement execution of the logic written in the own language. In addition the person defining the contracts has to learn the language.
- Use a well-known language for definition of enforcement logics and provide an interpreter for execution of the logic on the IoT unit.

We decided to use Javascript, which is widely known as programming language, to define the enforcement logics and use the open source Rhino Javascript interpreter to execute them on the IoT unit. The Rhino interpreter is injected via Aspect-oriented programming into the IoT unit so that no change of the base code of the unit is necessary.

5.3 Contract Governance Implementation

Contracts are built from contract templates composed of contract terms. Each contract term covers a certain aspect of a contract, e.g. access rights, service quality or payment. Enforcement of contract terms is based on scripts that are assigned to contract terms and executed on IoT units.

5.3.1 Components

The contract definition and governance framework contains following components:

- *Contract Repository* holding contracts, contract terms, contract templates and scripts.
- *Governance Controller* which retrieves contract information and manages attachment of contracts to IoT units.
- *Governance Enforcement* which retrieves enforcement scripts from the governance controller and executes them locally on the IoT unit.
- *IoT Units* running on top of an IoT unit runtime system.

5.3.2 Scenario

The following scenario is based on the diagram given in Fig. 5.1.

- Step 0: Based on existing contract terms and contract templates a contract between IoT service provider and IoT service user is negotiated, defined and created in the contract repository. The contract holds concrete values of all enforcement script parameters that are defined in the contract.
- Step 1: Service user checks the list of available IoT units via a REST web service resource of the governance controller. This requires that all IoT units register with the governance controller. As an alternative the IoT unit execution environment itself provides such a registry function or the list of IoT units is simply known in advance and agreed by service user and service provider.
- Step 2.1: Service user calls the REST web service contract assignment resource of the governance controller to attach a contract to an IoT unit.

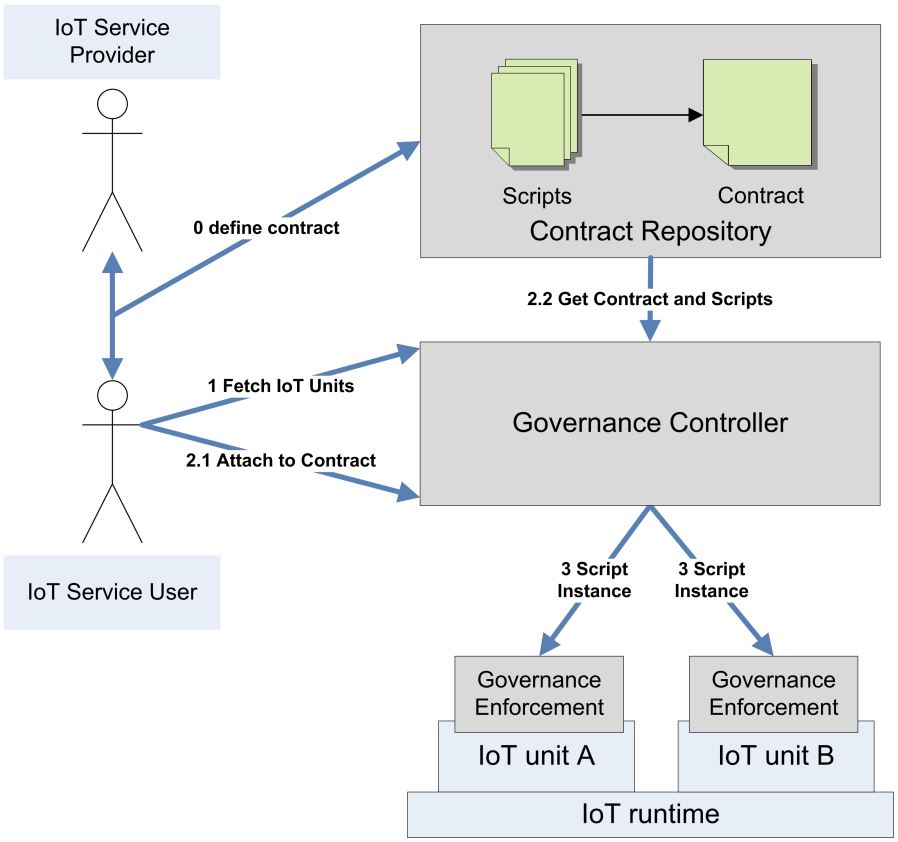

Figure 5.1: Governance Architecture

- Step 2.2: The governance controller fetches the contract information from the contract repository (scripts and parameters defined in the contract) and builds the concrete enforcement scripts including values for all parameters.
- Step 3: The enforcement component injected into the IoT unit fetches the enforcement scripts and executes them when triggered, e.g. every time a message is received or periodically for non message related contract terms.

5.3.3 Example

As an example we take the scenario of monitoring and controlling HVAC systems in Base Transceiver Stations (BTS). The company that runs the HVAC maintenance as IoT service user requires access to a certain set of data points of each BTS for a minimum amount of times per day whereas the operator of the BTS as an IoT service provider wants to ensure that only the defined set of data points are accessed and the maximum amount of data accesses per day is not exceeded. The contract between the maintenance company (called MAINT) and the BTS operator (called BTSOP) includes contract terms that manage the access and amount of requests for each managed BTS unit. The contract

terms will be enforced by scripts that are attached to the contract terms and executed on the IoT unit connected to the BTS. Parameters of the scripts (in the scenario the set of data points and the amount of allowed accesses) are defined in the contract terms, the concrete values of the parameters are defined when a contract is built.

5.3.4 Governance Controller

Its task is to manage attachment of IoT contracts to IoT units, make enforcement scripts available to be loaded and executed on the IoT units, store contract violations and execute payments. When a contract is attached information required for contract enforcement is loaded from the IoT contract data model managed by SALSA and placeholders of parameters in the script code are replaced with actual values from the contract according to following pseudo-code:

Listing 5.15: Replace Placeholders in Contract Template

```
1  ServiceTemplate = fetchServiceTemplate(Unit.ServiceTemplate)
2  ContractTemplate = fetchContractTemplate(ServiceTemplate)
3  FOR EACH ContractTerm IN ContractTemplate
4     ContractTerm = fetchContractTerm(ContractTerm)
5     FOR EACH Constraint IN ContractTerm
6        Script = fetchScript(ContractTerm.Script)
7        FOR EACH Parameter IN Constraint.Parameters
8           replace placeholder in Script with actual contract value
9        END FOR
10    END FOR
11 END FOR
```

5.3.5 Recording Contract Violations via Blockchain

We need to be able to record information about contract violations in a way that records are immutable and verifiable by both service user and service provider. A smart contract running on the Ethereum blockchain is used to store a fingerprint of the contract violation information. Both service user and service provider have an account (an entity able to hold units of a cryptographic currency) on the blockchain. When a contract violation is reported, a transaction is performed on the blockchain. As soon as the transaction is recorded in a block and the block is signed and accepted by the blockchain it is immutable and distributed between all nodes of the blockchain. Both service provider and service user can inspect the blockchain and verify that the transaction occurred. Following steps are performed to setup and use the smart contract in enforcement, see Fig. 5.2.

- Each of the partners involved in the contract creates an account on the blockchain
- When the contract is attached to the IoT unit by the governance controller, contract data and scripts are fetched from the contract repository and a smart contract

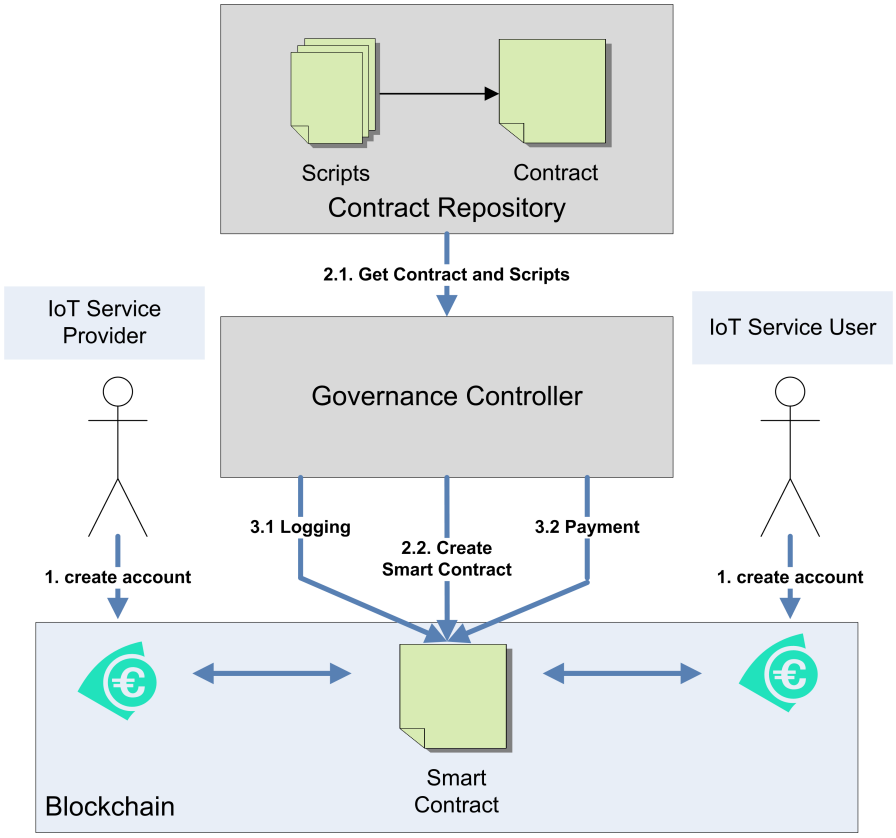

Figure 5.2: Logging and Payment via Blockchain

is created. It contains the account id's of the partners and a fingerprint of the contract and its enforcement scripts. Any partner can then at any time verify the contract in the repository using the fingerprint.

- When a contract violation is observed the governance controller sends a message containing a fingerprint of the log to the smart contract. Any partner can then verify the logs recorded by the governance controller using the fingerprint.

5.3.6 Enforcement of Payments via Blockchain

In a similar way payments due to contract terms are executed via the smart contract by generating a transaction sending a certain amount of coins used in the blockchain from one partner account to the other. Either the cryptographic currency used in the blockchain (e.g. Ether in Ethereum [54]) is directly used for payment or an own coin is created in the smart contract. Creating an own coin has the advantage that its value is independent of the varying exchange rate of the blockchain currency but the supplier of the smart contract has to provide exchange of the own coin to a public blockchain currency or a standard real world currency. Transactions on the blockchain require

transaction costs to be paid in a small amount of the blockchain currency (in Ethereum this is called "Gas" and has a conversion rate to Ether). Transaction fees in the blockchain serve the purpose to give reward to the "miners", those that use their computing power to put transactions into blocks and write them to the blockchain. For transactions with a very small value (called micro transactions), e.g. when data from individual sensor data feeds of an IoT Unit would be paid, the transaction fees become large compared to the value of the transaction. In this case an hybrid approach could be used that handles such micro transactions locally and synchronizes to the blockchain only when a certain amount is exceeded. The smart contract defines what to pay for usage of resources (e.g. computing power), data (e.g. sensor data feeds) or services (e.g. performing regular maintenance) and how to handle exceptions like contract violations. It could be agreed e.g. to pay a penalty when a constraint defined in the contract is violated. All payment transactions are recorded in the blockchain. Each of the contract partners needs to have an account on the blockchain to make or receive payments. The account holds a public address that is used as a source or target of a transaction and a private key that is used to sign the transaction. All transactions on the blockchain are public and can be inspected by anyone using a blockchain explorer, e.g. Etherscan for Ethereum [55]. In our implementation we use a simple own coin, called MetaCoin that is provided as a sample application from the Truffle Ethereum blockchain development framework [56]. Transactions to transfer MetaCoin are performed from the governance controller when a payment message is sent by an enforcement script running on the IoT unit.

CHAPTER 6

Prototype and Evaluation

6.1 Prototype Implementation

Implementation of the IoT contract model is done within SALSA, see Chapter 6.1.1. The governance controller managing the association of IoT units and contracts as well as the provisioning of enforcement is an independent process, see Chapter 6.1.2. Monitoring and execution of enforcement is performed locally on the IoT unit, see Chapter 6.1.3.

6.1.1 IoT Contract Implementation

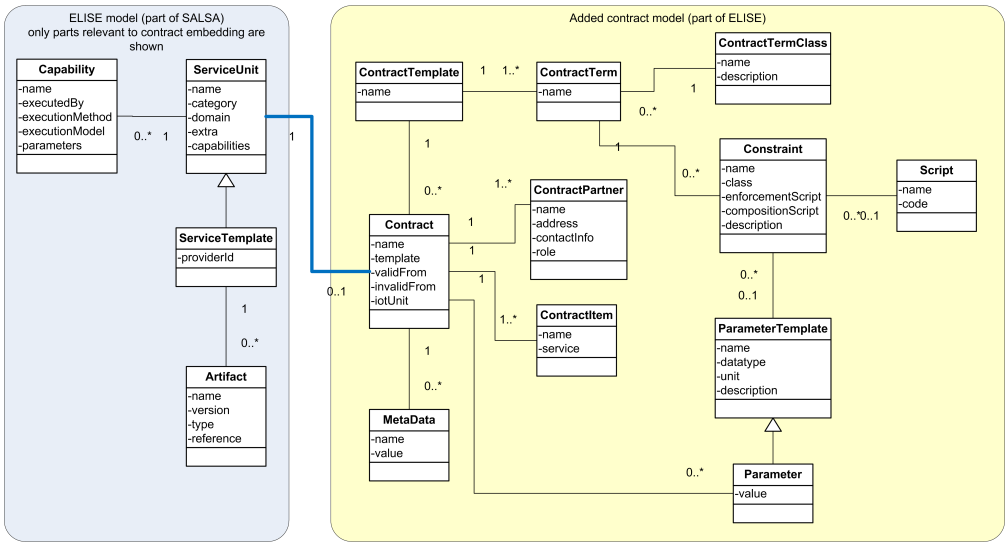

Figure 6.1: Embedding of Contract Model in SALSA

Figure 6.1 shows how the contract data model is connected to the data model of SALSA which is a framework for dynamic configuration of IoT cloud services. SALSA is written in Java using Spring as application framework and Tomcat as servlet container for web services. An IoT service consists of a service topology that itself consists of service units that consist of service instances. It is deployed on a deployment stack such as virtual machines, OS containers such as Docker, application containers or simply as an own application. ELISE is integrated in SALSA providing information of IoT services for the configuration process. It provides a set of APIs among them the repository API that allows to manage IoT components according to the ELISE information model. It contains entities such as service template, service instance, artifact or capabilities. The ELISE information model is designed to be extensible to cover other models such as licensing, contracts or quality. In the work presented here the IoT contract model, see Fig. 6.2 is embedded into ELISE to cover the domain of contract management by adding following entities:

IoT Contract Model Entities

- *Contract* manages a full defined instance of a contract.
- *ContractItem* links between contracts and services.
- *ContractPartner* holds information such as name, address or role of contract partners.
- *MetaData* manages arbitrary additional information about contracts such as validity, start data or version.
- *ContractTemplate* manages blueprints for contracts composed of contract terms.
- *ContractTerm* manages the basic building blocks of contracts.
- *ContractTermType* manages the different types of contract terms defined.
- *Constraint* manages enforcement of contract terms by referencing scripts.
- *Scripts* contain logic interpreted and executed when the contract is attached to man IoT unit. In the prototype Javascript is used as execution language for scripts.
- *ParameterTemplate* manages names and types of script parameters.
- *Parameter* manages concrete parameter values of scripts.

Entities are defined as POJOs (Plain Old Java Objects), annotations manage persisting of entities. In SALSA the Neo4J NoSQL database is used to store entities. Neo4J is a graph oriented database that allows to store configuration graphs in a natural way. The contract data model is represented as a set of graphs where in the figure below full lines

denote inclusion of nodes in the graph and dashed lines denote reference by name. Using reference by name name instead of inclusion enables us to reuse definitions, e.g. to use the same contract term in several contract templates.

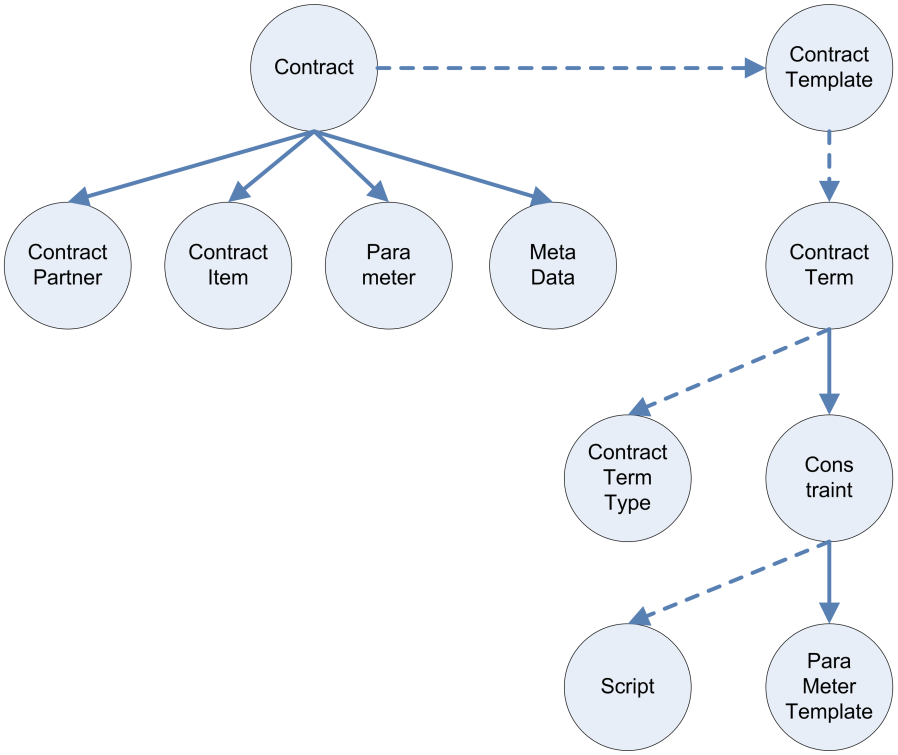

Figure 6.2: Graph Database

Listing 6.1 shows the entity representing a contract term.

Listing 6.1: ContractTerm Entity

```
1  @NodeEntity
2  public class ContractTerm {
3
4  @GraphId
5  Long graphID;
6  private String name;
7  private String type;
8
9  @RelatedTo @Fetch
10 private Set<Constraint> constraints;
11
12 public String getType() {
13   return type;
```

```
14 }
15
16 public void setType(String type) {
17    this.type = type;
18 }
19
20 public Set<Constraint> getConstraints() {
21    return constraints;
22 }
23
24 public void setConstraints(Set<Constraint> constraints) {
25 if (this.constraints == null) {
26   this.constraints = new HashSet<>();
27 }
28   this.constraints = constraints;
29 }
30
31 public String getName() {
32   return name;
33 }
34
35 public void setName(String name) {
36   this.name = name;
37 }
38
39 }
```

Entities are stored by the Spring Neo4J database support by defining interfaces that extend the Graph Repository interface. In the interfaces queries are defined the access the Neo4J database to retrieve the corresponding entity. The following listing shows the repository interface for the contract term.

Listing 6.2: ContractTerm Repository

```
1 public interface ContractTermRepository extends GraphRepository
     <ContractTerm> {
2
3 @Query("match␣(n:ContractTerm)␣return␣n")
4 Set<ContractTerm> listContractTerms();
5
6 @Query("match␣(n:ContractTerm)␣where␣n.name={name}␣return␣n")
7 ContractTerm findByName(@Param(value = "name") String name);
8 }
```

Web Service for Managing IoT Contracts

Access to IoT contract model entities is provided by a set of REST web services. They are built as an extension to the already existing web services of ELISE in the /salsa-engine/rest/elise/extracdg namespace. It is always checked by the web services that creation is only possible with a new unique name to prevent double creation (in case of violation an error 409 CONFLICT is reported back) and that modification, deletion and reading is only possible for existing objects (an error 404 NOTFOUND is returned in such a case).

Each entity supports following access methods:

Table 6.1: REST Web Service Functions

Function	**Method**	**Pattern**	**Description**
ReadAll{Entity}	GET	/{entity}	read all instances
Read{Entity}	GET	/{entity}/{name}	read a specific instance defined by name
Save{Entity}	POST	/{entity}	creates a new instance
Delete{Entity}	DELETE	/{entity}/{name}	delete a specific instance defined by name

Each of the functions described above can be applied to following entities (resources in REST terminology).

Table 6.2: IoT Contract Model Resources

Resource	**URL**	**Description**
Contract	.../elise/servicetemplate	Contract is part of existing SALSA ServiceTemplate resource
ContractTemplate	.../elise/extracdg/contracttemplate	Contract template as blueprint for contract
ContractTerm	.../elise/extracdg/contractterm	Contract term as basic building blocks of contract templates
ContractTermType	.../elise/extracdg/contracttermtype	Definition of types of contract terms
Script	.../elise/extracdg/script	Definition of scripts for enforcement and composition

Technically implementation of the web services is based on Java JAX-RS using annotations as shown in following listing snippet showing the web service to read a contract term.

Listing 6.3: Read ContractTerm Web Service

```
1 @GET
2 @Path("/contractterm/{name}")
3 @Produces(MediaType.APPLICATION_JSON)
4 ContractTerm readContractTerm(@PathParam("name") String name);
```

6.1.2 Governance Controller

It is implemented in Java and Spring and offers web services to attach a contract to an IoT unit, build the concrete enforcement scripts based on the contract and make them available for download by the IoT unit, serve as a registry for IoT units and handle contract violation messaging and logging. The governance controller is run as an own independent application. Spring web starts an embedded Tomcat servlet container to handle the web service requests. Following web service resources are provided:

Table 6.3: Governance Controller Resources

Resource	**URL**	**Description**
Assignment	POST /governor/assign	Assign a contract to a unit
Script	GET /governor/scripts/{unit}	Retrieve scripts for a unit
Registration	GET /governor/register/{unit}	Retrieve registrations for a unit
Registration	POST /governor/register	Create registration for a unit
Logging	POST /governor/log	Log a contract violation to the log
Logging	GET /governor/log/{contract}	Retrieve the logs for a contract
Logging	GET /governor/sclog/{contract}/{id}	Retrieve the fingerprint of a log from blockchain
Payment	POST /governor/payment	Perform a payment on the blockchain

In the prototype the governance controller provides a registry for IoT units. Each unit registers at startup with the registry. Either the unit contains the registration code already in the implementation or it is injected into any Java based IoT unit via AspectJ instrumentation. An alternative would be to have a registration capability already in the IoT unit runtime system. In order to be independent of a specific runtime system, registration via governance controller was chosen.

6.1.3 Monitoring and Enforcement

In the prototype IoT units are treated as a white box and they are assumed to be written in the Java programming language. Instrumentation of the unit is applied by AOP (Aspect-oriented Programming) introducing cross cutting concerns such as monitoring and contract enforcement. Using AOP it is possible to add and execute code that monitors and enforces certain contract terms. The code added by AOP may run before, after or instead of the original code, e.g. enforcement of access rights may deny access if the contract terms do not allow it. Enforcement scripts are written in Javascript, the code added by AOP includes loading of the scripts from the governance controller and execution by the Rhino Javascript engine. Scripts are loaded at startup of the IoT unit and executed locally on the unit when the trigger conditions as defined in the pointcut are met, e.g. a method to process a message received on a port is called. If a constraint defined in a contract term is detected to be violated it is enforced locally if possible (e.g. denying access) and a message is sent to the governance controller to log the violation and execute further actions, e.g. notifying the contract partners.

Enforcement Scripts

AspectJ is used to inject monitoring and enforcement into IoT units, therefore only IoT units written in Java are supported. Enforcement scripts are written in Javascript and executed by the Rhino execution engine. Enhancement to use other programming languages for enforcement scripts could be easily provided for JVM based languages. As an example the injected instrumentation code could use the Groovy language running on the JVM directly. Such a change would not require changes in the governance controller or the contract repository but just in the injected Java code. The following listing shows an example of a pointcut definition. It places itself around reception of messages and also has access to the arguments of the original method, therefore being able to analyze the content of the message performing following steps:

- on first access init the monitor and fetch the scripts
- copy the method call arguments to the Javascript environment
- call the script code
- analyze the script code results and store contract violation log message if necessary
- analyze the script code results and perform payment if necessary
- abort the processing if an access right was violated, otherwise continue with normal processing

Listing 6.4: Pointcut for processDataPoint Method

```
1 @Around("execution(* processDataPoint(..)) && args(fileName,dp,
      thing,port)")
```

```
2  public void aroundProcessDataPoint(ProceedingJoinPoint
       joinPoint,
3    String fileName, DataPoint dp, Thing thing, String port)
       throws Throwable {
4      System.out.println("Around before");
5      if (scripts == null) {
6        initMonitor();
7      }
8      for (Object script : scripts) {
9        try {
10         Context cx = Context.enter();
11         Scriptable scope = cx.initStandardObjects();
12         Object[] signatureArgs = joinPoint.getArgs();
13         Object wrappedDp = Context.javaToJS(dp, scope);
14         ScriptableObject.putProperty(scope, "dataPoint",
             wrappedDp);
15         Object wrappedPort = Context.javaToJS(port, scope);
16         ScriptableObject.putProperty(scope, "port", wrappedPort
             );
17         Object o = ((org.mozilla.javascript.Script)script).exec
             (cx, scope);
18         String r = Context.toString(o);
19         ScriptIF scriptIf = objectMapper.readValue(r, ScriptIF.
             class);
20         if (!(scriptIf.getReason().equals("OK"))) {
21           LogEntry l = new LogEntry();
22           l.setTs(new Date());
23           l.setReason(scriptIf.getReason());
24           l.setLog(scriptIf.getLog());
25           l.setUnit(scriptIf.getUnit());
26           l.setServiceTemplate(scriptIf.getServiceTemplate());
27           l.setId(UUID.randomUUID().toString());
28           String data = objectMapper.
                 writerWithDefaultPrettyPrinter().
                 writeValueAsString(l);
29           send(System.getProperty("thingGovernor") + "/governor
                 /log", data);
30         }
31         if (Integer.parseInt(scriptIf.getAmount()) > 0) {
32           PaymentEntry p = new PaymentEntry();
33           p.setAmount(scriptIf.getAmount());
34           p.setSender(scriptIf.getSender());
35           p.setReceiver(scriptIf.getReceiver());
```

```
36              String data = objectMapper.
                    writerWithDefaultPrettyPrinter().
                    writeValueAsString(p);
37              send(System.getProperty("thingGovernor") + "/governor
                    /payment", data);
38            }
39            if (scriptIf.getReason().equals("ABORT")) {
40              System.out.println("Access denied, aborting");
41              return;
42            }
43          } finally {
44            Context.exit();
45          }
46        }
47        joinPoint.proceed();
48        System.out.println("Around after");
49    }
```

Listing 6.5 shows an example of an enforcement script checking access rights. The variable parts marked with the @character will be replaced with concrete values from the contract by the governance controller when the scripts are built for download by the IoT unit. The following script allows access only to data points defined in the contract. If an access violation occurs the original method call is aborted, therefore enforcing to deny access, a log is written and its fingerprint stored in the blockchain.

Listing 6.5: Access Right Checking Script

```
1 var checkDPName = @DPName;
2 if (!Boolean(dataPoint.getName() == checkDPName)) {
3   _reason='ABORT';
4   _log='access not allowed';
5 }
```

Lifecycle and Startup Synchronization

The following figure 6.3 shows the lifecycle including synchronization at startup.

- Assumption is that contract templates and contract terms are available in the contract repository.
- The IoT service provider creates a contract for his service offering in the contract repository. Independently the IoT service registers itself in the governance controller.
- The IoT service user queries the governance controller for available services and then assigns a service to a contract via the governance controller.

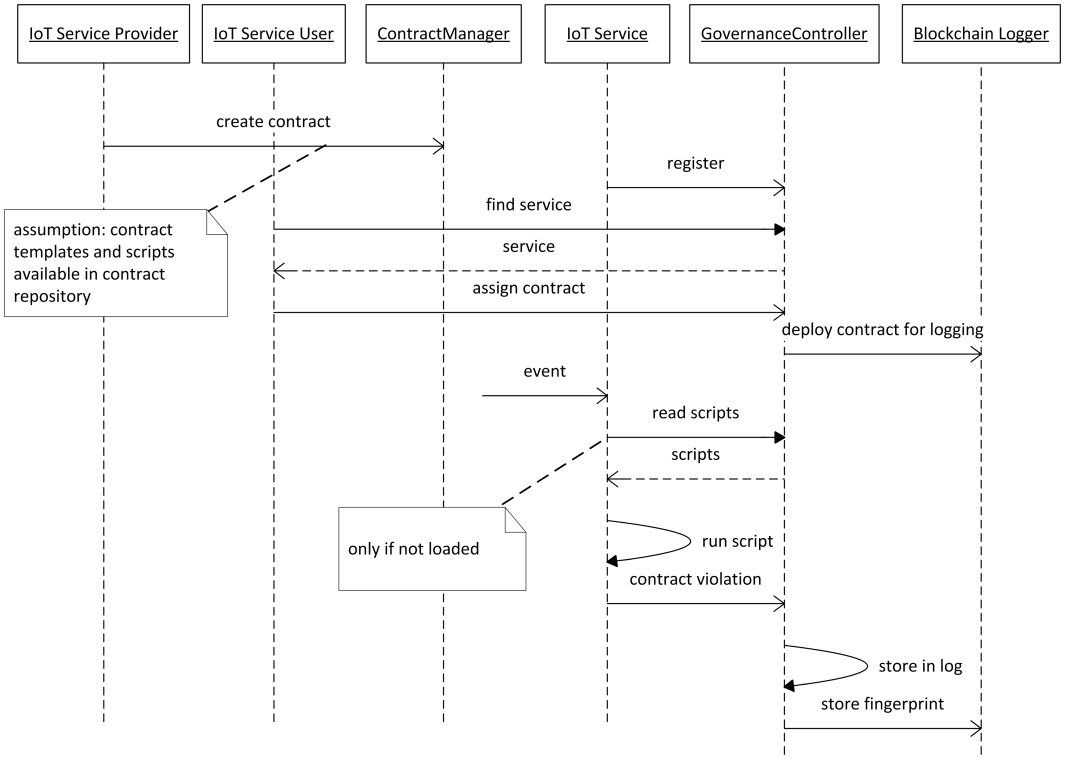

Figure 6.3: Lifecycle and Startup

- When the IoT service receives a trigger event (e.g. a measurement received) it checks for available enforcement scripts and downloads them if necessary.
- It then runs the scripts on the input data of the event, the scripts check for any kind of contract violations. If a violation is detected it is reported to the governance controller.
- The governance controller logs the contract violation in its logger and, in addition, sends a fingerprint of the log entry to the logging smart contract running in the blockchain.
- The fingerprint is stored as a transaction so that an immutable and distributed record of the fingerprint is created in the blockchain.

Reporting Contract Violations

If a constraint checked by an enforcement script is violated immediate action is performed if possible (e.g. if an unauthorized access is detected then access is denied) and the constraint violation is reported to the governance controller. Within the governance controller all violations are logged to a central log repository. Contract partners can query the log repository to check for constraint violations. As an extension it would also

be possible to notify contract partners about such violations via Email, SMS or web service calls. In addition fingerprint of logs are stored in a blockchain as described in the next section.

6.1.4 Logging to Ethereum

A blockchain based on Ethereum is used in the prototype implementation. Ethereum is available as open source on github and allows execution of application code in the blockchain, the code is written in the Solidity language to implement smart contracts. In addition to execution of transactions on the public Ethereum blockchain where real Ether crypto-currency units have to be spent it is also possible to build an own local blockchain for demonstration or test purposes or when a private blockchain, e.g. within a large organisation, should be created. In the prototype the testrpc tool or the Ropsten test network to avoid spending real Ether when testing and demonstrating the system. A smart contract for logging of fingerprints is defined in the Solidity language and then compiled using the solc compiler. The resulting contract interface and binary is then used by the web3j library to build a Java wrapper class for contract deployment and execution. The smart contract is deployed on the blockchain when an IoT unit is attached to a contract in the governance controller. The smart verification contract receives log messages from governance controller and stores a hash value derived from the message in a data structure of the smart contract. Hash values are stored instead of complete messages because storage of large messages is an expensive operation in the Ethereum blockchain. Every contract partner is able to retrieve the hash values and verify the correctness of the actual log messages stored in the governance controller. The smart contract in the blockchain works as a notary verifying the messages on behalf of the contract partners.

Listing 6.6: Smart Contract for Logging

```
1  contract Logstore {
2    string constant public defaultKey = "default";
3    mapping(address => mapping(string => string)) private owners;
4
5    function setLog(string key, string value) {
6      owners[msg.sender][key] = value;
7    }
8    function getLog(address owner, string key) constant returns (
       string) {
9      return owners[owner][key];
10   }
11 }
```

Payment is directly executed on the Ethereum blockchain using the web3j library from Java code as shown in listing 6.7.

Listing 6.7: Payment using Ethereum

```
1  void makePayment(String fromAddress, String toAddress,
       BigInteger amountWei)
2  throws Exception
3  {
4    EthGetTransactionCount transactionCount = web3
5    .ethGetTransactionCount(fromAddress,
         DefaultBlockParameterName.LATEST)
6    .sendAsync()
7    .get();
8
9    BigInteger nonce = transactionCount.getTransactionCount();
10   Transaction transaction = Transaction
11   .createEtherTransaction(fromAddress, nonce, GAS_PRICE_LOCAL,
         GAS_LIMIT_LOCAL, toAddress, amountWei);
12
13   EthSendTransaction response = web3
14   .ethSendTransaction(transaction)
15   .sendAsync()
16   .get();
17
18   TransactionReceipt receipt = waitForReceipt(web3, txHash);
19 }
```

6.2 Evaluation

6.2.1 Evaluation Criteria Types

In order to evaluate the applicability of the IoT contract and governance system we look at following criteria.

- Logical evaluation criteria focus at real world examples and check if proper contracts for them can be formulated in the contract framework. Different contract terms such as those for access rights, service and data quality or payment are analyzed together with their enforcement. In addition the applicability of the monitoring and enforcement component to different environments was evaluated, one environment being a simulator for IoT services, the other one a real IoT gateway running on a Raspberry PI computer.

- Runtime evaluation criteria focus on measurement of important terms such as impact of contract enforcement on performance and IoT service code size.

6.2.2 IoT Units and Runtime System

In order to be able to evaluate the contract definition framework we need to build experiments simulating a set of IoT units (single units but also more complex topologies of IoT units working together). The simulated IoT units should be able to process real world data and connect to simulated external processes. Topology of the IoT units should be configurable and the units should be able to execute processing logic so that complex behavior can be simulated. Contracts then have to deal with the behavior of the units and their topology. We followed a simple processing model to implement the IoT unit simulator.

- Each simulated IoT unit is communicating with other units and outside world via ports. Ports are unidirectional and can handle data and commands.
- Data handled by ports consists of a list of tuples (name, value, timestamp). Commands consist of a tuple (name, list of parameters).
- The unit receives information on input ports and routes it to output ports.
- A unit may execute logic after receiving information on input ports and before sending to output ports. Logic is defined as Javascript code that is interpreted an executed by the unit.
- Communication is possible via file, web services and MQTT. Different types of communication can be used within a unit, e.g. reading data from a file and sending it to another unit via MQTT.
- Configuration of a unit is done via a JSON file that covers definition of ports, data points, commands and routing of input to output ports. In addition the configuration file refers to the name of the file containing the script logic. There is one unit configuration file per unit.
- Configuration of the communication topology is in another JSON configuration file that exists once per simulated set of IoT units and contains the communication paths between the units and physical addresses of units.

The IoT simulator is implemented in Java using the Spring framework. Each unit runs as an own Java process but could also be placed on a Docker container or even a virtual machine. For our experiments we use one Java process per unit to avoid overhead of virtualization environments. When running with input from real world data the IoT unit simulator is able to replay the same set of messages as the real system and therefore able to provide a realistic test setup.

6.2.3 System Setup

Experiment Setup 1 with IoT Simulator

For evaluation we took a set of real-world sample data from the Monitoring and Controling HVAC Systems in Base Transceiver Station (BTS) scenario. The sample data covers alarms and status messages for a set of BTSs each covering a set of data points, in total 1.7 Mio entries in csv format. They were taken from a repository in Github, see [57]. From the set we took a small set of data points for one BTS to come to a subset with an reasonable amount of data for the evaluation. The subset was then fed into the IoT simulator as csv file in the same format. Data contained in the sample is BTS id, data point id, time-stamp and value. The evaluation was performed with following setup.

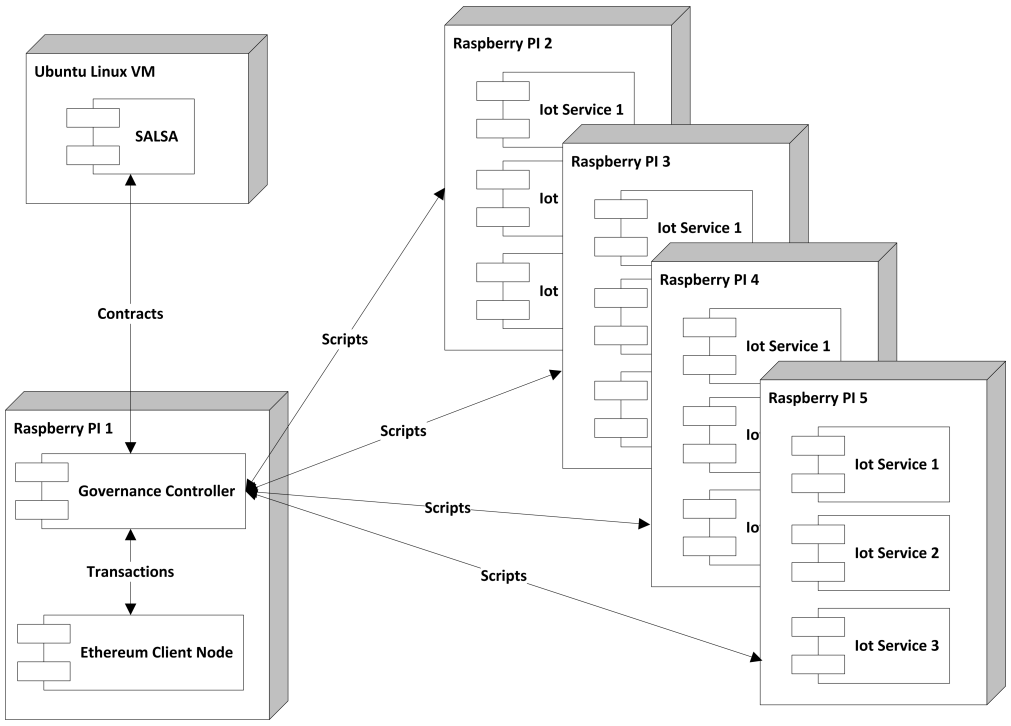

Figure 6.4: Deployment for Evaluation with IoT Simulator

- SALSA is running on an Ubuntu Linux 16.04 server in a virtual machine. 2 GB of main memory and one CPU was allocated to the virtual machine.
- SALSA is used as contract repository. Access to the contract repository is provided via web services on port 8080.
- Contracts are provided as JSON files and inserted into the repository via Linux shell scripts using the curl tool.

- Governance controller running is an own process on a Raspberry PI. Used port for the controller is 8088.
- Ethereum simulator testrpc is running on the same Raspberry PI as the governance controller.
- Simulated IoT units are running on four Raspberry PI nodes, where each node is running up to three units. Data is read into the IoT simulator where each simulated IoT unit is running as an own process. Ports for the IoT units are from 8081 up.

Experiment Setup 2 with IoT Gateway on Raspberry PI

In this real world example a humidity sensor is connected to a sensor IoT unit running on a small low power IoT device. The device uses wireless communication to a Raspberry PI that works as an IoT gateway translating from the CoAP protocol on the wireless side to REST web services sent to a cloud based IoT platform. The monitoring and enforcement component is injected via Aspect-oriented programming into the IoT gateway application monitoring every message sent to the cloud platform.

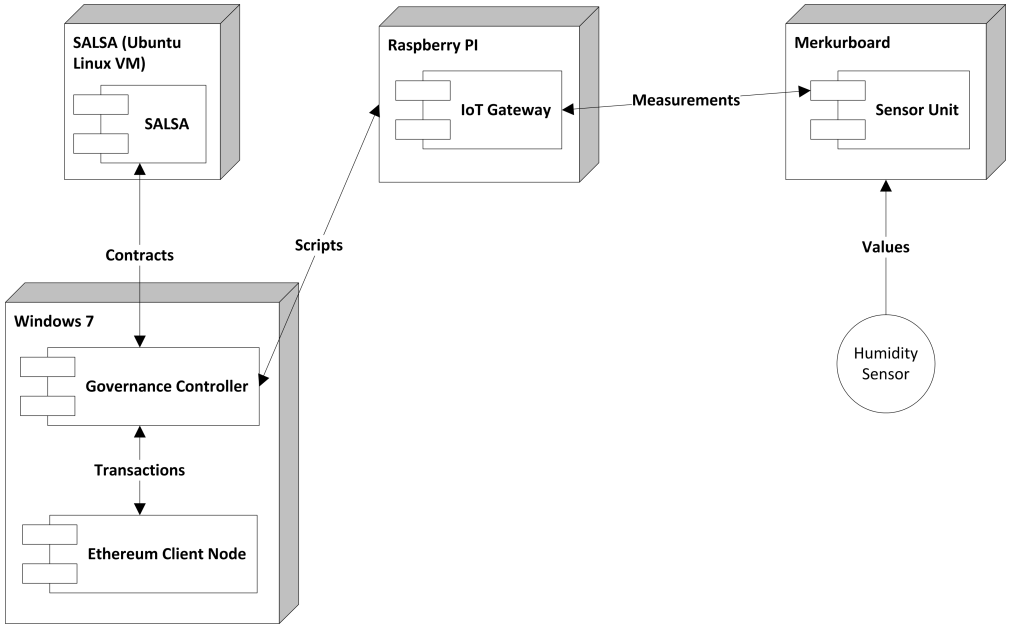

Figure 6.5: Deployment for Evaluation with IoT Gateway on Raspberry PI

6.2.4 Logical Evaluation

Contract for Access Right Check

The first evaluation contract covers access rights of an IoT unit to a set of data points. Depending on the contract terms access to a data point is allowed or denied by the

enforcement script. The contract is composed of a contract term for checking the name of the allowed data point and an enforcement script that checks the name of the data point received in a message with the name defined in the contract.

Listing 6.8: Contract Term for Access Right Check

```
{
  "name": "DPNameCheck",
  "type": "DPChecks",
  "constraints": [{
    "name": "DPNameCheck",
    "type": "",
    "enforcementScript": "DPNameCheck",
    "compositionScript": null,
    "description": "check the allowed data point name",
    "parameters": [{
      "name": "DPName",
      "datatype": "String",
      "description": "data point name",
      "unit": "identifier"
    }]
  }]
}
```

Using the _reason variable the script tells the enforcement core function in the injected monitoring and enforcement component that it should abort further processing if the access right is not granted.

Listing 6.9: Script for Access Right Check

```
var checkDPName = @DPName;
if (!Boolean(dataPoint.getName() == checkDPName)) {
  _reason='ABORT';
  _log='access not allowed';
}
```

In the contract concrete values are defined for the contract parameters. They are then substituted for the formal parameters in the script marked with the @ sign when the script is built by the governance controller and loaded by the monitoring and enforcement component.

Listing 6.10: Contract for Access Right Check

```
{
  "name": "DPNameCheckContract",
  "template": "DPNameCheckTemplate",
```

```
4    "partners": [{
5      "name": "Peter"
6    }],
7    "items": [{
8      "name": "SimpleSensor"
9    }],
10   "parameters": [{
11     "name": "DPName",
12     "datatype": "String",
13     "description": "allowed data point",
14     "unit": "dimensionless",
15     "value": "114"
16   }],
17   "metaData": [{
18   "name": "validFrom",
19     "value": "2016-11-01"
20   }]
21 }
```

Contract for QoS Check

An extension of the simple name check used in the sample above is to check also QoS parameters, e.g. to check that values received in the message are within a defined interval (min, max). Following additions is done in the script for the QoS check.

Listing 6.11: Contract Term for QoS check

```
1  {
2    "name": "DPValueCheck",
3    "parameters": [{
4      "name": "DPMinValue",
5      "datatype": "Integer",
6      "description": "data point min value",
7      "unit": "int"
8    },
9    {
10     "name": "DPMaxValue",
11     "datatype": "Integer",
12     "description": "data point max value",
13     "unit": "int"
14   }]
15 }
```

The script logs any contract violation to the governance controller but does not abort processing the event.

Listing 6.12: Script for QoD check

```
var checkDPMaxValue = @DPMaxValue;
var checkDPMinValue = @DPMinValue;
if (Boolean(dataPoint.getValue() < checkDPMaxValue) &&
    Boolean(dataPoint.getValue() > checkDPMinValue)) {
    _reason='NOTIFY';_log='value out of range'
}
```

Listing 6.13: Contract for QoD check

```
{
    "name": "DPValueCheckContract",
    ...
  "parameters": [{
    "name": "DPMinValue",
    "datatype": "Integer",
    "description": "min value",
    "unit": "int",
    "value": "0"
  },
  {
    "name": "DPMaxValue",
    "datatype": "Integer",
    "description": "max value",
    "unit": "int",
    "value": "1000"
  }]
    ...
}
```

Contract for Timeliness Check

In this sample we run a more complex IoT system composed of 2 sensor IoT units that send data to an analyzer IoT unit running in the cloud. The analyzer requires access to data of the sensors at least every minute to perform its task, this requirement should be reflected in a contract for the analyzer. The sensor IoT units run the enforcement script for the access check, the analyzer IoT unit runs the enforcement script for the timeliness check.

Listing 6.14: Script for Timeliness Check

```
var checkMaxTime = @DPMaxTime;
var newTS = new Date().getTime();
var lastTS = monitor.scratchpad.get(dataPoint.getName());
if (lastTS == null) {
  monitor.scratchpad.put(dataPoint.getName(), newTS);
} else { if ((newTS - lastTS) > checkMaxTime * 1000) {
      _reason='NOTIFY';
      _log="max wait time reached, contract violated"};
  } else {
    monitor.scratchpad.put(dataPoint.getName(), newTS);
  }
}
```

In this script the *scratchpad* which is a data structure provided by the monitor component is used to store temporary variable values between calls of the enforcement script so that it can base its checks not only on actual but also on historical values. The scratchpad is a map of strings to objects. In addition the *log* functionality is used to permanently store and make available contract violation log messages to both contract partners.

Contract with Payment

The script supplies the payment information (amount, sender, receiver) to the governance controller.

Listing 6.15: Script for Payment

```
_amount = @DPPaymentAmount;
_sender = @DPPaymentSender;
_receiver = @DPPaymentReceiver;
```

Evaluation Result

On the positive side logical evaluation shows that scripting is powerful. Javascript as programming language is Turing complete and therefore any algorithm for enforcement can be implemented in it. In addition scripts can also be implemented in other Java based languages. e.g. Groovy or in other interpreted languages such as Python. On the negative side writing enforcement scripts in a general purpose programming language requires programming skills and scripts have to be adapted to each type of IoT unit. It would be an improvement to build a domain specific language for enforcement so that users can concentrate on the enforcement logic and do not need to handle low level programming.

6.2.5 Runtime Evaluation

Evaluation Criteria

Following criteria were evaluated with respect to runtime performance using simulated IoT units running on the Raspberry PI cluster.

- number of contracts assigned to a unit.
- number of contract terms assigned to a contract.
- complexity of contract terms in terms of number of constraints.
- number of units.
- number of events processed.
- percentage of events resulting in a contract violation.
- whether contract violations are logged to the blockchain.

Table 6.4: Experiment Setup

ID	number of units	number of events total	constraints per contract	contract terms per contract	contracts per unit	contract violation percentage	use block-chain
M1	1	10	1	1	1	20	No
M2	1	100	1	1	1	20	No
M3	1	1000	1	1	1	20	No
M4	1	2000	1	1	1	20	No
M5a	5	5000	1	1	1	20	No
M5b	10	10000	1	1	1	20	No
M5c	15	15000	1	1	1	20	No
M6	1	1000	10	1	1	20	No
M7	1	1000	1	10	1	20	No
M8	1	1000	1	1	10	20	No
M9	1	1000	1	1	1	0	No
M10	1	1000	1	1	1	40	No
M11	1	1000	1	1	1	60	No
M12	1	1000	1	1	1	80	No
M13	1	1000	1	1	1	100	No
M2bc	1	100	1	1	1	20	Yes
M3bc	1	1000	1	1	1	20	Yes

Influence of Different Number of Events Processed

In the first set of experiments we evaluate influence of different number of events processed. Expectation is that from a certain number of events on the processing time will increase linear with the number of events. For results see table 6.5. From the measurements we see that throughput (number of events per second) stays the same from 1000 events on so that any effects of startup processing can be neglected. We will use 1000 events to process for the other experiments.

Table 6.5: Different Number of Events

ID	number of events	script loading time (sec)	execution time (sec)	events per sec
M1	10	1.38	0.79	12.66
M2	100	1.38	4.13	24.21
M3	1000	1.35	30.32	32.98
M4	2000	1.38	59.16	33.81

Influence of Different Number of Units running

We run IoT units on 4 different Raspberry Pi machines, first 1 unit on one machine, then 4 units spread over 4 different machines and then 8 and 12 units spread over 4 machines. For results see table 6.6.

Table 6.6: Different Number of Units

ID	number of units	script loading time (sec)	execution time (sec)	events per sec
M3	1	1.35	30.32	33.81
M5a	4	1.65	29.92	133.69
M5b	8	2.55	33.76	236.97
M4	12	3.26	39.02	307.53

From the measurement we see that throughput increases nearly linear with usage of parallel execution on IoT units. Processing of enforcement is parallel in different IoT units since they are running on different machines. Processing of messages on the governance controller also makes use of parallel execution depending on available processing capacity due to implementation of stateless web services running on a Spring application container. Increased script loading time (measured as from starting of script loading on the first unit of the first machine to finishing script loading on the last unit of the last machine) increases because not all units are started on the Raspberry Pi at the same time, especially if more units are run on the same machine. The same applies to execution time (also measured from starting event processing on the first unit of the first machine to finishing on the last unit of the last machine) slightly increasing although throughput on one

machine stays the same.

We see that the system scales well (approximately 10 fold increase in throughput from 1 - 12 units, even when the governance controller is running on a RaspberryPI) and from architectural point of view is able to handle a large number of units.

Influence of Different Contracts

Different number of constraints, number of terms per contract and number of contracts per IoT unit were applied. For results see table 6.7.

Table 6.7: Different Types of Contracts

ID	number of constraints	number of contract terms	number of customers	script loading time (sec)	execution time (sec)	events per sec
M3	1	1	1	1.35	30.32	32.98
M6	10	1	1	1.51	91.15	10.97
M7	1	10	1	1.51	88.47	11.3
M8	1	1	10	1.53	90.16	11.09

From the measurement we see that execution time increases linear with the number of constraints, contract terms and contracts per unit. There is no noticeable difference between increasing the number of constraints, contract terms and contracts per unit which is also visible from the design of the system since such an increase leads to an increase in the number of scripts executed during runtime independent how they are built from. For a 10 tenfold increase in the number of scripts from 1 to 10 we see an approximately 3 times reduction in throughput.

We see that even a large number of constraints added leads to reasonable (less than linear) system performance.

Influence of Different Percentage of Contract Violations

Event content was set in a way that from 0 up 100 % of the events cause a contract violation. For results see table 6.8.

Table 6.8: Different Contract Violation Percentage

ID	contract violation percentage	script loading time (sec)	execution time (sec)	events per sec
M9	0	1.30	24.07	41.55
M3	20	1.35	30.32	32.98
M10	40	1.35	34.53	28.96
M11	60	1.31	39.60	25.25
M12	80	1.32	39.51	25.31
M13	100	1.30	49.36	20.26

From the measurement we see that throughput decreases with the number of contract enforcement messages sent to the governance controller. Going from no enforcement messages to 100 % decreases throughput by about 50 %.

We see that reporting of contract violations reduces throughput but moderatly. Even for a full 100% of contract violations reported, which is an unlikely case meaning that the system is not able to fulfill the contract at all, we are able to deliver a reasonable performance.

Influence of Enforcement Technologies

Tests were run with logging of enforcement via blockchain and without. For results see table 6.9.

Table 6.9: Enforcement Performance via Blockchain

ID	number of events	use blockchain	script loading time (sec)	execution time (sec)	events per sec
M2	100	No	1.38	4.13	24.21
M3	1000	No	1.35	30.32	32.98
M2bc	100	Yes	1.45	15,47	6.46
M3bc	1000	Yes	1.37	112.77	8.48

From the measurement we see that enforcement via blockchain adds considerable performance penalties. The performance impact will be even bigger if we move from the testrpc blockchain simulator to using a real blockchain network since a real network will be able to handle transaction only at the order 20 transactions per second for the whole worldwide network.

We see that only especially important contract enforcement messages should be logged to the blockchain otherwise performance will be severely decreased.

6.2.6 Size of Enforcement Code

Evaluation was performed by analyzing the code size for the monitoring part added to the IoT simulator and the gateway and analyzing the code size for the access right monitoring script.

We see that both size of the monitoring and enforcement component as well as size of the enforcement scripts is rather small and has no substantial influence on the size of the IoT unit (size of IoT simulator without monitoring was 20 MB, size of gateway was 3.5 MB, most of the size of IoT simulator and gateway is caused by included libraries.)

Table 6.10: Monitoring and Enforcement Code Size

	size (characters)	class file size (KB)
Monitor IoT simulator	7716	16
Monitor gateway	7818	15
Access Right Script	122	n/a
QoD check script	218	n/a

CHAPTER 7

Summary

In this thesis we introduced a framework for IoT contract definition and governance including monitoring of constraints imposed by the contract and enforcement of contract violations by linking them to the blockchain. Contracts are built from contract templates which are composed of a set of contract terms. Each contract term includes a set of constraints where for each constraint the parameters and a piece of Javascript code for monitoring and enforcement is defined. Contract terms are defined to cover access rights, quality of service and data such as throughput, completeness or accuracy as well as payment and pricing conditions.

When a contract is assigned to an IoT unit, the constraints of the contract are collected and script code is generated and injected into the IoT unit via Aspect-oriented programming. During runtime the injected scripts check the constraints at defined points of the IoT unit execution (e.g. when a new value is received for a data point or periodically every x seconds) and, if a constraint violation is detected, send a message to the enforcement component. There enforcement covers informing contract parties about the constraint violation, logging a hash of the message to the blockchain for immutable storage and inspection by contract parties and initiation of payment transactions on the blockchain transferring coins from one wallet to the other.

Evaluation results show that the introduced framework is able to cover real world scenarios and provides performance and scalability to handle the workloads related to the scenarios. A future research direction in this area is the introduction of automatic reconfiguration of the IoT unit or infrastructure in case of a constraint violation e.g. to add more IoT units if throughput is below the threshold defined in the contract. Another is to add a domain specific constraints checking language so that contract enforcement does not require programming knowledge of script code.

Bibliography

[1] "Watson internet of things." https://www.ibm.com/internet-of-things/learn/what-is-iot. Accessed: 2017-12-21.

[2] S. Li, L. D. Xu, and S. Zhao, "The internet of things: A survey," *Information Systems Frontiers*, vol. 17, pp. 243–259, Apr. 2015.

[3] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen, "A survey on internet of things from industrial market perspective," *IEEE Access*, vol. 2, pp. 1660–1679, 2014.

[4] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things Journal*, vol. 1, pp. 22–32, Feb 2014.

[5] "Evrythng license agreement." https://evrythng.com/legal/license-agreement. Accessed: 2018-06-01.

[6] "Cumulocity license agreement." https://cumulocity.com/terms-and-conditions. Accessed: 2018-06-01.

[7] "The internet of things (you can sue about)." https://www.forbes.com/sites/danielfisher/2015/06/03/the-internet-of-things-you-can-sue-about/#72a66571cb12. Accessed: 2018-06-01.

[8] "Internet of things- five most famous legal lawsuits)." https://medium.com/@legalresolved/internet-of-things-five-most-famous-legal-lawsuits-84120106ac8. Accessed: 2018-06-01.

[9] F. Li, M. Vogler, S. Sehic, S. Qanbari, S. Nastic, H.-L. Truong, and S. Dustdar, "Web-scale service delivery for smart cities," *Internet Computing, IEEE*, vol. 17, pp. 78–83, July 2013.

[10] B. Mandler, F. Antonelli, R. Kleinfeld, C. Pedrinaci, D. Carrera, A. Gugliotta, D. Schreckling, I. Carreras, D. Raggett, M. Pous, C. V. Villares, and V. Trifa, "Compose – a journey from the internet of things to the internet of services," in *Proceedings of the 2013 27th International Conference on Advanced Information*

Networking and Applications Workshops, WAINA '13, (Washington, DC, USA), pp. 1217–1222, IEEE Computer Society, 2013.

[11] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, pp. 70–95, Feb 2016.

[12] P. Sotres, J. R. Santana, L. Sánchez, J. Lanza, and L. Muñoz, "Practical lessons from the deployment and management of a smart city internet-of-things infrastructure: The smartsantander testbed case," *IEEE Access*, vol. 5, pp. 14309–14322, 2017.

[13] C. E. Catlett, P. H. Beckman, R. Sankaran, and K. K. Galvin, "Array of things: A scientific research instrument in the public way: Platform design and early lessons learned," in *Proceedings of the 2Nd International Workshop on Science of Smart City Operations and Platforms Engineering*, SCOPE '17, (New York, NY, USA), pp. 26–33, ACM, 2017.

[14] H.-L. Truong, M. Comerio, F. De Paoli, G. Gangadharan, and S. Dustdar, "Data contracts for cloud-based data marketplaces," *Int. J. Computational Science and Engineering*, vol. 7, no. 4, pp. 280–295, 2012.

[15] B. Zhang, N. Mor, J. Kolb, D. S. Chan, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz, "The cloud is not enough: Saving iot from the cloud," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, (Santa Clara, CA), USENIX Association, 2015.

[16] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere, "Edge-centric computing: Vision and challenges," *SIGCOMM Comput. Commun. Rev.*, vol. 45, pp. 37–42, Sept. 2015.

[17] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, pp. 14–23, Oct 2009.

[18] "Data fusion mechanism based on a service." `https://www.google.at/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiq_sT5_vzXAhUIyKQKHf45BzoQFggrMAA&url=http%3A%2F%2Fwww.jornadassarteco.org%2Fjs2012%2Fpapers%2Fpaper_61.pdf&usg=AOvVaw1Il3SDv7cJFFUJ1C4pS-oE`. Accessed: 2017-11-19.

[19] D. Schuller, A. Polyvyanyy, L. García-Bañuelos, and S. Schulte, "Optimization of complex qos-aware service compositions," in *Proceedings of the 9th International Conference on Service-Oriented Computing*, ICSOC'11, (Berlin, Heidelberg), pp. 452–466, Springer-Verlag, 2011.

[20] "Architectural styles and the design of network-based software architectures." `https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation_2up.pdf`. Accessed: 2017-12-14.

[21] A. Klein and W. Lehner, "Representing data quality in sensor data streaming environments," *J. Data and Information Quality*, vol. 1, pp. 10:1–10:28, Sept. 2009.

[22] A. Sarimbekov, "Comparison of instrumentation techniques for dynamic program analysis on the java virtual machine," in *Proceedings of the 12th Annual International Conference Companion on Aspect-oriented Software Development*, AOSD '13 Companion, (New York, NY, USA), pp. 31–32, ACM, 2013.

[23] A. Nusayr, "Aop as a formal framework for runtime monitoring," in *Proceedings of the 2008 Foundations of Software Engineering Doctoral Symposium*, FSEDS '08, (New York, NY, USA), pp. 25–28, ACM, 2008.

[24] A. Nusayr and J. Cook, "Using aop for detailed runtime monitoring instrumentation," in *Proceedings of the Seventh International Workshop on Dynamic Analysis*, WODA '09, (New York, NY, USA), pp. 8–14, ACM, 2009.

[25] D. H. Le, H. L. Truong, and S. Dustdar, "Managing on-demand sensing resources in iot cloud systems," in *2016 IEEE International Conference on Mobile Services (MS)*, pp. 65–72, June 2016.

[26] "Smart contracts: Building blocks for digital markets." `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html`. Accessed: 2017-12-16.

[27] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[28] A. Dorri, S. S. Kanhere, and R. Jurdak, "Towards an optimized blockchain for iot," in *Proceedings of the Second International Conference on Internet-of-Things Design and Implementation*, IoTDI '17, (New York, NY, USA), pp. 173–178, ACM, 2017.

[29] M. Samaniego and R. Deters, "Using blockchain to push software-defined iot components onto edge hosts," in *Proceedings of the International Conference on Big Data and Advanced Wireless Technologies*, BDAW '16, (New York, NY, USA), pp. 58:1–58:9, ACM, 2016.

[30] P. H. Phung, H. L. Truong, and D. T. Yasoju, "P4sinc - an execution policy framework for iot services in the edge," in *2017 IEEE International Congress on Internet of Things (ICIOT)*, pp. 137–142, June 2017.

[31] R. Neisse, G. Steri, and G. Baldini, "Enforcement of security policy rules for the internet of things," in *2014 IEEE 10th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pp. 165–172, Oct 2014.

[32] "Mqtt message queuing telemetry transport)." `http://mqtt.org`. Accessed: 2018-13-01.

[33] C. Dsouza, G. J. Ahn, and M. Taguinod, "Policy-driven security management for fog computing: Preliminary framework and a case study," in *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, pp. 16–23, Aug 2014.

[34] S. Gusmeroli, S. Piccione, and D. Rotondi, "Iot access control issues: A capability based approach," in *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pp. 787–792, July 2012.

[35] O. Adinolfi, R. Cristaldi, L. Coppolino, and L. Romano, "Qos-monaas: A portable architecture for qos monitoring in the cloud," in *2012 Eighth International Conference on Signal Image Technology and Internet Based Systems*, pp. 527–532, Nov 2012.

[36] S. Guth, B. Simon, and U. Zdun, "A contract and rights management framework design for interacting brokers," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pp. 10 pp.–, Jan 2003.

[37] "Odrl community group." `https://www.w3.org/community/odrl`. Accessed: 2017-12-16.

[38] F. B. Balint and H. L. Truong, "On supporting contract-aware iot dataspace services," in *2017 5th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*, pp. 117–124, April 2017.

[39] C. Müller, H. L. Truong, P. Fernandez, G. Copil, A. Ruiz-Cortés, and S. Dustdar, "An elasticity-aware governance platform for cloud service delivery," in *2016 IEEE International Conference on Services Computing (SCC)*, pp. 74–81, June 2016.

[40] S. Nastic, S. Sehic, D. H. Le, H. L. Truong, and S. Dustdar, "Provisioning software-defined iot cloud systems," in *2014 International Conference on Future Internet of Things and Cloud*, pp. 288–295, Aug 2014.

[41] "Home - dfrc ag." `http://www.dfrc.ch`. Accessed: 2017-11-19.

[42] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, pp. 24–31, Apr 2015.

[43] "Rhino." `https://www.json.org`. Accessed: 2017-11-27.

[44] D. Le, H. L. Truong, G. Copil, S. Nastic, and S. Dustdar, "SALSA: A framework for dynamic configuration of cloud services," in *IEEE 6th International Conference on Cloud Computing Technology and Science, CloudCom 2014, Singapore, December 15-18, 2014*, pp. 146–153, 2014.

[45] "Ecmascript® 2017 language specification." `https://www.ecma-international.org/publications/standards/Ecma-262.htm`. Accessed: 2017-11-27.

[46] "Eclipse aspectj." https://www.eclipse.org/aspectj. Accessed: 2017-11-27.

[47] "Java is the go to language for iot applications)." https://jaxenter.com/java-is-the-go-to-language-for-iot-applications-127844.html. Accessed: 2018-09-01.

[48] "Creating the open source building blocks for iot)." https://www.w3.org/WoT/IG/wiki/images/5/59/Eclipse-IoT-W3C-WoT.pdf. Accessed: 2018-09-01.

[49] P. R. Krishna and K. Karlapalem, "Electronic contracts," *IEEE Internet Computing*, vol. 12, no. 4, pp. 60–68, 2008.

[50] S. Geisler, C. Quix, S. Weber, and M. Jarke, "Ontology-based data quality management for data streams," *J. Data and Information Quality*, vol. 7, pp. 18:1–18:34, Oct. 2016.

[51] F. Rosenberg, P. Leitner, A. Michlmayr, P. Celikovic, and S. Dustdar, "Towards composition as a service - a quality of service driven approach," in *2009 IEEE 25th International Conference on Data Engineering*, pp. 1733–1740, March 2009.

[52] "Qos-aware composition of adaptive service-oriented systems." https://www.google.at/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0ahUKEwiUnaG8gP3XAhVQ4KQKHcqBCEoQFggtMAA&url=http%3A%2F%2Fwww.infosys.tuwien.ac.at%2FStaff%2Fsd%2Fphd-thesis%2FPhDthesis_FlorianRosenberg.pdf&usg=AOvVaw14v5Aq0Zr2j_Z1ywkgmG_s. Accessed: 2017-11-19.

[53] "Board briefing on it governance." https://www.isaca.org/restricted/Documents/26904_Board_Briefing_final.pdf. Accessed: 2017-12-01.

[54] "Ethereum blockchain app platform)." https://www.ethereum.org. Accessed: 2018-13-01.

[55] "Etherscan, the ethereum block explorer)." https://etherscan.io. Accessed: 2018-13-01.

[56] "Your ethereum swiss army knife)." http://truffleframework.com. Accessed: 2018-13-01.

[57] "Github iotcloudsamples)." https://github.com/rdsea/IoTCloudSamples/tree/master/data/bts. Accessed: 2018-13-01.