Universität Stuttgart

A model-based approach for data processing in IoT environments

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Ana Cristina Franco da Silva

aus Manaus / Brasilien

Hauptberichter: Prof. Dr. -Ing. habil. Bernhard Mitschang

Mitberichter: Prof. Dr. Marco Aiello

Tag der mündlichen Prüfung: 04.11.2020

Institut für Parallele und Verteilte Systeme

2020

Contents

Acronyms

API Application Programming Interface

BPEL Business Process Execution Language

BPMN Business Model and Notation

CEP Complex Event Processing

CSAR Cloud Service Archive

dDSPM deployed Data Stream Processing Model

DBMS Database Management System

DSMS Data Stream Management System

DSP Data Stream Processing

DSPM Data Stream Processing Model

ETL Extract, Transform, and Load

GPIO General-purpose Input/Output

HTTP Hypertext Transfer Protocol

HVAC Heating, Ventilation and Air Conditioning

IEEE Institute of Electrical and Electronics Engineers

IoT Internet of Things

IoTEM IoT Environment Model

IT Information Technology

JAR Java Archive

JSON JavaScript Object Notation

M2M Machine-to-Machine

MBP Multi-purpose Binding and Provisioning Platform

MQTT Message Queuing Telemetry Transport

OASIS Organization for the Advancement of Structured Information Standards

QoS Quality of Service

RAM Random Access Memory

REST Representational State Transfer

RFID Radio Frequency Identification

RMP Resource Management Platform

SOA Service-oriented Architecture

SSH Secure Shell

SQL Structured Query Language

TDLIoT Topic Description Language for the Internet of Things

TOSCA Topology and Orchestration Specification for Cloud Applications

UDDI Universal Description, Discovery and Integration

UI User Interface

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML Extensible Markup Language

WAR Web Archive

WSN Wireless Sensor Networks

Zusammenfassung

Die heutigen Fortschritte in den Bereichen Sensor-Technologie, Netzwerke und Datenverarbeitung haben die Vision des Internet der Dinge mehr und mehr zu einer Realität im alltäglichen Leben gemacht. Dabei ermöglicht das Internet der Dinge die Entwicklung von anspruchsvollen Anwendungen für IoT-Umgebungen, wie Smart Cities, Smart Homes oder Smart Factories. Durch kontinuierliche Sensormessungen sowie hochfrequentem Datenaustausch zwischen sogenannten IoT-Objekten (z.B. IoT-Geräte), nehmen die Daten in IoT-Umgebungen die Form von Datenströmen an. Mit dieser immer größer werdenden Datenmenge, die kontinuierlich als Strom verarbeitet werden muss, treten jedoch einige Herausforderungen auf, die für ein effizientes Verarbeiten von IoT-Daten gelöst werden müssen. Beispielsweise stellt sich die Frage wie IoT-Daten verarbeitet werden können damit einerseits relevante Informationen gewonnen werden können und andererseits die Reaktivität der IoT-Applikationen nicht beeinträchtigt wird. Des Weiteren müssen verschiedene funktionale und nicht-funktionale Anforderungen der IoT-Applikationen durch die Datenverarbeitung erfüllt werden. In dieser Doktorarbeit wird ein neuer holistischer Ansatz vorgestellt, um strombasiere Daten durch IoT-Anwendungen zu verarbeiten. Der Fokus liegt dabei auf der effizienten Platzierung von Datenvearbeitungsoperatoren der datenstrombasierten Anwendungen auf heterogene, verteilte und dynamische

IoT-Umgebungen. Im Gegensatz zu bestehenden Ansätzen im Bereich der Platzierung von Operatoren werden in dem in dieser Arbeit vorgestellten Ansatz auch zusätzliche Anforderungen beachtet, die sich speziell auf die Eigenschaften des IoT beziehen. Des Weiteren werden auch nicht-funktionale und nutzerspezifische Anforderungen beachtet. Diese Doktorarbeit stützt sich auf verschiedene Informationsmodelle und Techniken, um Operatoren zu platzieren, so dass der gesamte Lebenszyklus von IoT-Umgebungen und datenstrombasierten Anwendungen einfach verwaltet werden kann. IoT-Umgebungen und ihre Fähigkeiten zur Datenverarbeitung werden durch sogenannte IoT Environment Models beschrieben (IoTEM). Analog wird die Geschäftslogik der IoT-Applikationen sowie deren Anforderungen durch ein Informationsmodell, genannt Data Stream Processing Model (DSPM), beschrieben. Basierend auf diesen Informationsmodellen bestimmen Algorithmen eine bestmögliche Platzierung von Operatoren auf die IoT-Objekte der IoT-Umgebung, so dass die vorher definierten Anforderungen der Anwendungen und Fähigkeiten der IoT-Umgebung zusammenpassen. Dabei ist es bei diesem Ansatz das Hauptziel die IoT-Daten so nah wie möglich an den Datenquellen zu verarbeiten, so dass Cloud-Infrastrukturen nur im Falle von Ressourcenmangel der IoT-Umgebung zum Einsatz kommen. Die simultane Ausführung der Datenverarbeitung in der IoT-Umgebung und in der Cloud wird allgemein als Fog-Computing bezeichnet. Durch die Konzepte dieser Doktorarbeit kann die Datenverarbeitung von IoT-Applikationen auf spezifische Szenarien zugeschnitten werden, wobei die charakteristischen Anforderungen der Domänen sowie der Anwender der IoT-Applikation berücksichtigt werden. Sobald eine mögliche Platzierung gefunden wurde, werden die Operatoren auf die zugehörigen IoT-Objekte installiert. Hierfür können etablierte Standards wie TOSCA zum Einsatz kommen. Nach der Installation der Operatoren ist die IoT-Anwendung lauffähig. Während der Laufzeit der IoT-Anwendung wird diese kontinuierlich überwacht, um mögliche Störungen zu bemerken, die während der Datenverarbeitung auftreten. Die Ansätze dieser Doktorarbeit werden unterstützt durch die Multi-Purpose Binding and Provisioning Platform, eine Open-Source IoT-Plattform, die als Proof-of-Concept für die Konzepte dieser Doktorarbeit implementiert wurde.

Abstract

The recent advances in several areas, including sensor technologies, networking, and data processing, have enabled the Internet of Things (IoT) vision to become more and more a reality every day. As a consequence of these advances, the IoT of today allows the development of sophisticated applications for IoT environments, such as smart cities, smart homes, or smart factories. Due to continuous sensor measurements and frequent data exchange among so-called IoT objects, the data generated within an IoT environment incorporate the form of data streams. With this increasing amount of data to be continuously processed, several challenges arise while aiming at an efficient processing of IoT data. For instance, how IoT data processing can be realized, so that meaningful information can be derived without affecting the reactiveness of IoT applications. Furthermore, how different functional, non-functional, and user-defined requirements of IoT applications can be satisfied by the IoT data processing. In this PhD thesis, a new holistic approach for processing data stream-based applications within IoT environments is presented. Its focus lies on efficient placement of operators of data stream applications onto heterogeneous, distributed, dynamic IoT environments. In contrast to state-of-the-art operator placement, this approach takes into consideration additional requirements introduced by the peculiar characteristics of the Internet of Things. Furthermore, non-

functional and user-defined requirements are also taken into consideration. This PhD thesis is supported by different informational models and operator placement techniques, so that the entire life cycle of IoT environments and data stream-based applications can be easily managed. IoT environments and their processing capabilities are described by IoT environment models (IoTEM). Likewise, the business logic of IoT applications and their requirements are defined by data stream processing models (DSPM). Based on these informational models, several algorithms determine feasible placements of processing operators onto IoT objects of IoT environments, so that the aforementioned requirements and capabilities are matched. In this approach, one of the main goals is to process IoT data as near to data sources as possible, so that cloud infrastructures are employed only in cases where IoT environments do not offer sufficient processing resources for the IoT application. The execution of data processing on both IoT environments and cloud infrastructures is commonly known as fog computing. Through the approach of this PhD thesis, data processing of IoT applications can be tailored to particular use cases, supporting the specific requirements of the domains, and furthermore, of IoT application users. Once feasible placements are determined, processing operators are then deployed onto corresponding IoT objects using standards, such as TOSCA, and the IoT application is considered up and running. Finally, the IoT environment is continuously monitored in order to recognize and react to disturbances affecting the data processing of deployed IoT applications. The approach of this PhD thesis is supported by the Multi-purpose Binding and Provisioning Platform (MBP), an open-source IoT platform, which has been developed as a proof-of-concept of the contributions of this PhD thesis.

Acknowledgements

First of all, I would like to thank my PhD supervisor Prof. Dr. Bernhard Mitschang for his excellent ongoing support and his confidence in my work. I would also like to thank Prof. Dr. Marco Aiello, who has agreed to be the second examiner. Also many thanks to the other members of the examination board, chair Prof. Dr. Stefan Wagner and co-examiner Prof. Dr. Miriam Mehl. My thanks also go to all my colleagues in the AS department of the IPVS, with whom I was able to hold discussions that enriched the results of this PhD thesis. In particular, I would like to thank my post-doc Dr. Pascal Hirmer, who was always able to provide me with valuable feedback. Furthermore, I would like to thank all the colleagues with whom I wrote scientific publications as well as the students who supported me in creating prototypes of my concepts. Last but not least, I would like to thank my friends and family, who always supported me in many ways.

Chapter 1

Introduction

> "The Internet of Things has the potential to change the world, just as the Internet did. Maybe even more so."
> – Kevin Ashton (2009)

In this PhD thesis, a new approach for processing data stream-based applications within IoT environments is introduced. Its focus lies on efficient placement of operators of data stream applications onto heterogeneous, dynamic IoT environments. In contrast to state-of-the-art operator placement, this approach takes into consideration additional requirements introduced by the peculiar characteristics of the Internet of Things. Furthermore, non-functional and user-defined requirements are also taken into consideration. This PhD thesis is supported by different informational models and operator placement techniques, so that the entire life cycle of IoT environments and data stream-based applications can be easily managed.

In this introductory chapter, the motivation of this PhD thesis is described in Section 1.1. The research questions and goals are presented in Section 1.2. Finally, Section 1.3 provides a summary of the contributions and Section 1.4 describes the overall structure of this PhD thesis.

1.1 Motivation

The Internet of Things (IoT) envisions the pervasive presence of heterogeneous devices in enclosed environments [GBMP13; LL15]. These devices are equipped with sensors and actuators. Furthermore, they share a common network and, thereby, are able to exchange information among each other to reach common goals [VF13].

In this thesis, the term *IoT object* is used as the generic term for devices, sensors, and actuators. The IoT and the characteristics of IoT objects enable the development of sophisticated applications for *IoT environments*, such as smart cities [ARJ19], smart homes [GBMP13], or smart factories [TCZN15].

Due to continuous sensor measurements and frequent data exchange among IoT objects, the data generated within an IoT environment incorporates the form of *data streams*. These data are not persistent but rather arrive for processing in multiple, continuous, rapid, time-varying streams [BBD+02].

In contrast to traditional database systems and to traditional batch processing systems, such as Hadoop [MW15], the processing and management of data streams lead to major challenges, specially in the IoT, where large amounts of data streams are continuously produced. To respond to these challenges, well-established techniques can be employed, such as *data stream processing* [CM12] and *complex event processing* [Luc01].

A further challenge of processing data streams is the suitable location of data processing. To derive meaningful information from data streams, one common approach is to transfer IoT data to cloud infrastructures, where the processing is mainly centralized. This cloud-only processing of IoT data is depicted in Figure 1.1 on the left. The advantage of this approach is that the required resources to process the data can be provisioned on-demand as needed [Ran+18]. With a rising amount of data, the underlying infrastructure can be easily scaled vertically or horizontally. However, this approach has an impact on an important requirement of reactive IoT applications, namely, the timely processing of data streams [Ope17]. Sending IoT data to cloud infrastructures increases latency and network traffic, and might

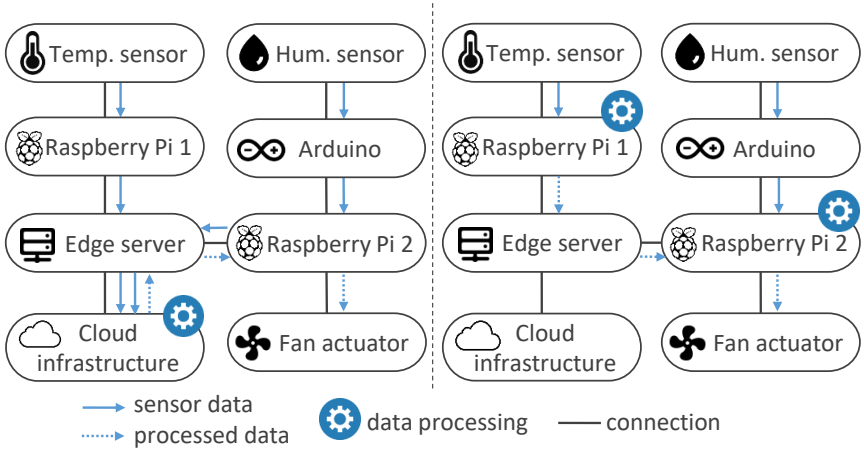


Figure 1.1: Processing data streams in IoT environments. Left: cloud-only IoT data processing, right: decentralized IoT data processing

provoke delays before and after the data stream processing.

Therefore, to achieve timely data processing, an alternative approach is required, which avoids sending all IoT data to cloud infrastructures. More precisely, data should be processed as close to the sources as possible by exploiting first the already provided computing infrastructure in IoT environments. In this approach, which is depicted in Figure 1.1 on the right, the processing of data streams needs to be distributed among different processing nodes for execution. This is commonly known as the *operator placement problem* [CM12], which aims to find an optimal placement for data processing operators onto different processing nodes distributed over a network. Over the last decades, many solutions have been proposed that tackle the operator placement problem. Many of them decide the placement location based on the fulfillment of Quality of Service (QoS) requirements, such as latency and bandwidth.

However, with the increasingly upcoming of IoT applications, further aspects have emerged that additionally need to be taken into consideration when realizing operator placement within IoT environments. These aspects

include, for example, the heterogeneity of processing nodes due to the existent different types of IoT objects. Moreover, IoT environments can also encompass many different types of networks and technologies, whose heterogeneity also needs to be considered.

Currently, existent solutions lack supporting such additional requirements introduced by the IoT domain. Furthermore, they normally do not take into consideration non-functional and user-defined requirements for IoT applications, such as data privacy and anonymization, security, and interoperability.

Therefore, in this PhD thesis, an approach for the placement of data stream processing operators onto IoT environments is presented, which also takes into consideration the characteristics of the IoT domain, and furthermore, non-functional and user-defined requirements.

To realize this, it is required to know which IoT objects are available in an IoT environment and which resources they offer, i.e., their *capabilities*. Furthermore, it is also required to know how IoT data should be processed, i.e., the business logic of IoT applications, and the *requirements* of these IoT applications. These information can be described by different models, such as *IoT environment models* (cf. Table 4.1) and *data stream processing models* [Hir18], which enables a clear separation of concerns. Consequently, such informational models are crucial on the placement of operators in IoT environments.

In summary, in this PhD thesis, IoT environments and their capabilities are described by IoT environment models (IoTEM). Likewise, the business logic of IoT applications and their requirements are defined by data stream processing models (DSPM). Based on these informational models, several algorithms determine feasible placements of processing operators onto IoT objects of IoT environments, so that the aforementioned requirements and capabilities are matched.

In this approach, one of the main goals is to process IoT data as near to data sources as possible, so that cloud infrastructures are employed only in cases where IoT environments do not offer sufficient processing resources or do not provide the required capabilities for the IoT application. The execution of data processing on both IoT environments and cloud infrastructures is

commonly also known as *fog computing* [Ope17].

Through the approach of this PhD thesis, data processing of IoT applications can be tailored to particular use cases, supporting the specific requirements of the domains, and furthermore, of IoT application users. Once feasible placements are determined, processing operators are then deployed onto corresponding IoT objects using standards, such as TOSCA [OAS13], and the IoT application is considered up and running. Finally, the IoT environment is continuously monitored in order to recognize and react to disturbances affecting the data processing of deployed IoT applications. All concepts of this PhD thesis are based on established standards or de-facto standards in order to ensure their long lasting applicability and future-proofness.

1.2 Research questions and goals

In Section 1.1, the issues that this PhD thesis aims to tackle were motivated. These issues correspond mainly to the current absence of solutions to realize operator placement tailored to IoT environments that should also consider: (i) additional, diverse requirements of IoT applications and IoT environments, and (ii) user-defined and non-functional requirements during the operator placement decision. The described issues lead to the following research questions (RQ):

RQ1: How can different functional, non-functional, and user-defined requirements be taken into consideration during the realization of operator placement onto IoT environments?

RQ2: How can it be decided which IoT objects are suitable to execute which data stream processing operators?

RQ3: How can data stream processing operators be efficiently deployed onto heterogeneous, dynamic IoT environments?

RQ4: How can the processing of IoT applications within IoT environments be guaranteed throughout the entire life cycle of IoT applications?

The aforementioned research questions will be addressed in the scope of this PhD thesis and lead to the following goals (G):

G1: Timely, efficient processing of IoT data within IoT environments. This goal aims to realize the processing of data streams primarily by the computing resources of IoT objects, which are available within an IoT environment. A timely processing will be achieved by processing IoT data as close to the sources as possible, so that the provided computing infrastructure in IoT environments is exploited first. However, if an IoT environment does not provide enough computing resources, cloud infrastructures will be also employed to support the processing.

To achieve this goal, operator placement will be realized based on the characteristics of the IoT domain, i.e., IoT environments and IoT applications. Furthermore, this goal also aims to assure that the data processing of IoT applications stays *correct*, which is defined as follows:

Definition 1.1 (Data processing correctness)
The data processing of an IoT application is correct if the processing is deployed and running as defined and expected by the domain analyst.

By achieving this goal through a timely, efficient processing, IoT applications and users highly benefit from the consequent improved quality of the processing results.

G2: User-friendly, easy modeling of IoT environments. This goal aims to support users by providing means for easy modeling of IoT environments. Based on the characteristics of the IoT domain, it should be possible to model heterogeneous IoT objects, their interconnections, and furthermore, the capabilities of IoT objects and connections.

This goal will be accomplished by providing a graphical modeling tool with good usability accomplished through separation of concerns between technical knowledge and high-level domain knowledge. Therefore, IoT environments should be modeled by so-called *domain experts*, which have technical knowledge about the IoT environment,

e.g., information about the computing capabilities of diverse IoT objects and how to interact with these IoT objects through their provided interfaces.

G3: Modeling of data stream processing supporting IoT requirements. This goal aims to support users that have only high-level domain knowledge, by providing them with means to model the business logic of data stream-based applications including the requirements introduced by the IoT domain. These requirements should be taken into consideration, in a further step, in order to automatically decide which IoT objects are suitable to execute the processing.

This goal will be accomplished by providing a further graphical modeling tool, in which only high-level domain knowledge is required, i.e., technical knowledge about sensors and actuators within an IoT environment are abstracted as data sources and sinks. Therefore, data stream processing should be modeled by so-called *domain analysts*, which have domain knowledge about the data generated in the IoT environment and how this data need to be processed. In this way, domain analysts will be able to describe the processing logic of IoT applications for domain-specific use cases.

G4: Requirements-based placement of processing operators onto IoT environments. This goal aims to achieve operator placement for IoT applications, so that the diverse requirements introduced by the IoT domain are also supported. Furthermore, user-defined requirements for IoT applications, e.g., data privacy and anonymization, security, and interoperability, will be also taken into consideration.

This goal will be accomplished by employing informational models for IoT environments and data stream processing to decide how operators should be placed onto IoT environments considering the requirements of operators to be fulfilled by the capabilities of IoT objects.

G5: Efficient deployment of operators onto heterogeneous, dynamic IoT environments. This goal aims at the efficient deployment of data

stream processing operators onto heterogeneous IoT objects existent in IoT environments.

To achieve this goal, automatic deployment approaches should be employed that are able to deal with the heterogeneous and dynamic nature of IoT environments. Furthermore, manual actions and hardware configuration (e.g., plugging in sensors) should be also supported by the deployment, since manual actions are common tasks to be realized during setup and deployment of IoT environments.

G6: Timely recognition of disturbances in IoT environments. This goal aims to recognize disturbances affecting the processing of deployed IoT applications onto IoT environments as soon as possible. Such disturbances can be caused by changes in the IoT environment, e.g., a faulty device, or changes in the IoT application, e.g., a new user-defined requirement is introduced that requires data encryption. A timely recognition is important in order to be able to eliminate such disturbances as soon as possible and, therefore, to assure that the data processing of IoT applications stays correct (cf. Definition 1.1).

This goal will be achieved by using well-established techniques, such as complex event processing, to continuously monitor IoT environments and their corresponding IoT data.

1.3 Contributions summary

This section provides a compact overview on the contributions (C) of this PhD thesis, which address the aforementioned research questions (RQ) and goals (G). A detailed overview of these contributions is provided in Chapter 3, and subsequently, each contribution is explained in detail in Chapters 4 to 7.

The contributions are based on established standards in order to ensure their long lasting applicability. They have been published in several journals, national and international conferences. An overview of the publications can be found at the end of this document (p. 159 ff.). In addition, software

artifacts developed as part of this PhD thesis are available as open-source projects on GitHub[1,2,3,4].

This PhD thesis provides the following four main contributions:

C1: Modeling of IoT environments and data stream processing. This contribution provides the *IoT environment model (IoTEM)* to describe IoT objects and *capabilities* commonly found in IoT environments. Furthermore, C1 also provides the *data stream processing model (DSPM)* to describe the data processing logic of IoT applications. This model describes data sources, data sinks, processing operators, and the data flow among them. Furthermore, it describes *requirements* of processing operators for IoT objects.

C2: Mapping of DSPMs onto IoTEMs. This contribution provides the mapping of data stream processing models (DSPMs) onto IoT environment models (IoTEMs), considering the diverse requirements of the processing operators and the capabilities of the IoT objects. It contains two approaches to realize the mapping: (i) an automatic approach, in which a *mapping plan* containing at least one set of IoT objects fulfilling the requirements of the DSPM is automatically generated, and (ii) a manual approach, in which domain analysts decide themselves which IoT objects should execute which operators. For this, domain analysts create a mapping plan manually. To realize the mapping, this contribution provides the IoT platform called *Multi-purpose Binding and Provisioning Platform (MBP)* [FHS+20], which also enables the management of IoTEMs and DSPMs provided in contribution C1.

C3: Deployment of operators onto IoT environments. This contribution provides the deployment of processing operators onto IoT objects available in IoT environments. The concepts of this contribution uses the Topology and Orchestration Specification for Cloud Applications

[1] https://github.com/IPVS-AS/MBP
[2] https://github.com/IPVS-AS/MBP-Docker
[3] https://github.com/IPVS-AS/MBP2Go
[4] https://github.com/IPVS-AS/TDLIoT

(TOSCA) standard [OAS13] approved by the Organization for the Advancement of Structured Information Standards (OASIS). This contribution provides two approaches to realize the deployment: (i) an *automatic deployment approach*, in which the operators of a DSPM are deployed automatically based on the mapping plan generated in contribution C2, and (ii) a *semi-automatic deployment approach*, which supports manual actions, called *human tasks* [OAS10].

C4: Monitoring of deployed DSPMs. This contribution provides the means to monitor deployed data stream processing models (dDSPM), in order to recognize disturbances (i. e., concept drifts [WK96]) affecting their data processing. To recognize such disturbances, the MBP continuously monitors IoT objects in IoT environments and dDSPMs. This monitoring is realized using complex event processing (CEP) techniques [Luc01], which are well-established for the processing of data streams to timely recognize situations (i.e., changes requiring correcting actions) [BD15; BK09; FHWM16].

1.4 Structure of this thesis

This PhD thesis is further structured as follows. In Chapter 2, the basic concepts for this PhD thesis are introduced. It comprises an overview on the Internet of Things (IoT), data stream processing, complex event processing (CEP), and the TOSCA standard.

In Chapter 3, the overview of this PhD thesis is provided, in which the contributions are introduced, and furthermore, the methodical approach and overall architecture are presented. Moreover, each contribution is explained in detail in Chapters 4 to 7. For each contribution, the corresponding concepts and architectural components, as well as related work are described.

In Chapter 8, the Multi-purpose Binding and Provisioning Platform (MBP) is described, which has been developed as a proof-of-concept of the contributions of this PhD thesis. Furthermore, the overall architecture of this thesis is evaluated against the IEEE standard 1934-2018 [Ope17], which provides

a reference architecture for fog computing platforms. Finally, in Chapter 9, the results of this PhD thesis are summarized and assessed, and future work is discussed.

Chapter 2
Background

In this chapter, the basic concepts for this thesis are introduced. First, Section 2.1 presents the main concepts of the Internet of Things. Second, Section 2.2 explains data stream processing and complex event processing. Third, Section 2.3 gives an overview on the operator placement problem. Finally, the TOSCA standard is described in Section 2.4.

2.1 Internet of Things

The term *Internet the Things (IoT)* has first surfaced at the end of the 90s, with Ashton's idea [Ash+09] of letting computers know everything about things, however, based on data gathered autonomously. His idea was to enhance computers with radio frequency identification (RFID) and sensor technologies to gather information, observe and identify an environment without the need of human assistance. In this way, it would be possible to track and monitor things in order to reduce costs, and furthermore, to know when things needed to be repaired or replaced.

Vermesan et al. [VFG+13] define the IoT as a paradigm in which a variety of things/objects is pervasively present in an environment. In this

so-called IoT environment, these things are connected wireless or wired, uniquely identifiable, and able to cooperate with each other in order to reach common goals. To make this paradigm possible, the IoT benefits from several enabling technologies originated from many different research fields, such as machine-to-machine (M2M) communication, RFID, wireless sensor networks (WSN), semantic data, cloud computing, and service-oriented architectures (SOA) [AIM10].

Nowadays, many applications for the IoT have been developed in a variety of domains, such as healthcare [SSF17], environment monitoring, or smart factories [ARJ19]. Furthermore, there are several commercial IoT solutions, partially from highly regarded companies available on the market [DEDP15], such as AllJoyn [Ope16], Apple HomeKit [App14], Google Cloud IoT [Goo17], IoTivity [Lin17], Samsung SmartThings [Sma12] and Thread [Thr14], which are mainly designed for the domain smart home.

2.2 Data stream processing and complex event processing

The data generated within IoT environments are normally delivered for processing in multiple, continuous, rapid, time-varying *data streams* [BBD+02]. An important requirement of IoT applications is the ability to process this kind of data in a continuous and timely fashion [CM12], enabling IoT applications to be scalable, dynamic, and reactive [BK09; CM12]. By employing traditional database management systems (DBMS) using extract-transform-load (ETL) processes, which require data to be stored and indexed for processing, the requirements of IoT applications cannot, however, be met. Therefore, several approaches have emerged that were specifically designed to process data as streams based on a set of processing rules [CM12]. Well-established approaches for this kind of processing are *data stream processing* [BBD+02] and *complex event processing* [Luc01].

Data stream processing, which is an evolution of the data processing in DBMS, runs continuous queries on incoming data streams [CM12; Luc19]. While DBMSs work with persistent data which are not updated frequently,

data stream management systems (DSMS) work on transient data, i.e., data that is continuously updated. Furthermore, queries on DBMSs run once and return complete answers, while DSMSs run queries continuously and provide updated answers upon the arrival of new data. Normally, DSMSs process data streams through a sequence of transformations based on SQL operators, such as selection, aggregation, or join, defined by relational algebra [CM12].

On the other hand, complex event processing encompasses a set of principles and techniques to analyze sets of events partially ordered by time as these events arrive. That is, CEP provides the means to process sets of interrelated events in a continuous and timely fashion [Luc19]. An *event* is defined by Etzion and Niblett [EN10] as a programming entity representing an occurrence, i.e., something that has happened, in a system or domain. A single event might contain a portion of information that is only meaningful if considered with other related events. Events are normally delivered for processing in patterns, however, they can be mixed with other unrelated events. An important characteristic of CEP is the ability to detect patterns (i.e., relationships) among events [BK09; Luc11]. For instance, CEP enables the definition of constraints of a system (e.g., an IoT environment) as event patterns. The output of the system under observation can then be monitored in real-time for violations of those constraints [Luc19]. These constraints can be, for example, occurrences that might need a correcting action, i.e., situations [Luc01], such as a machine breakdown.

The decision which processing approach should be used depends on the application specific goals and what problems it aims to solve [Luc19]. If the application needs to analyze a stream of data or events, ordered by time, data stream processing should be used. The focus of stream processing lies on high-speed querying of data in streams and applying mathematical algorithms to this data. There are currently many free, open-source data (or event) stream processing frameworks, such as Apache Flink [CKE+15], Apache Heron [KBF+15], Apache Samza [NPP+17], Apache Storm [TTS+14], or Esper [Esp06a]. On the other hand, if the application needs to analyze a set of unordered events, CEP should be applied. CEP focuses on extracting information from sets of events created, for

example, in IT and business systems. CEP also provides data analysis, but focuses more on patterns of events, and abstracting information in these patterns. Many approaches providing CEP functionalities have been developed, such as Esper [Esp06a], flowthings.io [flo10], FIWARE CEP GE [FIW16], Odysseus [Uni07], or WSO2 Siddhi [SGL+11; WSO13].

2.3 Operator placement problem

The processing of data streams can be realized through a centralized instance or it can be distributed among different processing nodes for execution. The distributed processing, however, implicates a further challenge besides the timely processing, known as the *operator placement* problem. This problem aims to find an optimal placement of either entire continuous queries or single operators onto a set of different processing nodes distributed across a network. The optimal placement is normally computed based on system-defined or user-defined cost functions, which aim to provide, for example, higher performance or better load distribution [CM12].

Lakshmanan et al. [LLS08] provide a taxonomy for operator placement and survey different operator placement strategies according to their specific goals for the placement. These strategies originate from research in academia and industry. Many distributed stream processing systems that provide operator placement are available, such as Apache Flink [CKE+15], Borealis [AAB+05], SBON [PLS+06] and SPADE [GAW+08]. Furthermore, Cugola and Margara [CM13] present several operator placement strategies for distributed complex event processing.

2.4 TOSCA

The cloud computing paradigm has recently emerged for hosting and delivering services over the Internet [LFWW16; ZCB10]. This paradigm has been increasingly employed together with the IoT paradigm, in order to provide IoT environments with properties, such as scalability and interoperability.

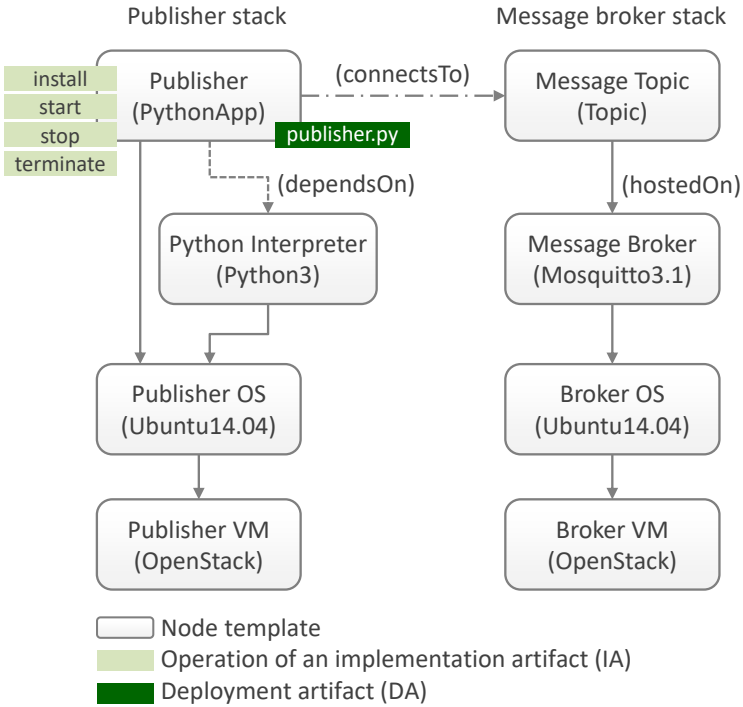


Figure 2.1: TOSCA topology model example of a publish-subscribe application (based on [FBK+16])

Cloud computing can enable a rapid setup and integration of new IoT objects and IoT applications, while maintaining low costs for the deployment of entire IoT environments [BDPP16].

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13] is an approved OASIS standard for the modeling, deployment, and management of cloud applications [BBKL14]. It can be divided in two main parts, (i) the topology model of an application, and (ii) the orchestration defining the steps for the deployment of this application.

The *topology model* describes the structure of the application, i.e., its software components, and furthermore, its platform and infrastructure.

Consequently, the concepts of the TOSCA standard unifies the paradigms software-as-a-service, platform-as-a-service, and infrastructure-as-a-service. A topology model, called *topology template*, is a graph composed of typed nodes, called *node templates*, and directed typed edges, called *relationship templates*. TOSCA is highly generic so that it enables the definition of arbitrary types to describe application components, called *node types*, and their dependencies, called *relationship types*.

In Figure 2.1, an exemplary topology model of an application using the topic-based publish-subscribe interaction model [EFGK03] is depicted. In this communication pattern, a data producer (i. e., publisher) publishes messages to a topic hosted on a message broker, which routes published messages to corresponding subscribers (i. e., a data consumer). The used graphical representation in TOSCA is based on the Visual Notation for TOSCA (Vino4TOSCA) [BBK+12].

The TOSCA topology model in Figure 2.1 contains two stacks: (i) the publisher stack on the left and (ii) the message broker stack on the right. The publisher stack is composed of four node types. It contains an *OpenStack* node type, which corresponds to the cloud platform providing a virtual machine for the publisher application. The *Ubuntu14.04* node type defines the operating system of the virtual machine. Furthermore, the *PythonApp* corresponds to the publisher's software component, which requires *Python3* to be installed on the virtual machine. In the message broker stack, the *Mosquitto3.1* node type corresponds to the message broker, which hosts the *Topic* to which the publisher can send messages. Finally, this topology model contains three relationship types. In the publisher stack, for example, the *PythonApp* node type *dependsOn* the *Python3* node type. Both of them are *hostedOn* the *Ubuntu14.04* node type. Furthermore, the PythonApp *connectsTo* the *Topic* node type.

The concrete implementation artifacts of components can be attached to node templates using *deployment artifacts* (DA), which are, for example, binary files, Shell or Python scripts. To install or manage a software component, management operations can be defined for the corresponding node type. The implementations of these operations can be provided using

implementation artifacts (IA), e.g., shell scripts to install the component. Furthermore, TOSCA defines a packaging format called *cloud service archive* (CSAR), which enables the grouping of all the above described entities in a self-contained manner.

The second part of the TOSCA specification deals with the *orchestration* of services, in order to execute the necessary steps to set up an application. This part of the TOSCA specification describes all actions necessary to provision, manage, and deprovision a cloud application based on its topology model.

TOSCA supports two approaches for application provisioning: (i) an imperative approach, and (ii) a declarative approach. The imperative approach requires defining so-called *build plans*, which describe the concrete order of steps that need to be conducted to set up the application components. That is, the orchestration is realized through these build plans, which invoke operations (i.e., implementation artifacts) to set up the individual components of the application following a certain order. More concretely, build plans can be created by employing workflows and processes technologies, such as BPEL [ACD+03] or BPMN [CT12].

On the other hand, the declarative approach only requires the definition of the topology model. In this case, the corresponding TOSCA runtime consequently provisions the application by itself. However, in the declarative approach, only components can be set up that are known to the corresponding runtime environment [BBK+14]. That is, this approach is not generic and only works for a specific set of components. A combination of the imperative and declarative approach by generating build plans automatically is provided by Breitenbücher et al. [BBK+16].

An example of a combined imperative and declarative TOSCA ecosystem is *OpenTOSCA* [BBH+13], which provides a TOSCA runtime environment and the corresponding graphical modeling tool *Winery* [KBBL13] for TOSCA topology templates.

Chapter 3

Thesis overview

This chapter gives an overview of this PhD thesis. A detailed overview of the contributions of this thesis is provided in Section 3.1. Each contribution is further explained in detail in Chapters 4 to 7. In Section 3.2, the overall methodical approach is presented. Finally, in Section 3.3, the resulting architecture and the corresponding employment of the methodical approach are described.

3.1 Contributions

The contributions presented in this section are based on established standards (e.g., MQTT, TOSCA, XML) in order to ensure their long lasting applicability. They have been published in journals, as well as on national and international conferences. An overview of the publications can be found at the end of this thesis (p. 159 ff.). In addition, software developed as part of this thesis have

been made available as open-source projects in GitHub[1,2,3,4].

This thesis provides the following four main contributions (C):

C1: Modeling of IoT environments and data stream processing. In order to decide how data should be processed within an IoT environment, knowledge about the available IoT objects in the environment and about the processing logic of an IoT application is required. For this purpose, this contribution provides the *IoT environment model (IoTEM)* to describe IoT objects commonly found in IoT environments. These IoT objects include both hardware objects (e.g., devices, sensors, actuators) and virtual objects (e.g., virtual machines). This model contains, for example, information about how to access IoT objects (e.g., IP address of a device) and about their *capabilities* (e.g., processing power, available storage of a device). Furthermore, this contribution also provides the *data stream processing model (DSPM)* to describe the data processing logic of an IoT application. This model describes data sources (e.g., sensors), data sinks (e.g., actuators), processing operators (e.g., data filter, aggregation), and the data flow among them. Furthermore, it describes *requirements* of processing operators for IoT objects, such as minimum required available memory.

C2: Mapping of DSPMs onto IoTEMs. In contribution C2, means are provided for the mapping of data stream processing models (DSPMs) onto IoT environment models (IoTEMs), considering the requirements of the processing operators and the capabilities of the IoT objects. This contribution provides two main approaches to realize the mapping: (i) an *automatic approach*, in which a *mapping plan* containing at least one set of IoT objects fulfilling the requirements of the DSPM is automatically generated, and (ii) a *manual approach*, in which domain analysts decide themselves which IoT objects should execute which operators. For this, domain analysts create a mapping plan manually.

[1] https://github.com/IPVS-AS/MBP
[2] https://github.com/IPVS-AS/MBP-Docker
[3] https://github.com/IPVS-AS/MBP2Go
[4] https://github.com/IPVS-AS/TDLIoT

To realize the mapping, this contribution provides the *Multi-purpose Binding and Provisioning Platform (MBP)*, which enables the management of IoTEMs and DSPMs provided in contribution C1. Furthermore, the MBP provides two mapping algorithms that match the overall requirements of the DSPM with the capabilities of the IoTEM. The manual mapping approach is recommended for small use cases, e.g., for smart homes, with a minor effort to manually choose IoT objects to execute operators. The automatic mapping approach is more suitable for larger use cases, which require a major effort to choose suitable IoT objects for the execution, e.g., in the smart factory domain, where a large amount of IoT objects are available.

C3: Deployment of operators onto IoT environments. Contribution C3 provides the deployment of processing operators onto IoT objects in IoT environments based on the results of contribution C2. The concepts of this contribution uses the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [OAS13] approved by the Organization for the Advancement of Structured Information Standards (OASIS). For contribution C3, the MBP provides two paradigms for the deployment of operators onto IoT objects: (i) an *automatic deployment*, in which the operators are deployed automatically based on the mapping plan generated in contribution C2, and (ii) a *semi-automatic deployment*, which is also based on the mapping plan generated in contribution C2, however, manual actions called *human tasks* (e.g., plugging in sensors) are required during or before the deployment of operators. In this case, the mapping plan is enhanced with corresponding human task definitions. Once the deployment of the operators is finished, the MPB creates and manages a running instance of the corresponding DSPM, which is called *deployed data stream processing model (dDSPM)*.

C4: Monitoring of deployed DSPMs. After the deployment of processing operators, the overall data stream processing of an IoT application is started. To assure that this processing stays correct (cf. Defini-

tion 1.1) as long as needed by the IoT application, the contribution C4 provides the means to recognize disturbances affecting the deployed data stream processing model (dDSPM). Such disturbances can occur through changes in the IoT environment (e.g., a faulty device) or changes in the DSPM (e.g., by adding a new requirement). To recognize such disturbances, the MBP continuously monitors IoT objects and dDSPMs. This monitoring is realized mainly using complex event processing (CEP) techniques [Luc01]. Such techniques are well-established and have been used for the continuous processing of large amounts of data, for example, to timely recognize critical situations (i.e., changes requiring correcting actions) [BD15; BK09; FHWM16].

The contributions of this thesis are applied through a methodical approach, which manages the entire life cycle of IoT environments and data stream-based IoT applications.

The life-cycle method is depicted in Figure 3.1. The contributions are highlighted by color. It consists of six main steps: ❶ creation of the IoT environment model (IoTEM), ❷ creation of the data stream processing model (DSPM), ❸ mapping of processing operators and IoT objects, ❹ deployment of processing operators onto IoT objects, ❺ recognition of disturbances affecting the data processing, and ❻ retirement of the data processing.

Two main roles are defined in this approach: the *domain expert*, which conducts step ❶ and part of step ❺, and the *domain analyst*, which conducts step ❷ and part of step ❻. The other steps are conducted in an automated fashion. Exceptions are the optional manual approaches in which the domain analyst also interacts in steps ❸ and ❹ (cf. contributions C2 and C3).

3.2 Methodical approach

Domain experts have technical knowledge about the hardware objects (i.e., devices, sensors, actuators), the virtual objects (i.e., virtual machines) and their network interconnections within an IoT environment. This technical knowledge comprises information about the *computing capabilities* (e.g.,

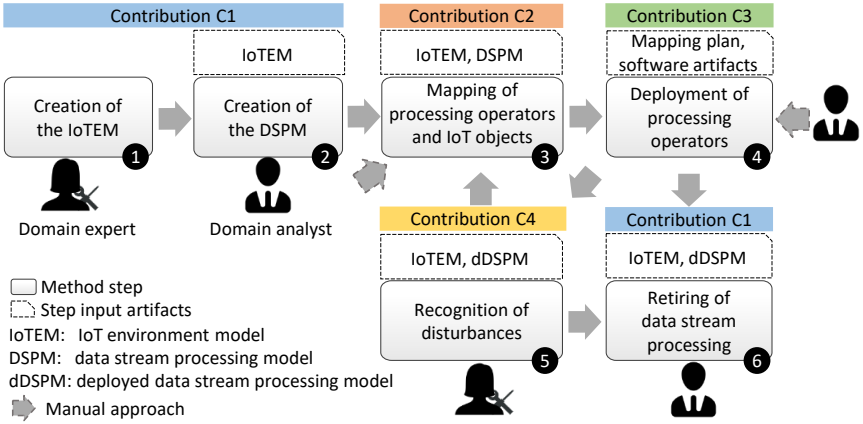


Figure 3.1: Life-cycle method for the contributions of this thesis

available main memory, connectivity, processing power) of IoT objects. Furthermore, domain experts have the knowledge about how to access these IoT objects to, for example, extract sensor data or send control commands to an actuator. Therefore, in step ❶, the main task of domain experts is the creation of IoTEMs, which are directed graphs containing IoT objects as nodes, and their network interconnections as edges. IoTEMs are based on ontologies [BEBT16], which associate semantics, and hence, reasoning. Created IoTEMs are then automatically registered in the MBP, which instantiates the digital counterparts of the modeled IoT objects.

Domain analysts on the other hand have domain knowledge about the processing of data generated within the IoT environment, i.e., they have the required knowledge to model different IoT applications for domain-specific use cases. In step ❷, the main task of domain analysts is the creation of DSPMs representing the processing logic of IoT applications. A DSPM is graph-based and contains data sources, data sinks, and processing operators as nodes, and the data flow connections among these nodes as edges. DSPMs are based on the pipes and filters design pattern [Meu95]. To the domain analysts, the sensors and actuators modeled in step ❶ are abstracted as data

sources and sinks, so that they only need to handle the modeling of the processing logic of an IoT application. Furthermore, such DSPMs contain *computing requirements* (e.g., minimum main memory, secure data storage) of processing operators of an IoT application. For this, knowledge about IoT applications and corresponding operators is required from domain analysts.

Based on the overall capabilities of the IoTEM and the requirements specified by the DSPM resulting from steps ❶ and ❷, the third step aims at automatically deciding on which IoT objects the operators should be deployed. This task is conducted by algorithms, which match requirements in the DSPM with capabilities of the IoTEM. The algorithms consider not only capabilities of IoT objects, but also capabilities of the network connections in between. This results in a *mapping plan*, which contains at least one set of available IoT objects, whose capabilities fulfill the requirements of the DSPM. One of the main goals of step ❸ is to place operators as close to the data sources as possible, in order to possibly reduce network traffic and exchanged data volume within an IoT environment. Furthermore, a manual approach is also enabled, in which domain analysts decide themselves which IoT objects should execute which operators. For this, domain analysts create a mapping plan manually.

In step ❹, according to the resulting mapping plan of step ❸, the processing operators can be deployed onto their corresponding IoT objects and the execution of the DSPM can be started. This step also supports manual actions, called human tasks in the semi-automatic deployment approach. The IoT objects are first prepared for executing the processing operators, i.e., the *software artifacts* required by a processing operator are installed and configured as necessary. Further software artifacts to collect empirical values and to monitor the processing operators and the IoT objects are also installed and configured. Afterwards, the processing operators are deployed onto their corresponding IoT objects and are then started, resulting in a running instance of the DSPM, which is called dDSPM. The preparation of IoT objects and the subsequent deployment of processing operators are realized by employing the TOSCA standard.

In step ❺, to assure that the overall processing in the IoT environment

stays correct (cf. Definition 1.1), IoT objects and the dDSPM are continuously monitored in order to recognize disturbances. If a disturbance affects the original processing negatively, the algorithms in step ❸ are restarted and a new mapping plan is created.

In step ❻, the data stream processing of an IoT application can be retired. In this case, IoT objects are cleaned, i.e., processing operators and software artifacts that are not needed anymore are stopped and uninstalled. For this, the TOSCA standard is employed as well. This step can be triggered by domain analysts or through programmatic analysis, e.g., when disturbances in the data processing occur and no correcting actions can be determined.

3.3 Overall architecture

This section presents an overview of the architecture comprising the contributions of this thesis. The architecture is depicted in Figure 3.2, in which the contributions are distinguished by color.

The overall architecture is composed of three main layers: the IoT physical environment layer, the IoT application layer, and the Multi-purpose Binding and Provisioning Platform (MBP) layer, which bridges the gap between IoT physical environments and IoT applications. The contributions of this thesis lie in the middle layer, for which the MBP was designed.

The *IoTEM modeler and manager* (a) component is the entry point of the methodical approach (cf. Section 3.2) and provides tools for domain experts to create, store, and manage IoTEMs. Through this component, the IoT objects of an IoTEM are registered in the MBP and their digital counterparts are instantiated. The digital counterparts provide APIs that can be accessed by IoT applications, for example, to access devices, sensors, and actuators registered in the MBP.

The *DSPM modeler and manager* (b) component provides further tools that enable domain analysts to create, store, and manage DSPMs describing the processing logic of IoT applications. Through this component, data analysts can also retire running instances of DSPMs (dDSPMs) once the processing is

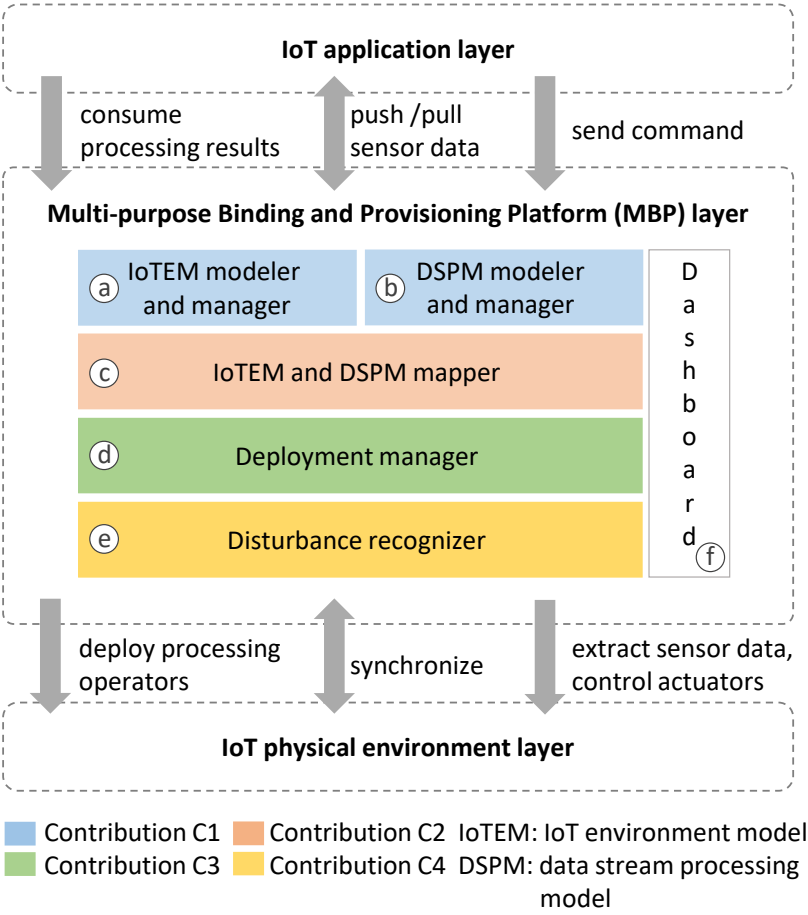


Figure 3.2: Overall architecture of this thesis

not needed anymore.

The *IoTEM and DSPM mapper* (c) component provides means to support the decision about where processing operators should be deployed. Furthermore, it provides algorithms to automatically decide about the operator placement, based on the requirements of the processing operators and the capabilities

of the IoT objects.

The *Deployment manager* (d) component is responsible for deploying processing operators onto IoT objects, so that the execution of DSPMs can be started. Furthermore, any further required software artifact, e. g., to monitor IoT objects, can be deployed through this component as well.

In the *Disturbance recognizer* (e) component, the digital counterparts of IoT objects and the dDSPM are continuously monitored to recognize disturbances during the data processing. This monitoring can be realized through scripts, such as Python or Shell scripts, or through more sophisticated implementations, such as complex event processing (CEP) queries. Hence, this component provides different runtime environments to realize the monitoring, including a CEP engine to continuously evaluate CEP queries.

Finally, metadata and dynamic data of IoT objects can be visualized by the *Dashboard* (f) component. It provides, for example, information about the availability and current disc space of IoT objects, historical data, and last measurements of sensor values.

Chapter 4

Modeling of IoT Environments and Data Stream Processing

This chapter presents contribution C1, the *IoT environment model* (IoTEM) and the *data stream processing model* (DSPM). In this thesis, the term *hardware objects* corresponds to devices, sensors and actuators, while the term *virtual objects* corresponds to virtual resources. The term *IoT object* is the generic term for hardware and virtual objects.

The IoTEM, which defines IoT objects within IoT environments, and the DSPM, which defines the data processing logic of IoT applications, describe different aspects of the IoT domain. Therefore, this thesis defines different IoT layers (depicted in Figure 4.1), in which further IoT models, comprehending different aspects, can be classified as well. There are two main layers: the physical layer and the digital layer. The *physical layer* refers to aspects describing hardware objects and their physical interconnections. In the *digital layer*, the *digital twin* refers to aspects describing running

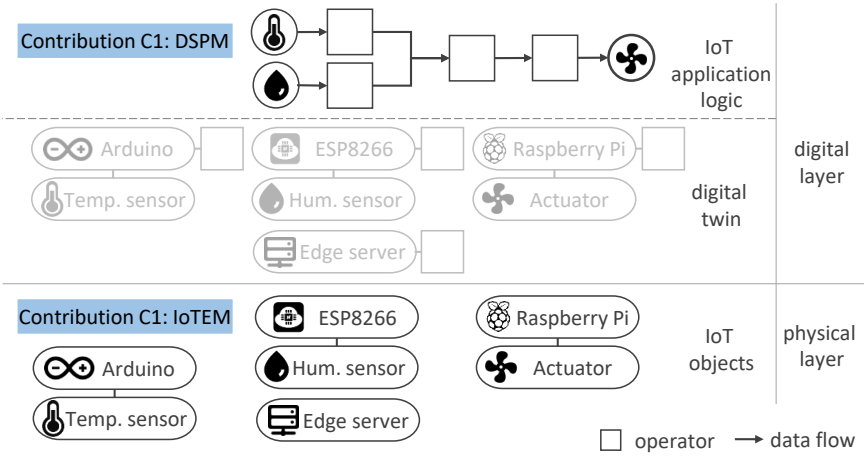


Figure 4.1: Layers of the Internet of Things [FH20]

operators provided within an IoT environment, for example, to access sensor data [HWBM16b]. The *IoT application logic* refers to models that logically use the operators provided by the *digital twin* to achieve specific goals of an IoT application. In the above described layers, the IoTEM focuses on the physical layer, while the DSPM focuses on the application logic in the digital layer. The IoTEM describes the *computing capabilities* of IoT objects in the IoT environments. On the other hand, the DSPM describes the *computing requirements* of IoT applications.

The IoTEM and DSPM are created by different roles: the *domain expert* and the *domain analyst*. Domain experts have technical knowledge about the IoT environment, e.g., information about the computing capabilities and how to interact with IoT objects. Domain analysts have domain knowledge about the data generated in the IoT environment and how these data need to be processed. In this way, domain analysts are able to describe the processing logic of IoT applications for domain-specific use cases.

The models IoTEM and DSPM are explained in detail in Section 4.1 and Section 4.2, respectively.

4.1 Modeling of IoT environments

The continuous progress in sensor and network technologies has enabled the existence of *IoT devices*, which are interconnected and continuously exchanging information about their surroundings and about themselves [AAS13; GBMP13]. IoT devices are typically embedded with or connected to *sensors* and *actuators*, through which the devices can sense and act in their surroundings. An environment containing one or more of such devices is called an *IoT environment*. Such environments exist in a variety of domains, such as smart homes [GBMP13], smart factories [Jaz14], or smart cities [VF13]. Furthermore, to enhance IoT environments with computing power, for example, to handle large amounts of data, virtual resources provided by cloud computing technologies can be employed as well [GBMP13].

In recent years, many IoT platforms have been developed [MMST16]. Normally, IoT objects are managed by IoT platforms, which provide access for IoT applications through high-level APIs. In many approaches, such as FIWARE [RGSE14], IBM Watson IoT [Nel16], OpenMTC [CCE+12], or Microsoft Azure IoT [Kle17], IoT objects are manually and individually registered and configured with these IoT platforms. This is, however, a complex and time-consuming task. Furthermore, IoT environments are very dynamic, e.g., devices might become faulty. Therefore, IoT platforms need to be in-sync with IoT environments in order to enable IoT applications to be aware of changes.

For this, contribution C1 provides the IoTEM to describe whole IoT environments. IoTEMs are integrated into and used by the Multi-purpose Binding and Provisioning Platform (MBP) to automatically register and configure IoT environments as a whole instead of configuring each IoT object individually. Registered IoT environments are constantly monitored to recognize critical changes in the environment, e.g., when a device becomes faulty.

Within this thesis, a comprehensive survey comparing different IoT models was conducted in [FH20], in order to choose a suitable model as IoTEM. In Table 4.1, the results of this survey are shown, in which a criteria-based comparison of several IoT models is presented.

Table 4.1: Criteria-based comparison of IoT models [FH20]: maturity ❶, hierarchy ❷, availability ❸, implementation ❹, geolocation ❺

Model	❶	❷	❸	❹	❺	Remarks
homeML	non-std.	✗	✗	✗	✓	Designed for smart homes [MNH+13]
IEEE 1451	std.	✗	✓	✓	✗	Focuses on transducers [IEE10]
IoT ARM	non-std.	✓	✗	✗	✓	Generic reference model [BBD+13]
IoT-Lite	submitted	✓	✓	✓	✓	Uses SSN ontology [BEBT17]
IoT MC	std.	✓	✓	✓	✗	Also known as IoTivity [Lin17]
IoT-O	std. ext.	✓	✗	✗	✗	Uses SSN ontology [AMMD15]
Nexus	non-std.	✓	✓	✓	✓	Focuses on geolocalization [NSM17]
oneM2M	std.	✓	✓	✓	✓	Focuses on services of IoT devices [one18]
OPC-UA	std.	✓	✓	✓	✗	Established in smart factories [Fou17a]
SenML	std.	✗	✓	✓	✓	Focuses on sensors and sensor values [JSA+18]
SensorML	std.	✗	✓	✓	✓	Supports processes [OGC14]
SSN	std.	✓	✓	✓	✓	Used by IoT-Lite / IoT-O [W3C05]
TDLIoT	non-std.	✗	✓	✓	✓	Research prototype [FHB+18]
Vorto	non-std.	✗	✓	✓	✗	Programming language [Ecl17]

In order to compare the IoT models, five criteria covering their most important characteristics are provided. These criteria were identified by a thorough investigation of available IoT models, and furthermore, from experiences in the scope of the German industry projects SmartOrchestra [Sma16] and IC4F [IC417], which have many industry partners with expertise in IoT applications. The criteria are explained in the following.

The first criterion specifies the **maturity** ❶ of the IoT models, e.g., whether an IoT model is an approved standard of an organization, such as OASIS, W3C or OGC, or not. It is assumed that an approved standard by such

organizations went under a thorough reviewing process and, thus, has been checked for feasibility. Furthermore, it is assumed that a standard provides advantages in contrast to, e.g., IoT models that were published in scientific papers and have not yet been extensively validated regarding feasibility. Consequently, maturity is an important factor while evaluating IoT models.

The second criterion refers to the support of **hierarchy ❷**. This is an important factor when modeling environments in the IoT, since they normally contain hierarchical deployments among the different existing IoT objects. There are two main types of hierarchies, *grouping* and *abstraction*. For example, through *grouping*, it should be possible to model complex systems, such as production machines in a smart factory, which contain a high amount of devices, sensors and actuators. This enables group-based querying. Such relations can be of vital importance, for example, when conducting monitoring for predictive maintenance [MS17]. Furthermore, through *abstraction*, generic types can be defined. For example, different sensor modules measuring temperature can be aggregated by the generic type *temperature sensor*. Consequently, this thesis investigate whether some support of hierarchies can be expressed in the IoT models. For example, an ontology-based model supports natively both mentioned types of hierarchies. Other models normally need to provide such means separately.

The third criterion **availability ❸** refers to whether the IoT model is publicly available or not, and furthermore, if a wide community is involved in its future development. Clearly, a large community of users and developers, or a larger organization, is required in order to establish and to further develop an IoT model. To realize this, the model should either be available open source, or, if it is closed source, it should be developed and used by a larger organization.

The fourth criterion refers to whether an **implementation ❹** of the IoT model exists. In scientific papers, for example, interesting concepts are created that, however, might not have a corresponding implementation. For the usage in real-world scenarios, an available implementation is of vital importance. This also includes available tools for model creation and management.

Finally, the fifth criterion **geolocation ❺** refers to whether the IoT model can describe (geo-)locations of IoT objects, which enable sophisticated features, such as location-based querying. Especially in the IoT, locations are important, for example, when recognizing situations, i.e., events that might require a reaction, which occur in a specific location, e.g., in a smart home.

As a result of the investigation, the IoTEM is based on the ontology IoT-Lite [BEBT16; BEBT17]. IoT-Lite is a lightweight instantiation of the SSN ontology [W3C05] and aims to reduce the complexity of other IoT models by describing only main IoT concepts, however, it can be extended to describe further concepts or details, enabling, in this thesis, also the modeling of computing capabilities of IoT objects. Furthermore, using an ontology as IoTEM enables the definition of semantics within the IoT environment and also reasoning can be conducted.

4.1.1 IoTEM definition

The IoTEM is an undirected graph containing IoT objects as nodes, and their network interconnections as edges [FHM19]. It is formalized as follows:

Definition 4.1 (IoTEM)
An IoT environment model is a tuple
$IoTEM := (Objects, Connections, distance, ObjectCapabilities, ConnectionCapabilities, objectCapabilityAssignment, connectionCapabilityAssignment)$, where:

- $Objects \neq \emptyset$: set of all IoT objects of the IoT environment.
- $Connections \subseteq \{(obj_i, obj_j) \mid i \neq j \wedge obj_i \neq obj_j \wedge obj_{i,j} \in Objects\}$: set of all undirected connections among the IoT objects in $Objects$.
- $distance$: $Connections \rightarrow \mathbb{R}$: weight function, which assigns the respective network distance to the connections.
- $ObjectCapabilities$: set of all IoT object capabilities in the IoT environment.

- $ConnectionCapabilities$: set of all connection capabilities in the IoT environment.
- $objectCapabilityAssignment: Objects \rightarrow \mathcal{P}(ObjectCapabilities)$: mapping, which assigns a set of capabilities to IoT objects.
- $connectionCapabilityAssignment: Connections \rightarrow \mathcal{P}(ConnectionCapabilities)$: mapping, which assigns a set of capabilities to connections.

so that $(Objects, Connections, distance)$ is an undirected and weighted graph with the network distance to define the weight.

For such an IoTEM, a network path corresponding to a series of connections, is defined as follows:

Definition 4.2 (IoTEM network path)
Let $IoTEM$ be an IoT environment model. A network $path$ in $IoTEM$ is a finite series of connections, where,
$path = \{(obj_0, obj_1), (obj_1, obj_2), \ldots, (obj_{n-1}, obj_n)\} : n, i \in \mathbb{N}, obj_i \in Objects$, such that:

1. $\forall obj_i \in \{obj_0, obj_1, \ldots, obj_{n-1}\} : \{obj_i, obj_{i+1}\} \in Connections$.
2. $\forall obj_i, obj_j \in \{obj_0, obj_1, \ldots, obj_n\} : i \neq j \wedge obj_i \neq obj_j$.

Furthermore, the following properties are defined:

- $section = (obj_i, obj_{i+1}) \mid i \in \{0, 1, \ldots, n-1\}$: a section of the network $path$.
- n: length of the network $path$.
- start$(path) = obj_0$: start of the network $path$.
- target$(path) = obj_n$: target of the network $path$.
- $Paths = \{path_0, path_1, \ldots, path_j \mid j \in \mathbb{N}\}$: set of all network paths in $IoTEM$.

The first condition for a network path ensures that all connections existing in the path also have a connection in the IoTEM. The second condition implies that a network path must not contain cycles or loops. It is assumed that the IoT objects are connected directly to each other in a Peer-to-Peer network [Dea15].

Furthermore, the distance of a network path $distance(path)$ in an IoTEM is defined as:

Definition 4.3 (Network distance of an IoTEM network path)

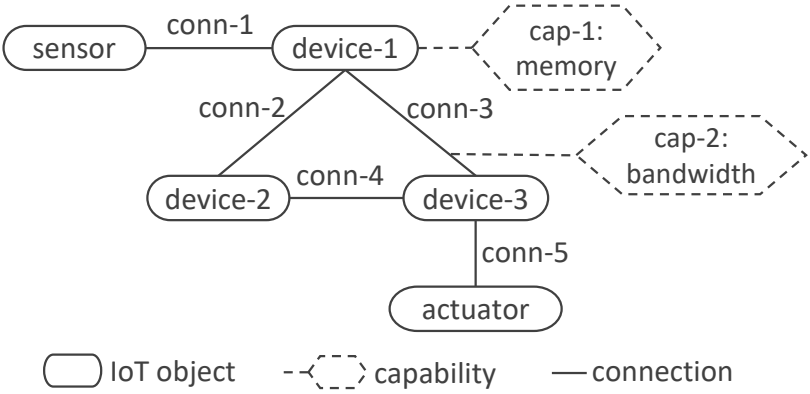
$$distance(path) = \sum_{\forall section \in path(\mathbb{N})} distance(section)$$

The total distance of a network path is the sum of all distances of its sections. The concrete distance calculation of the sections is implementation-specific and depends, for example, on latency or bandwidth. For instance, if a cloud infrastructure solution is available, the distance to it is typically much higher than to an on-premise IoT object. One possible metric to measure the distance of such sections within a network is introduced by Dean [Dea15]. An example for an IoTEM is depicted in Figure 4.2 including a graphical and formal representation.

4.1.2 IoT object and connection capabilities

In terms of computing capabilities, McEwen et al. [MC13] define an IoT object as a minicomputer, which provides network connectivity, processing power, and storage. Examples of IoT hardware objects are Arduino Yún, BeagleBone, ESP8266, and Raspberry Pi [SK17].

The following capabilities, which can be used to describe an IoT object, are derived from the McEwen et al. definition, and furthermore, from the constraints of heterogeneous and distributed environments presented by Cipriani et al. [CLM10]. The following alphabetic list represents a subset of possible capabilities, and hence, is not complete.



Objects = {device-1, sensor, device-2, device-3, actuator}
Connections = {conn-1, conn-2, conn-3, conn-4, conn-5}
ObjectCapabilities = {memory}
ConnectionCapabilities = {bandwidth}
objectCapabilityAssignment = {(device-1, cap-1)}
connectionCapabilityAssignment = {(conn-3, cap-2)}
distance = {(conn-1, 0), (conn-2, 10), ...}

Figure 4.2: Example of an IoTEM

- **Authentication**. This capability describes the supported authentication mechanisms of an IoT device. Specially important in the IoT domain is machine authorization, which is the authorization for machine-to-machine or human-to-machine communication. Examples of authentication mechanisms are passwords, digital credentials, and certificates [HA94].
- **Confidentiality**. This capability describes how the information stored in the IoT device is protected even if an unauthorized access occurs. Confidentiality can be reached by employing, for example, encryption mechanisms [HA94].
- **Device category**. This thesis defines four device categories, which dis-

tinguish among different deployment paradigms onto devices. These categories are the result of our investigation in [HBF+16] about how operator placement (i.e., software deployment) can be realized onto different types of IoT devices:

A *plug-and-play device* has embedded sensors and/or actuators. However, it does not allow remote deployment but rather provides APIs through a cloud server to access its sensors and actuators. Examples of such devices are wireless wearables [Fit09], which constantly synchronize their data to cloud servers. Therefore, operator placement on a plug-and-play device is not possible, however, operators can be deployed in other IoT objects connected to a plug-and-play device through the provided APIs.

A *configurable device*, such as the Raspberry Pi computer [Ras09], allows sensors and actuators to be attached to it through its physical interface (e.g., GPIO). Such a device has an operating system (OS), which enables a remote configuration and deployment of operators, for example, to extract and forward sensor data to an IoT platform.

A *constrained configurable device*, such as Arduino Yún [Ard13] and the Bosch XDK node [Rob16], has sensors and actuators embedded or attached to it and enables their remote configuration and software deployment. However, such devices are constrained, i.e., have limited processing and storage capabilities. In this case, complex processing operators should be avoided in this kind of device and the data to be processed should be rather forwarded to more powerful devices than being handled by the device itself.

A *gateway-dependent device* enables its configuration (e.g., firmware flashing) and software deployment only through a connection to a host device, in this thesis called a *gateway*. Generally, a gateway connects two systems using different formatting, communication protocols, or architectures [Dea15]. Examples for such devices are ESP8266 modules [Esp14] or Arduino Leonardo boards [Ard05]. In this case, the role of a gateway can be assumed by configurable devices physically

connected (e.g., through an USB port) to gateway-dependent devices. Through such physical connections, it is possible to upload operators into the gateway-dependent devices.

- **Memory**. This capability corresponds to the currently available working memory (i.e., RAM) of the IoT device.
- **Networking type**. This capability describes how an IoT device connects to other devices in the IoT environment. Examples of networking types are Ethernet (IEEE 802.3), Wi-Fi (IEEE 802.11), Bluetooth (IEEE 802.15.1), Bluetooth Low Energy (IEEE 802.15.4), 4G LTE, or 5G [AGM+15; SBH16; SK17].
- **Power consumption mode**. In the IoT domain, devices depending on batteries should be able to run on low-power consumption modes, in order to keep up with long-lasting IoT applications [MVD+14]. This capability describes the power consumption modes that an IoT device can support. For example, a Raspberry Pi provides four power modes, among them a run mode (default) and a standby mode [Hor13].
- **Processing power**. This capability corresponds to the CPU speed of the IoT device. For example, a Raspberry Pi 3 model B has four processors running at 1.2 GHz.
- **Storage capacity**. This capability corresponds to the available amount of storage. This is important for operators collecting sensor data or storing intermediate results.
- **Supported runtime**. This capability corresponds to a list of supported and available software runtime environments (e.g., Java, Python, CEP engines) of an IoT device.

As formalized in Definition 4.1, the capabilities of the network connections among IoT objects are modeled by the IoTEM as well. The following list shows a subset of possible connection capabilities, and hence, is not complete.

- **Bandwidth**. This capability provides an estimation about the amount

of data that can be transferred in a fixed amount of time by the connection.

- **Encryption**. This capability describes if the data exchange through the referred connection is encrypted.
- **Latency**. This capability provides an estimation of the time required to exchange data by a connection.
- **Networking type**. This capability describes the type of network connection, for example, wired or wireless.

4.1.3 Architecture component and implementation – IoTEM modeler and manager

In the overall architecture (cf. Section 3.3), the *IoTEM modeler and manager* component supports domain experts by providing the means to create and store IoTEMs, and furthermore, to manage registered IoTEMs to the MBP.

The IoTEM modeler and manager component is depicted in Figure 4.3. It contains the graphical *IoTEM modeling tool* (a), which is used to model the IoT environment by dragging and dropping *IoT object types* (b) into a modeling area. IoT object types, such as Raspberry Pis, temperature sensors, and virtual machines, are provided by an ontology. IoTEMs, which are instances of the ontology, are stored in the *IoTEM storage* (c).

Furthermore, the IoTEM modeling tool enables domain experts, besides storing and retrieving IoTEMs, to register all IoT objects of an IoTEM to the MBP at once. This registration triggers the *IoT objects manager* (d) to instantiate digital counterparts (i.e., digital twins) of the modeled IoT objects. These digital counterparts are stored in the *IoTEM storage*. The IoT objects manager constantly synchronizes with the physical IoT environment in order to recognize disturbances in the physical environment, such as when devices become faulty. The current status of the registered IoT objects can be checked and visualized in the MBP dashboard. How the recognition of disturbances is realized will be explained in detail in Chapter 7.

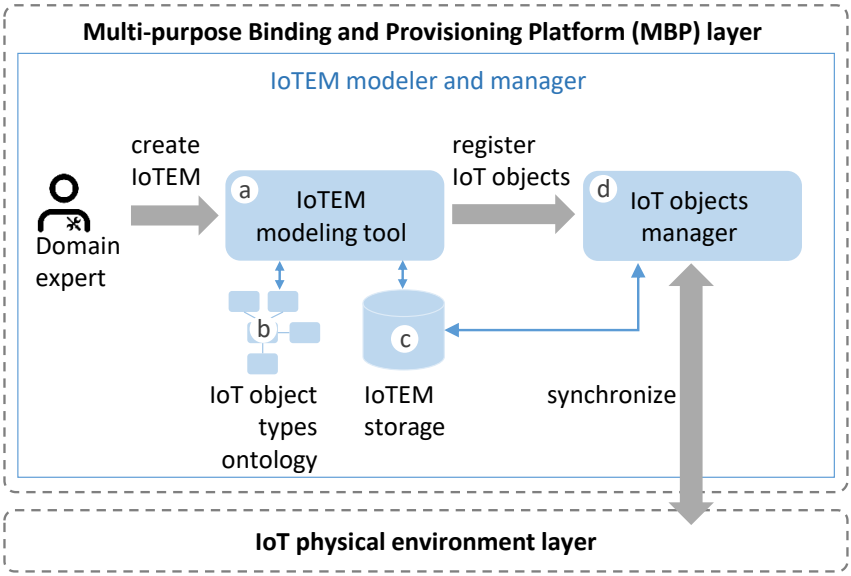


Figure 4.3: Architecture component: IoTEM modeler and manager

The *IoTEM modeler and manager* component has been prototypically implemented as part of the MBP, which is available as an open-source GitHub project[1]. Section 8.2 gives an overview on the MBP functionalities. The implementation of this component uses mainly JavaScript on the graphical user interface and Java on the server side. The *IoTEM modeling tool* uses the JavaScript library jQuery and the jsPlumb Toolkit for the implementation of a drag and drop editor[2] for the modeling of IoT environments in a user-friendly manner. The *IoT object manager* is implemented on the server side using Java. A REST API[3] reflecting the same functionality for the management of IoT objects has been implemented on the server side as well. The MongoDB database is used as the *IoTEM storage* to store IoT object types and modeled

[1] https://github.com/IPVS-AS/MBP
[2] https://github.com/IPVS-AS/MBP/wiki/Modeling-IoT-Environments
[3] https://github.com/IPVS-AS/MBP/wiki/API-Reference

IoTEMs. Section 8.1 presents the integration architecture of all architecture components and shows how they fit together.

4.1.4 Related work

Many approaches exist for the modeling of IoT environments (cf. Table 4.1). They provide IoT models and tools for the creation and management of instances of these IoT models. Tool support is of vital importance since it abstracts the complexity of IoT models (e.g., complex data formats) and eases the task of modeling and management. However, state-of-the-art approaches do not provide the means to support the entire life cycle of IoT environments, i.e., from modeling, through deployment, until undeployment. More concretely, they do not provide a holistic concept as in this thesis for the modeling, configuration, and monitoring of IoT environments.

McDonald et al. [MNH+13] introduce a modeling tool for the model homeML [NFD+07], which is, however, tailored for smart homes. In contrast, this thesis employs an IoT model that is generic and can be also applied to other IoT domains, such as smart factory or smart city.

Mayer et al. [MIV+14; MIVV14] show an approach combining semantic metadata and reasoning with graphical modeling, in which goals of an IoT environment can be configured, for example, the desired temperature of a room during the day. However, this approach assumes that the IoT environment is already deployed and computing operators are already running.

Moreover, there exist a large amount of IoT platforms [BMP13; MMST16; SK17], such as FIWARE [RGSE14], GSN [AHS06; AHS07], IBM Watson IoT [Nel16], OpenIoT [SKH+15], OpenMTC [CCE+12], or Microsoft Azure IoT [Kle17]. However, they do not provide the means to model and manage complete IoT environments, i.e., IoT objects are manually and individually registered and configured to these IoT platforms.

This thesis employs the IoT-Lite ontology [W3C15] for the modeling of IoT environments and provides a graphical modeling and management tool for IoT environments, the IoTEM modeling tool (cf. Figure 4.3). Besides the IoT-Lite ontology, other formats and standards could also be used to model

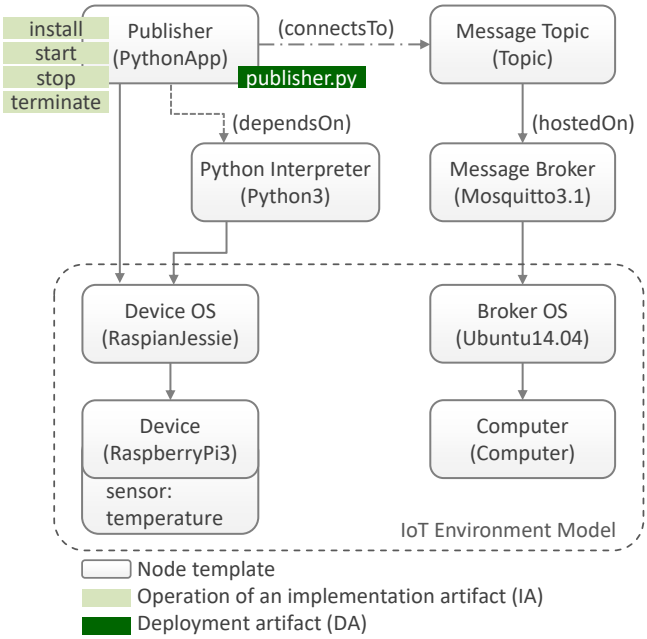


Figure 4.4: TOSCA topology model for an IoT environment (based on [FBK+16])

IoT environments, such as the SSN [W3C05] or oneM2M Base [one18] ontologies. In this thesis, a thorough literature review on state-of-the-art IoT models was conducted. Table 4.1 shows the results of this literature review, in which a criteria-based comparison of IoT models is presented. As a consequence of this comparison, the IoT-Lite ontology has shown to be the best suitable model for the purposes of this thesis.

Furthermore, more generic models that are not specifically tailored for IoT environments, can be employed as well. For instance, the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [OAS13] could be employed, which is originally designed for cloud applications, but offers nonetheless the means to model IoT environments and IoT applications as we investigated in [FBH+17; FBK+16; FHB+17]. An abstract TOSCA

topology model including an IoT device and an IoT application is depicted in Figure 4.4. The node templates on the bottom correspond to the IoT objects of the IoT environment.

However, IoT-specific models, such as ontologies, offer many advantages, such as predefined types, semantic descriptions and tool-support for semantic resolution [SI02]. Vermesan et al. [VFG+13] argue that, to reach the full potential of IoT applications, semantic information about IoT objects is a premise. For these reasons, the IoT-specific ontology model IoT-Lite was employed in this thesis.

4.2 Modeling of data stream processing

This section explains in detail the data stream processing model (DSPM).

Due to continuous sensor readings and frequent data exchange among IoT objects, high amounts of data are generated within IoT environments. These data incorporate the form of *data streams*, which are not persistent but rather arrive for processing in multiple, continuous, rapid, time-varying streams [BBD+02]. Well-established paradigms to process data streams are complex event processing and stream processing [CM12]. Contribution C1 provides the DSPM to describe the processing logic of IoT applications, i.e., how data streams in the IoT environment should be processed.

4.2.1 DSPM definition

The DSPM is a graph-based model containing data sources, data sinks, processing operators, and the data flow between these operators [FHM19]. It is based on the pipes and filters design pattern [Meu95], which provides a structure for the processing of a data stream [BMR+96]. The input for the processing is provided by the data sources, e.g., sensors, producing sequences of data values in the same structure. On the other hand, the output of the processing reaches data sinks. A processing operator corresponds to a *filter*, and the data flow between two adjacent processing operators corresponds to a *pipe*. Furthermore, the connections between data sources and processing

operators and between processing operators and data sinks correspond to pipes as well. This thesis uses a pipes and filters variant, in which a processing operator can have multiple inputs and outputs [BMR+96].

The DSPM is formalized as follows:

Definition 4.4 (DSPM)
A data stream processing model is a tuple
$DSPM := (Sources, Sinks, Operators, Edges, OperatorRequirements, EdgeRequirements, operatorRequirementAssignment, edgeRequirementAssignment)$, where:

- $Sources \neq \emptyset$: set of all data sources of the DSPM.
- $Sinks \neq \emptyset$: set of all data sinks of the DSPM.
- $Operators \neq \emptyset$: set of all processing operators of the DSPM. This set also contains operators that extract data from data sources, and furthermore that serve data to data sinks.
- $Edges \subseteq ((Operators \times Operators) \cup (Sources \times Operators) \cup (Operators \times Sinks))$: set of edges between operators, between sources and operators, and between operators and sinks.
- $OperatorRequirements$: set of all operator requirements in the DSPM.
- $EdgeRequirements$: set of all edge requirements in the DSPM.
- $operatorRequirementAssignment : Operators \rightarrow \mathcal{P}(OperatorRequirements)$: mapping, which assigns a set of requirements to operators.
- $edgeRequirementAssignment : Edges \rightarrow \mathcal{P}(EdgeRequirements)$: mapping, which assigns a set of requirements to edges.

so that $(Operators, Edges)$ forms a directed, unweighted, loop-free and acyclic graph. Moreover, the DSPM describes the data flow and not the control flow [ACÇ+03].

To properly create DSPMs, domain analysts need to know which data sources and sinks are available in the IoT environment. For this, only general

descriptions of sensors and actuators need to be extracted from the IoTEM in order to abstract them to domain analysts as data sources and sinks. Furthermore, operators and connections are annotated with computing requirements. An example of a requirement is the minimum storage required by an operator that stores intermediate results. Based on the computing requirements of operators, it is possible to search for a suitable operator placement in the IoT environment, which meets all the requirements of a modeled DSPM. Possible computing requirements are those meeting the exemplary computing capabilities of IoT devices described in Section 4.1.2.

For some requirements, such as required main memory, domain analysts typically do not know which concrete values should be used. This knowledge is highly dependent on the amount and complexity of the data and, thus, needs to be inferred through empirical values. Therefore, empirical values of operators are collected during their execution. These values are then processed using analytical techniques to provide domain analysts with recommendations about the requirements at modeling time.

Figure 4.5 shows an example of a DSPM including data sources, sinks, and processing operators containing graphical and formal representations. The goal of the exemplary application is to monitor mold levels in different rooms of a building in order to recognize when mold levels rise to unhealthy ranges. The data sources are represented by temperature sensors (So1, So3) and humidity sensors (So2, So4), while the data sinks are represented by dashboards (Si1, Si2, Si3). Furthermore, the example uses operators to *extract* sensor data, to *join* sensor data based on time windows, to *calculate* mold levels, and to *serve* data to data sinks. For each room, the mold level is continuously calculated based on temperature and humidity values and is analyzed over time in order to detect increases.

4.2.2 Processing operators

In the context of this thesis, a processing operator is a software component that executes an operation (e.g., data filtering, average function) on one or more input data streams. These software components can be, for example,

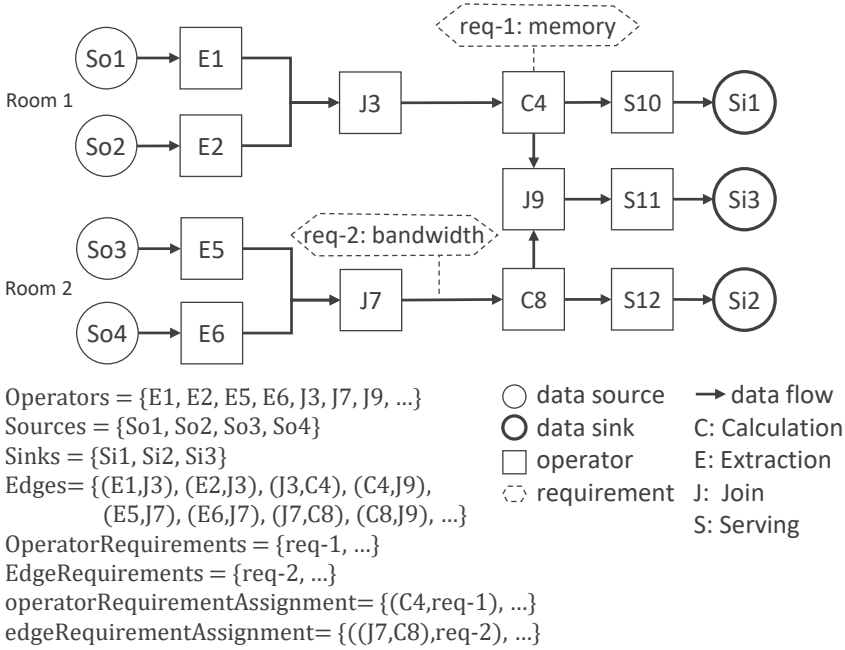


Figure 4.5: A DSPM for an application in the domain smart building

Python or Shell scripts, JAR files, or continuous processing queries. The suitable software runtime environment (e.g., Java RE, Python interpreter, stream processing engine) required by a processing operator needs, therefore, to be modeled in the DSPM as a computing requirement of this operator.

Processing operators are normally executed continuously and are based on windows, i.e., intervals of data based on time or number of elements, due to the data stream's nature of being infinite [BBD+02; CM12]. Furthermore, window-based operations are imperative, since the data within IoT environments can be imprecise or become stale rapidly [ACÇ+03]. The output of a processing operator leads to one or more data streams, which can be sent to further processing operators or to data sinks.

Processing operators are assumed to be ready for operator placement

procedures, i.e., necessary mechanisms to distribute the data processing have been already employed. Approaches realizing stream processing in distributed environments can be found in [AAB+05; CM13; LF98; SMP09]. The following alphabetic list shows an incomplete subset of processing operators, which can be employed to process data within IoT environments.

- **Aggregation**. This operator applies an aggregation function to a set of values (i.e., a window) in a data stream and returns a single value [ACÇ+03]. Examples of aggregation are average, minimum value, and maximal value calculations.
- **Calculation**. This operator realizes a calculation based on values of one or more input streams. For example, in Figure 4.5, the calculation operator is a function, which calculates mold levels based on temperature and humidity values.
- **Extraction**. This operator extracts data from the data sources, for example, it can be the software code accessing the physical interface of a sensor (e. g., through GPIO) to get actual sensor measurements.
- **Filter**. This operator filters the data based on parameters.
- **Join**. This operator joins two data streams.
- **Pattern detection**. This operator recognizes certain patterns in a data stream [EBB+11]. An example of a pattern derived from CEP concepts is an event sequence, i.e., specified events occurring in a certain order.
- **Serving**. This operator serves data to data sinks, for example, it can be the software code accessing the physical interface of an actuator to control it, or the software code sending data to a dashboard application.
- **Transformation**. This operator realizes transformations, e.g., among different data formats or data schemas.

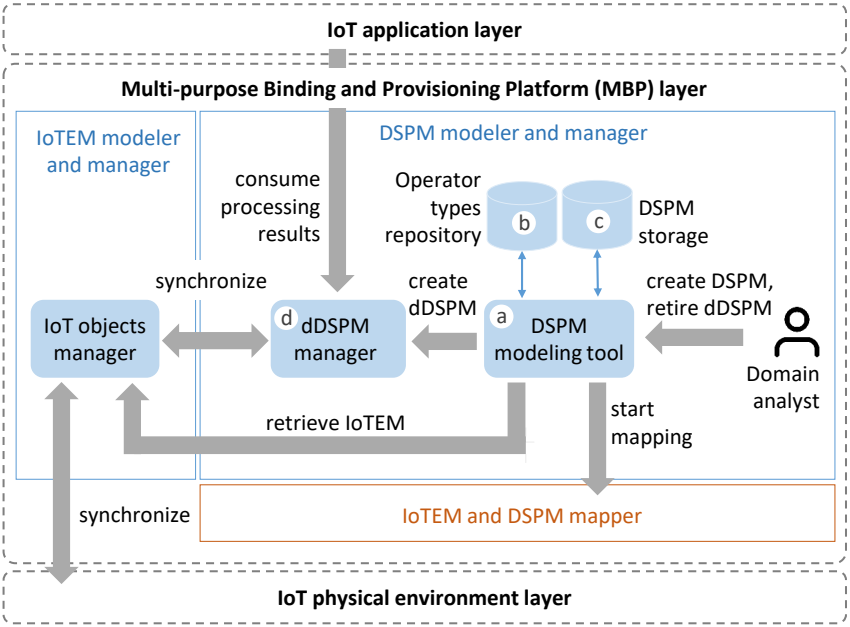


Figure 4.6: Architecture component: DSPM modeler and manager

4.2.3 Architecture component and implementation – DSPM modeler and manager

In the overall architecture (cf. Section 3.3), the *DSPM modeler and manager* component supports domain analysts by providing the means to create and store DSPMs, and furthermore, to manage deployed DSPMs onto IoT environments. This component is depicted in Figure 4.6.

This thesis uses the *FlexMash tool* [HB17; Hir18; HM16], which was proposed by Hirmer et al., as the *DSPM modeling tool* (a) for the creation of DSPMs. It provides a graphical interface and its JSON-based underlying model can be extended to enable the annotation of computing requirements

on nodes and edges, which is a premise for the concepts of this thesis.

The DSPM modeling tool enables the creation of DSPMs by dragging and dropping data sources, data sinks, and processing operators into the modeling area. While operators are available in the *Operator types repository* (b), data sources and sinks are abstracted from sensors and actuators existent in IoTEM models, which are retrieved from the IoT objects manager. Furthermore, modeled DSPM are stored in the *DSPM storage* (c).

Through the DSPM modeling tool, the mapping of a DSPM onto an IoTEM is started, and subsequently, the deployment based on the created mapping plan is started as well. Once a DSPM has been successfully deployed and started by the *Deployment manager* component(cf. Chapter 6), the *dDSPM manager* (d) gets informed about the success of the deployment and creates a corresponding dDSPM, which contains information about which IoT object is processing which operator. The dDSPM manager constantly retrieves status information from the IoT objects manager, to get, for example, insights of the health of the deployed operators and existing IoT objects. This information can be visualized in the MBP dashboard. Finally, the modeling tool also enables domain analysts to stop and retire dDSPMs, what is managed by the dDSPM manager.

The *DSPM modeler and manager* component has been prototypically implemented as part of the MBP, except for the DSPM modeling tool, which corresponds to an external tool available in the FlexMash GitHub project[1]. The *DSPM modeling tool* corresponds to the graphical user interface of the FlexMash tool [Hir18]. In the scope of this thesis, the FlexMash graphical user interface and its JSON-based underlying model were extended in the master thesis conducted by Chaudhry [Cha18]. The extensions include enabling the annotation of requirements on processing operators and on edges. FlexMash was also extended with communication interfaces to the server side of the MBP, in order to import sources and sinks from IoTEMs, to trigger the automated mapping and deployment of operators, and to retire dDSPMs. The FlexMash graphical user interface was implemented in

[1] https://github.com/hirmerpl/FlexMash

JavaScript and uses jQuery and the jsPlumb Toolkit. Modeled DSPMs are stored in the MBP server in a MongoDB database. The *dDSPM manager* is implemented in the MBP server using Java. The *Operator types repository* corresponds to a file system in the installation location of the MBP. Several operators have been implemented in Python and Shell, which can be found in the MBP GitHub project[1]. Finally, Section 8.1 presents the integration architecture of all architecture components and shows how they fit together.

4.2.4 Related work

There exist many approaches to model data flows and data processing for the IoT [BL14; SBS+17], such as COMPOSE [Man+13; PVC+14], Huginn [Can+13], IFTTT [IFT11], LabVIEW [TK07], Node-RED [JS 13], WoTKit [BL12a; BL12b], WoT Flow [BL14], or Yahoo! Pipes [Pru07]. However, they either assume that the processing is executed centralized or that operators and services are already running in the IoT environments.

A thorough literature review on state-of-the-art data flow models was conducted by Hirmer [Hir18], who proposed an approach for the modeling and execution of requirements-based data flows. In his work, the FlexMash tool [HB17; HM16] has been developed, which serves as a basis for this thesis in respect to the modeling of data stream processing.

More generic models that are not specifically tailored to model data processing can be employed as well, for instance, the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard [OAS13]. In [FHB+17], we investigate how to model complex event processing in TOSCA and afterwards use the resulting TOSCA model to deploy and start the processing. An abstract TOSCA topology model for complex event processing is depicted in Figure 4.7, in which several types of operators are modeled. For example, it contains operators extracting temperature and humidity sensor values, and subsequently, one operator transforming these values into a format that can be understood by the CEP system.

[1] https://github.com/IPVS-AS/MBP/tree/master/resources/operators

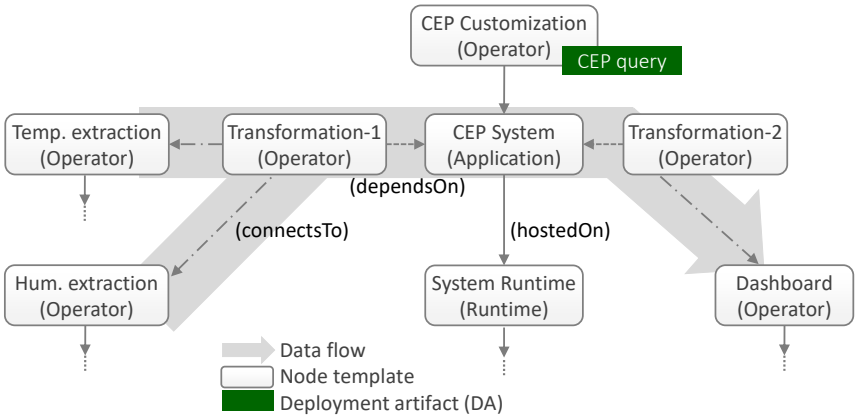


Figure 4.7: TOSCA topology model of a complex event processing example (based on [FBH+17])

Finally, many IoT platforms have been developed that include a data processing layer [BMP13; MMST16; SK17], however, they either do not provide the means to model the data processing in a user-friendly way (e.g., the modeling is realized by textual input) or only the modeling of simple rules (e.g., if-statements) is supported. Furthermore, they assume the processing is executed centralized, for example, by the IoT platform itself, and do not support, in this way, the concepts of operator placement.

Chapter 5

Mapping of DSPMs onto IoTEMs

This chapter presents contribution C2, which is the mapping of data stream processing models (DSPMs) onto IoT environment models (IoTEMs). This contribution contains two approaches to realize the mapping: (i) an automatic approach, in which a *mapping plan* is automatically generated by different algorithms, and (ii) a manual approach, in which domain analysts decide themselves which operators should be executed by which IoT objects [FHST20]. In this case, a mapping plan is created manually.

The resulting mapping plan can be, for example, based on a software orchestration standard, such as TOSCA [OAS13], or based on other established provisioning tools, such as Shell scripts, Chef [Che09], or Puppet [Pup05]. Based on the mapping plan, the next contribution C3 (cf. Chapter 6) deals with the deployment of operators in the IoT environment.

The mapping approaches are explained in detail in Sections 5.1 and 5.2. The architecture component realizing the mapping approaches is described in Section 5.3. Finally, Section 5.4 presents related work to contribution C2.

5.1 Automatic mapping approach

This approach aims to automatically decide which IoT objects should process which operators. This is realized by means of algorithms that are able to evaluate and match the overall capabilities of the IoTEM and the requirements of the DSPM. In this thesis, this is referred to as the operator placement problem (cf. Section 2.3). The problem of distributing operators is NP-complete, therefore, heuristic algorithms, such as those presented by Lo [Lo88], are normally used to solve this kind of problem.

Based on the DSPM and the IoTEM, the operator placement problem of this thesis is formalized as follows:

Definition 5.1 (DSPM operator placement problem)
Let $IoTEM := (Objects, Connections, distance, ObjectCapabilities, ConnectionCapabilities, objectCapabilityAssignment, connectionCapabilityAssignment)$ be an IoT environment and $DSPM := (Operators, Sources, Sinks, Edges, OperatorRequirements, EdgeRequirements, operatorRequirementAssignment, edgeRequirementAssignment)$ be a data stream processing model for the $IoTEM$. The operator placement problem aims to find a $solution := (operatorMapping, edgeMapping)$, where:

- $sourceMapping : Sources \rightarrow Objects$: mapping that assigns the data sources of the DSPM to the dedicated IoT objects in the IoTEM.
- $sinkMapping : Sinks \rightarrow Objects$: mapping that assigns the data sinks of the DSPM to the dedicated IoT objects in the IoTEM.
- $operatorMapping : Operators \rightarrow Objects$: mapping that assigns computing operators of the DSPM to IoT objects of the IoTEM.
- $edgeMapping : Edges \rightarrow Paths(IoTEM)$: mapping that assigns edges in the DSPM to connection paths of the IoTEM.
- $satisfies : \mathcal{P}(OperatorRequirements) \times \mathcal{P}(ObjectCapabilities) \wedge \mathcal{P}(EdgeRequirements) \times \mathcal{P}(ConnectionCapabilities) \rightarrow \{\text{true}, \text{false}\}$:

function determining if the requirements of the DSPM are fulfilled by the capabilities of the IoTEM.

The solution is considered *valid*, if:

1. $\forall so \in Sources : sourceMapping(so) \neq \emptyset$
2. $\forall si \in Sinks : sinkMapping(si) \neq \emptyset$
3. $\forall (op_1, op_2) \in Edges$ with $map := edgeMapping((op_1, op_2)) :$
   $start(map) = operatorMapping(op_1) \ \wedge$
   $target(map) = operatorMapping(op_2)$
4. $\forall op \in Operators : satisfies(operatorRequirementAssignment(op),$
   $objectCapabilityAssignment(operatorMapping)(op)))$
5. $\forall edge \in Edges$ with $map := edgeMapping(e) :$
   $\forall section \in map(\mathbb{N}) : satisfies(edgeRequirementAssignment(edge),$
   $connectionCapabilityAssignment(section))$

The first and second conditions ensure that the mapping of extraction and serving operators in a DSPM includes the mapping of the corresponding data sources and sinks.

The third condition ensures that the structure of the DSPM is preserved in the edge mapping. For this, the IoT object to which the start operator of an edge is mapped must be the start of the connection path to which the edge is mapped. Furthermore, the IoT object to which the target operator of an edge is mapped must be the destination of this connection path.

The fourth and fifth conditions ensure that IoT objects and connection paths meet all requirements of the operators and edges to which they are mapped. For this, all the sections in a connection path have to meet the requirements of an edge.

In this thesis, two algorithms are provided to solve the DSPM operator placement problem: (i) a greedy variant and a (ii) backtracking variant.

The *greedy variant* finds a solution timely, however, it does not guarantee to find the best possible solution. Lo [Lo88] shows that a simple greedy

algorithm is highly efficient and performs nearly as well as more complex heuristic algorithms. In contrast, the *backtracking variant* computes all possible solution, thus, it is able to select the best possible one. The backtracking variant, however, requires higher execution time and should, therefore, be used for simple scenarios involving fewer IoT objects.

The main goal of both algorithms is to find a set of IoT objects, whose capabilities satisfy the requirements of the operators. IoT objects near to data sources are preferred, enabling, e.g., early detection of critical situations and corresponding timely actions [FHWM16]. Furthermore, the data volume exchanged in the IoT environment can be reduced by aggregating and filtering data near to their sources and as early as possible in the data processing chain. For both algorithms, it is assumed that IoT objects physically connected to sensors or actuators through their hardware interfaces offer enough computing capabilities to run sensor data extraction or data serving operators. That is, a data extraction operator can always be placed directly onto the IoT object physically connected to the corresponding sensor. Similarly, a data serving operator can always be placed onto the IoT object physically connected to the corresponding actuator.

The algorithms are explained in detail in Sections 5.1.1 and 5.1.2.

5.1.1 Matching algorithm – greedy variant

The algorithm in pseudo-code depicted in Algorithm 5.1 requires as input the IoTEM and the DSPM (cf. Definition 4.1 and Definition 4.4). The output returns a mapping of operators and edges of the DSPM onto the IoT objects and their connections of the IoTEM.

In the following, the greedy algorithm is described step-by-step. In line 6, the function $GetTopologicalOrder$ is called, which takes as input the DSPM with its $Operators$ and $Edges$. This function returns a mapping order: $\{0, 1, \ldots, |Operators|\} \rightarrow Operators$ which corresponds to the topological sorting [CLRS09] of the operators. In line 7, the function $FillSourceMapping$ is called, which takes as input the DSPM and IoTEM and returns a mapping of sources abstracted from sensors and the dedicated IoT

Algorithmus 5.1 Greedy variant (based on [FHM19])

```
1: function GreedyMatching((Sources, Sinks, Operators, Edges, ...),
   (Objects, Connections, ...))
2:     DSPM ← (Sources, Sinks, Operators, Edges, ...)
3:     IoTEM ← (Objects, Connections, ...)
4:     operatorMapping : Operators → Objects
5:     edgeMapping : Edges → Paths(IoTEM)
6:     order ← GetTopologicalOrder(Operators, Edges)
7:     sourceMapping ← FillSourceMapping(IoTEM, DSPM)
8:     sinkMapping ← FillSinkMapping(IoTEM, DSPM)
9:     i ← 0
10:    while i < |Operators| do
11:        op_j ← order(i)
12:        if (op_j.isConnectedToSource()) then
13:            opObject ← sourceMapping(op_j)
14:            ConsumeCaps(operatorRequirementAssignment(op_j),
   objectCapabilityAssignment(opObject))
15:            operatorMapping(op_j) ← opObject
16:        else if (op_j.isConnectedToSink()) then
17:            opObject ← sinkMapping(op_j)
18:            ConsumeCaps(operatorRequirementAssignment(op_j),
   objectCapabilityAssignment(opObject))
19:            operatorMapping(op_j) ← opObject
20:        else
21:            PreOperators ← {op_i ∈ Operators : ∃(op_i, op_j) ∈ Edges}
22:            PreObjects ← operatorMapping(PreOperators)
23:            closestOrder ← GetClosestObjects(IoTEM, PreObjects)
24:            foundObject ← false
25:            j ← 0
26:            while (j < |Objects| & ¬foundObject) do
27:                opObject ← closestOrder(j)
28:                if TryMapOperator(op_j, opObject, operatorMapping,
   edgeMapping, DSPM, IoTEM) then
29:                    foundObject ← true
30:                end if
31:            end while
```

```
32:             if ¬foundObject then
33:                 return "No solution found"
34:             end if
35:         end if
36:     end while
37:     return (operatorMapping, edgeMapping)
38: end function
```

objects connected to these sensors. Similarly, the function $FillSinkMapping$ in line 8 returns a mapping of sinks abstracted from actuators and dedicated IoT objects connected to these actuators.

Next, the sorted list of operators is traversed. If the current operator is connected to a data source node, the IoT object attached to the data source (i.e., the sensor) is retrieved (line 13). The capabilities of this IoT object are then adjusted by subtracting the required resources of the operator (line 14). Finally, the operator is mapped onto the IoT object (line 15). Then, the loop continues with the next operator. If the current operator is connected to a data sink node (line 16), it is handled similar to a data source as above described.

In line 21, a list of predecessor operators of the current operator in the DSPM is determined and, subsequently, the IoT objects hosting these predecessor operators are also retrieved (line 22). The function $GetClosestObjects$ sorts all IoT objects in the IoTEM (line 23) based on their network distance to the predecessor IoT objects. The result is the map $closestOrder: \{0, 1, \ldots, |Objects|\} \rightarrow Objects$, whereas $closestOrder(0)$ refers to the IoT object closest to the predecessor IoT objects. The predecessor IoT objects are also part of this sorting and they appear first because their network distance is 0.

After that, the list of closest IoT objects is traversed (line 26) and it is checked, whether the operator can be mapped onto one of the closest IoT objects, depending on their available capabilities. If the mapping is possible, i.e., the function $TryMapOperator$ returns $true$ (line 28), an IoT object was found and the operator was successfully mapped onto the found IoT object. Otherwise, the next nearest IoT object is checked. If no IoT object

has the required capabilities, no solution can be found. In this case, the algorithm will use a cloud infrastructure as a fallback, so that a solution can be ensured.

Finally, the algorithm returns a set of mappings of operators and objects (line 37). The greedy variant returns a solution, however, it might not be the best one regarding to the minimum network distance between operators.

5.1.2 Matching algorithm – backtracking variant

Algorithm 5.2 shows the backtracking algorithm in pseudo-code. The goal of this variant is not only to find one solution, but the best one. The input is the same as for the greedy algorithm, i.e., the IoTEM und DSPM.

In lines 11–15, all operators connected to the sources of the DSPM are traversed, the corresponding IoT objects are retrieved, and the data extraction operators are mapped onto these IoT objects. After that, the extraction operators are removed from the list of operators. In lines 16–21, all operators connected to sinks are also traversed, the corresponding IoT objects are retrieved, and the data serving operators are mapped onto these IoT objects. After that, the serving operators are also removed from the list of operators.

Next, the function $FindSolution$ is called (line 24), which contains the logic for the requirements and capabilities matching.

The $FindSolution$ function, which is depicted in Algorithm 5.3, realizes a depth-search for possible solutions using recursion. In line 5, it is checked how far the algorithm has already advanced. If the set of operators still contains elements, these are first mapped onto the IoT objects.

In line 3, the $foundSolution$ variable is defined, which memorizes if a solution could be found or not. In line 4, the $GetOneElement$ function returns the current operator. After that, IoT objects are searched that are suitable for executing this operator by iterating all IoT objects of the IoTEM. It is important to note that the order in which the IoT objects are processed is the same for each iteration. Otherwise, it is no longer possible to backtrack, i.e., undo the mapping steps later and to look for alternatives.

In line 6, it is checked whether the current device satisfies the require-

Algorithmus 5.2 Backtracking variant (based on [FHM19])

```
1: function BacktrackingMatching((Sources, Sinks, Operators, ...),
   (Objects, Connections, ...))
2:     DSPM ← (Sources, Sinks, Operators, ...)
3:     IoTEM ← (Objects, Connections, ...)
4:     operatorMapping : Operators → Objects
5:     edgeMapping : Edges → Paths(IoTEM)
6:     solution ← (operatorMapping, edgeMapping)
7:     sourceMapping ← FillSourceMapping(IoTEM, DSPM)
8:     sinkMapping ← FillSinkMapping(IoTEM, DSPM)
9:     RestOperators ← Operators
10:    for op ∈ RestOperators do
11:        if (op.isConnectedToSource()) then
12:            opObject ← sourceMapping(op)
13:            ConsumeCaps(operatorRequirementAssignment(op)
   objectCapabilityAssignment(opObject))
14:            operatorMapping(op) ← opObject
15:            RestOperators ← RestOperators \ op
16:        else if (op.isConnectedToSink()) then
17:            opObject ← sinkMapping(op)
18:            ConsumeCaps(operatorRequirementAssignment(op)
   objectCapabilityAssignment(opObject))
19:            operatorMapping(op) ← opObject
20:            RestOperators ← RestOperators \ op
21:        end if
22:    end for
23:    Solutions ← ∅
24:    foundSolution ← FindSolution(solution, RestOperators, Edges,
   Solutions, DSPM, IoTEM)
25:    if (Solutions = ∅ | ¬foundSolution) then
26:        return "No solution found"
27:    end if
28:    return GetBestSolution(Solutions, Edges, distance)
29: end function
```

ments of the operator *nextOperator*. If this is not the case, the next IoT object is considered. If an IoT object is a match, the operator is removed

Algorithmus 5.3 *FindSolution* function pseudo-code (based on [FHM19])

```
1: function FindSolution(solution, Operators, Edges, Solutions,
   DSPM, IoTEM)
2:     if Operators ≠ ∅ then
3:         foundSolution ← false
4:         nextOperator ← GetOneElement(Operators)
5:         for currObject ∈ Objects do
6:             if SatisfiesReqs(operatorRequirementAssignment(next
   Operator), objectCapabilityAssignment(currObject)) then
7:                 ConsumeCaps(operatorRequirementAssignment(next
   Operator), objectCapabilityAssignment(currObject))
8:                 operatorMapping(nextOperator) ← currObject
9:                 Operators ← Operators \ {nextOperator}
10:                if FindSolution(solution, Operators, Edges,
   Solutions, DSPM, IoTEM) then
11:                    foundSolution ← true
12:                    return true
13:                else
14:                    Undo(operatorRequirementAssignment(next
   Operator), objectCapabilityAssignment(currObject))
15:                    operatorMapping(nextOperator) ← null
16:                    Operators ← Operators ∪ {nextOperator}
17:                end if
18:            end if
19:        end for
20:        return foundSolution
21:    else
22:        if Edges ≠ ∅ then
23:            // similar to operators
24:        else
25:            Solutions ← Solutions ∪ {solution}
26:            return true
27:        end if
28:    end if
29: end function
```

from the set of operators that still need to be mapped (line 12). Then, the *FindSolution* function is called recursively. By doing so, all possible

solutions are found, considering the mapping of each operator with each IoT device. This leads to a higher runtime compared to the greedy algorithm. Consequently, for a mapping realized at runtime or for applications that require constantly updating the mapping of operators, the backtracking algorithm is not recommended.

The mapping of the edges is similar to the operator mapping as described above. In this case, however, all edges are checked to be possible solutions based on the already existing mapping of operators onto devices. If the set of operators and the set of edges are both empty, they were all mapped and a solution was found. In this case, the solution is appended to the set of $Solutions$ (line 25).

After the $FindSolution$ function terminates, the $GetBestSolution$ function in line 28 in Algorithm 5.2 determines the best solution of the set of total solutions. This function traverses all found solutions and calculates the sum of the distances of the contained paths. The solution with the lowest distance is considered the best solution, as defined in Definition 5.2.

Definition 5.2 (Definition of the best solution)
Let $Solutions$ be the set of possible solutions for an $IoTEM$. A solution $solution \in Solutions$ is defined as the best possible solution, if:

$$\forall t \in Solutions : solution \neq t \Rightarrow \text{dist}(solution) \leq \text{dist}(t)$$

Since the backtracking algorithm calculates all possible solutions, its runtime becomes exponential. Therefore, the backtracking algorithm is recommended for small IoT environments and data stream processing models. If an IoT object from a cloud infrastructure is modeled in the IoTEM, the algorithm will use it as a fallback solution, thus, a solution can be ensured. Otherwise, the algorithms might not return a solution. In order to optimize the introduced algorithms, heuristics [Pea84] can be considered, which predefine a coarse mapping of specific operators. The introduced algorithms consider one heuristic, mapping the source operators onto the IoT object which is attached to the corresponding sensor to extract data from the hardware interfaces. In the future, more heuristics can be supported. For

example, if it is known that some sensors produce a large amount of data, e.g., video streams, operators processing this data can already be mapped onto powerful IoT objects, while low-resource IoT objects are not considered.

5.1.3 Case scenario: monitoring of mold levels in smart buildings

In the following, a case scenario in the smart building domain and how the mapping concepts are used to implement this scenario are presented.

A smart building is defined in this thesis as a building equipped with computing and information technology, which supports the needs of their occupants in order to provide them with a comfortable living.

The goal of this case scenario is to monitor mold levels in different rooms of a building in order to recognize when these levels rise to unhealthy ranges. For each room, the mold level is continuously calculated based on live measured temperature and humidity values. The mold levels are analyzed over time in order to detect any increase.

The smart building of this case study is composed of four rooms and a further room in the basement. Each room is provided with two Raspberry Pis as IoT devices, where one of them is connected to a temperature sensor and a humidity sensor. The second Raspberry Pi of each room provides additional computing resources if necessary. Furthermore, the basement room is equipped with an edge server as an IoT device, which has more computing capabilities than the other Raspberry Pis in the smart building. The involved IoT devices are connected to the same network, i.e., they are able to communicate with each other through standard internet protocols.

The described physical scenario is depicted in Figure 5.1 on the bottom. The IoTEM for this scenario is modeled based on the concepts presented in Section 4.1, while the data stream processing realizing the monitoring of mold levels is modeled based on the concepts presented in Section 4.2. The resulting DSPM is depicted in Figure 5.1 on the top.

In this scenario, the operators of the DSPM should be distributed among the IoT devices available in the building. The greedy variant of the matching algorithm is employed in order to decide where to deploy the operators.

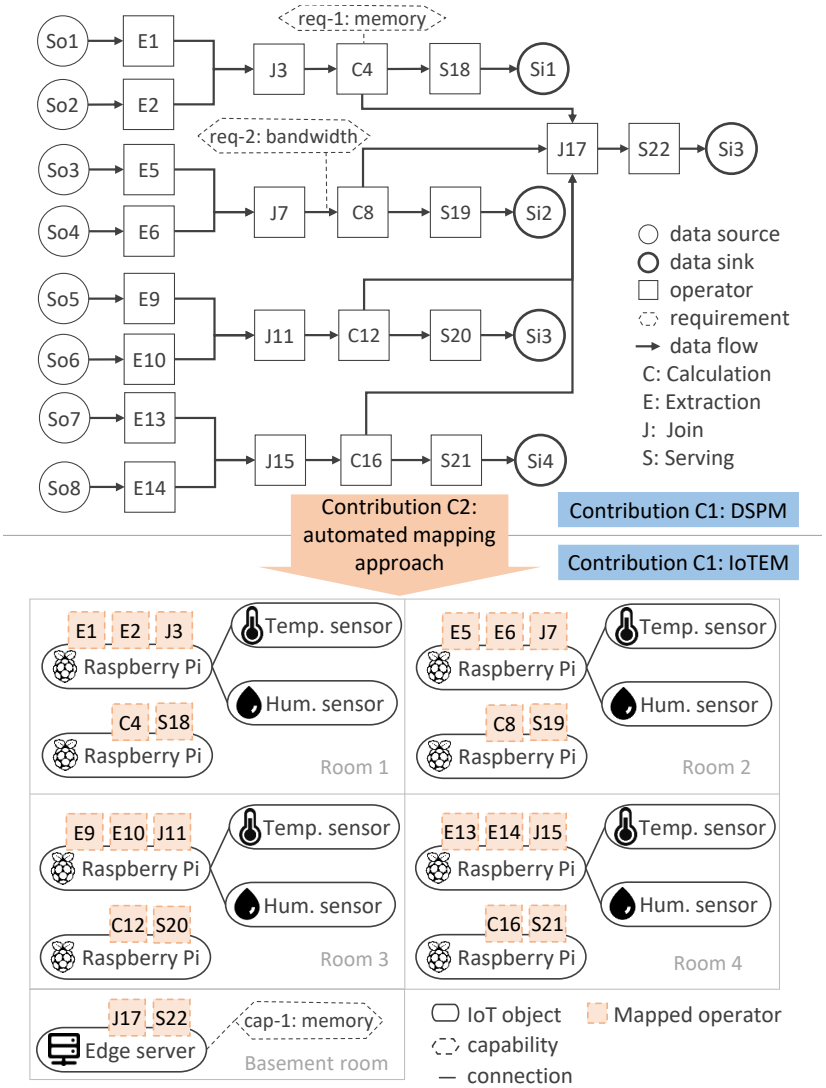


Figure 5.1: Case scenario: automated mapping in smart buildings

In a first step, the extraction operators, E1 through E14, are deployed onto the IoT devices, to which the sensors are physically attached to (as depicted in Figure 5.1). These operators contain, for example, scripts to extract the sensor data and send them to the next processing operator. Therefore, these extraction operators need to be deployed directly onto the IoT device attached to the sensors. Thus, it is a premise that there are enough resources available on these devices. After the extraction operators are mapped, their required resources are subtracted from the capabilities of the selected IoT devices in order to decide in a further step whether additional operations could be mapped onto these IoT devices. Afterwards, similar to extraction operators, the serving operators S18, S19, S20, and S21 are then mapped.

After the serving operators are mapped, the join operators are mapped next. For J3, which joins the output data from the extraction operators E1 and E2, the nearest device to the deployed extraction operators is searched within the IoT environment model. Obviously, the nearest device is the same device the extraction operators are deployed on. Assuming that this device fulfills the requirements attached to the join operation J3 and that there are enough capabilities left, thus, the join operator J3 can be deployed on this same device. This same procedure is done for the remaining join operators J7, J11, and J15. After that, the required resources are once again subtracted from the capabilities of the devices.

Next, the calculation operators C4, C8, C12, and C16 are mapped. For C4, again, the nearest device in the IoT environment model to its predecessor operator (i.e., the mapped operator J3) is searched. This is the device that already contains the mapped operators E1, E2, and J3. However, assuming that the calculation is based on a sophisticated algorithm to calculate the mold level consuming more resources, this device does not match the requirements of the C4 operator. Consequently, the next nearest device is searched, which is the one in the same room as the device containing E1, E2, and J3. Assuming that this device has enough computing capabilities to calculate the mold level, the operator C4 is mapped onto this device and its resources are consumed. In the remaining rooms, the operators are mapped similarly.

Finally, the calculated mold level of all four rooms is aggregated using operator J17, which produces output data suitable for a dashboard visualization for users. In this example, the IoT devices in different rooms have the same network distance to each other. Consequently, the algorithm selects them randomly. Next, it is checked whether an IoT device can fulfill the requirements of operator J17. If not, it checks the other devices. In this example, it is assumed that none of the IoT devices in the four rooms can fulfill the requirements of the operator J17. Consequently, J17 has to be mapped on another IoT device. In this case, the edge server in the basement room of the building is marked to execute this operator.

Since the explained concepts for the mapping are based on a generic method (cf. Figure 3.1), i.e., generic models are employed (IoTEM, DSPM), and furthermore, an extensible set of mapping algorithms is provided, this approach is transferable to other domains as well, such as the smart factory domain. This approach also scales for such large scenarios, i.e., the algorithms return a solution eventually.

5.2 Manual mapping approach

In this approach, a mapping plan is created manually by domain analysts, who decide the placement location explicitly for each operator involved in the data processing of IoT applications. That is, domain analysts are required to know whether the IoT objects of an IoT environment provide enough resources, and furthermore, meet the requirements of the operators of the IoT application. In this case, no separation of concerns can be guaranteed between technical knowledge and high-level domain knowledge.

This thesis employs the TOSCA standard for the creation of mapping plans manually. In [FBH+17; FBK+16; FHB+17], we have shown how TOSCA-based manual mapping plans for IoT scenarios in the smart home domain can be realized, and furthermore, how they can be, in a further step, deployed using a TOSCA-based deployment engine.

In the following, a manual mapping using the TOSCA standard is ex-

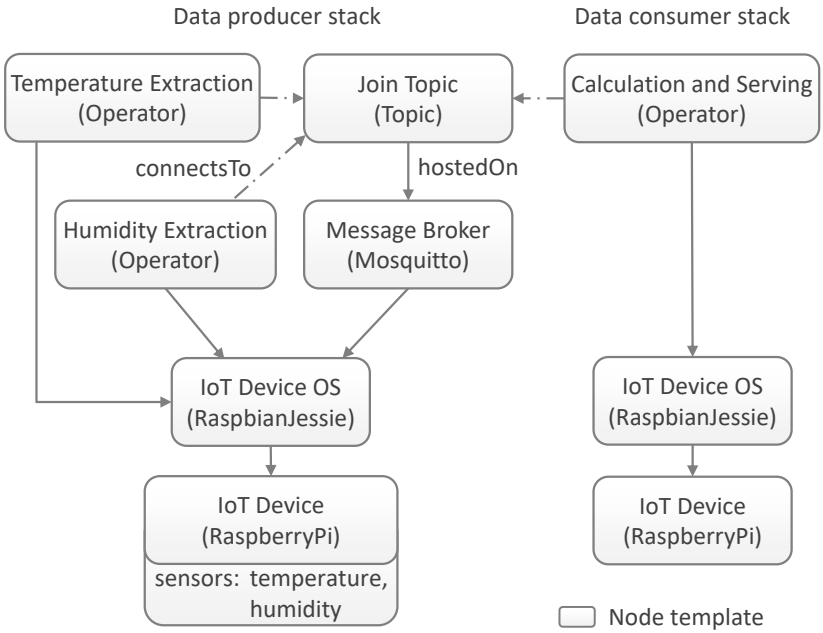


Figure 5.2: TOSCA topology model for a IoTEM and DSPM manual mapping (based on [FBH+17])

plained. The TOSCA-based mapping plan is created for the case scenario presented in Section 5.1.3, which aims at the monitoring of mold levels in smart buildings.

Figure 5.2 depicts an exemplary TOSCA topology, in which the IoT objects and the processing logic of the IoT application for the monitoring of mold levels in one single room are represented. For instance, IoT objects of an IoTEM are abstracted as TOSCA node templates, as shown in Figure 5.2 on the bottom. Furthermore, the processing operators of a DSPM are abstracted as node templates as well and are visualized in Figure 5.2 on the top.

The modeled TOSCA topology is divided into two stacks: a data producer stack and a data consumer stack. The data exchange among data producer and consumer is realized through the topic-based publish-subscribe pat-

tern [HW03]. In this communication pattern, a data producer publishes messages to a topic hosted on a message broker, which routes published messages to corresponding subscribers, i. e., data consumers.

Normally, a data producer stack contains the following components modeled as node templates: (i) one or more physical IoT devices, e.g. Raspberry Pis, smart watches, or smart phones, which can be embedded or attached with various sensors producing data, (ii) the operating system of the device, e.g., an operating system hosted on the IoT device, (iii) an operator that binds the sensors, able to extract their data and to hand them over, and (iv) the *Topic*, to which data is published and that is accessible by the data consumer stack. For devices that do not provide an accessible operating system, for example, a *smart watch*, a node template for the operating system is not required. In contrast, a remote *runtime* node template (e.g., an edge server) acting as a gateway needs to be provided, which enables placing an operator on it that connects to such devices and extracts data of corresponding embedded sensors. The described example in the following focuses mainly on IoT devices that provide an operating system, e. g., Raspberry Pis.

The data producer stack in Figure 5.2 encompasses an IoT device, which has attached temperature and humidity sensors, and the infrastructure providing access to these sensor values. For instance, temperature and humidity values are measured by sensors plugged into a Raspberry Pi, which executes two extraction operators to measure these values and send them to a topic-based message broker. In this case, temperature and humidity values are joined by sending both values to the same topic in the message broker.

Using the concept of *TOSCA node types*, a *Raspberry Pi* node type and its properties, e. g., the attached or embedded sensors, are modeled. Furthermore, the *IoT Device OS* node template specifies the operating system type of the IoT device being used, in this case, the *Raspbian Jessie* operating system. To properly model such an operating system, information needs to be provided about its type (e. g., Unix-based), how to access it (e. g., using SSH connections), and its access credentials, which could be through user authentication or based on a SSH key. Based on this operating system, operator scripts can be automatically deployed to access the hardware interfaces

of the sensors and extract their data.

The *Temperature Extraction* node template and the *Humidity Extraction* node template serve as interface between the physical IoT device, i. e., the Raspberry Pi, and the topic that is providing the data to consumers. Furthermore, the *Join Topic* node template needs to be modeled. This topic is hosted on a message broker software component represented by a corresponding *Message Broker* node template in the TOSCA topology. This message broker could be, for example, the Mosquitto Broker [Lig17], which is a messaging middleware based on the MQTT protocol. Finally, these extraction node templates are then connected to the *Join Topic* node template by a *connects to* relationship template.

The data consumer stack in Figure 5.2 encompasses an IoT device and the infrastructure to consume, process and monitor sensor values from the data producer stack. This stack provides a calculation and serving operator, which is executed on the IoT device. This operator subscribes to the message broker of the data producer stack in order to receive sensor values, and furthermore, to calculate the current mold level in the room based on the temperature and humidity sensor values.

Data consumers work in a similar fashion than data producers in the sense that they build on the publish-subscribe pattern. Consequently, for their modeling using TOSCA, the already introduced node types can be reused, i.e., the *Raspberry Pi*, the *RaspbianJessie* and the *Operator* node types. As depicted in Figure 5.2, three node templates are necessary: the IoT device, the operating system of the device, and the calculation and serving operator, which consumes and processes the data received through the *Join Topic* node template. Similar to the extraction node templates, the *Calculation and Serving* node template is also connected to the *Join Topic* node template by a *connects to* relationship template.

The data flow of the scenario is not explicitly modeled in the TOSCA topology, but rather implicitly implemented in the software artifacts of the operators. For instance, the data flow originates from the temperature and humidity sensors as data sources, passes through the topic hosted on the message broker, and reaches the calculation and serving operator hosted on

a Raspberry Pi, which represents a data sink.

In summary, in order to create a mapping plan in the form of a TOSCA topology model for such a case scenario, the above mentioned components need to be modeled and connected as described. However, the manual mapping approach is recommended only for use cases with a low amount of IoT objects or operators. That is, the manual creation of a mapping plan increases considerably the complexity of this task and becomes error-prone in larger scenarios, such as those in smart factories or smart cities.

5.3 Architecture component and implementation – IoTEM and DSPM mapper

In the overall architecture (cf. Section 3.3), the *IoTEM and DSPM mapper* component provides the means to support the decision about where operators should be executed based on their requirements. In this component, depicted in Figure 5.3, domain analysts have the possibility to perform the mapping of operators and IoT objects for the current DSPM and IoTEM using two approaches: (i) an automatic approach, in which a *mapping plan* is automatically generated by different algorithms, and (ii) a manual approach, in which domain analysts decide themselves by creating a mapping plan manually where operators should be executed.

In the automated approach, the *Mapping plan creator* (a) starts the *Matching algorithms* (b) that return a mapping plan. In the manual approach, domain analysts can create mapping plans themselves through the mapping plan creator. In this case, a modeler for TOSCA topology models, such as Winery [KBBL13], can be used to create a TOSCA-based mapping plan. Created mapping plans are stored in the *Mapping plan storage* (c). Once a mapping plan is created, the deployment of TOSCA-based mapping plans (cf. Chapter 6) can be started using a TOSCA-based deployment engine, such as OpenTOSCA [BBH+13], by triggering the *Deployment manager* component (cf. Section 6.4).

The *IoTEM and DSPM mapper* component has been prototypically imple-

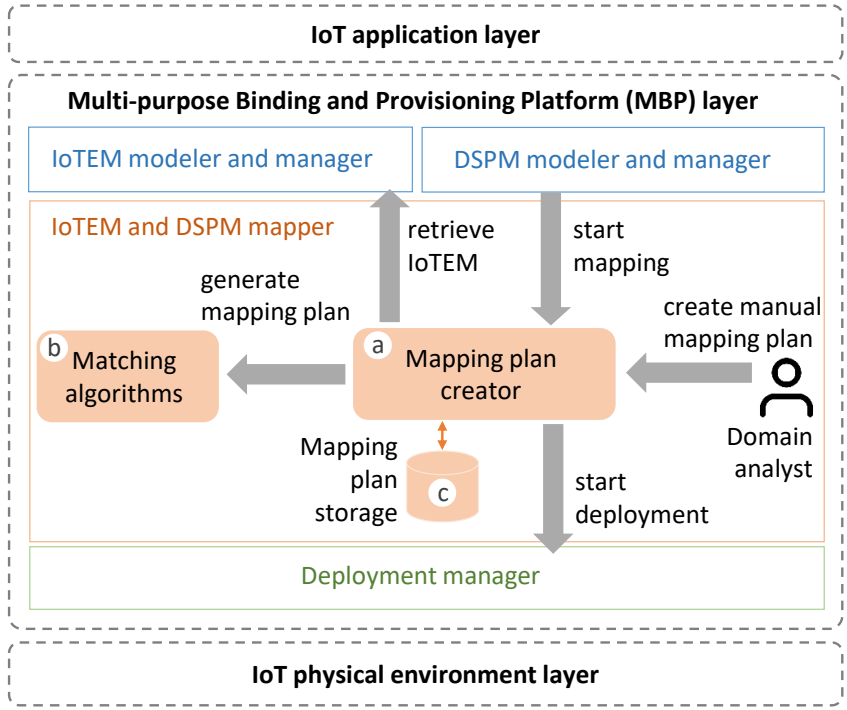


Figure 5.3: Architecture component: IoTEM and DSPM mapper

mented as part of the MBP. For the manual mapping approach, however, the *Mapping plan creator* corresponds to the external tool Winery[1]. In this case, the mapping plans are based on the TOSCA standard and are stored in a configured file system for Winery. For the automatic mapping approach, the Mapping plan creator and the Matching algorithms (the greedy variant and the backtracking variant) have been implemented in the server side of the MBP in Java in the scope of the bachelor thesis conducted by

[1] https://github.com/OpenTOSCA/winery

Schneider [Sch18]. As the *Mapping plan storage*, a MongoDB database is used to store created mapping plans. Section 8.1 presents the integration architecture of all architecture components and shows how they fit together.

5.4 Related work

Rizou et al. [Riz+10] present a heuristic algorithm that allows distribution of data stream operators in order to enable lowest possible latency for data processing. This algorithm searches for the best possible distribution to reduce the network latency to a minimum. The algorithm is executed directly on the available IoT objects, not centrally, in contrast to this thesis. However, the approach of Rizou et al. requires that the IoT objects in the network are known, connected, and able to communicate with each other, e.g., to share information about latency. The approach in this thesis rather aims for a loose coupling between the IoT objects. In the best case, the IoT objects do not have to know each other and, consequently, do not have to communicate directly. Furthermore, the approach of Rizou et al. only considers latency as a requirement for data processing, therefore, other requirements, as described in this thesis, are not supported.

Cipriani et al. [CSM11] provide an approach for the operator placement of stream processing queries, in which specific objectives (i.e., requirements) of stream-based applications are taken into consideration to realize necessary placement decisions. This approach is called Multi-target Operator Placement of Query Graphs for Data Streams (M-TOP). Similar to this thesis, M-TOP enables the annotation of requirements onto processing operators. These requirements need to be fulfilled by the available computing nodes (i.e., IoT objects) in order to achieve a suitable operator placement. However, in contrast to the approach of this thesis, M-TOP does not take post-deployment operator placement into consideration, i.e., monitoring of the deployed stream processing is not available [Cip14; CLM10; CSM11]. Furthermore, the approach in this thesis also considers application-based and user-based requirements that emerged with the upcoming of the IoT

domain. Thus, new kinds of IoT hardware objects that are eligible to be computing nodes are considered as well.

Bumgardner et al. [BHM18] present a graph-based modeling language named Cresco Application Model (CAM), in order to describe the logic of distributed stream-based applications. The CAM models are similar to the DSPMs presented in this thesis. Furthermore, Bumgardner et al. search for an optimal distribution of operators onto IoT objects through a greedy algorithm. The presented algorithms in contribution C2 of this thesis, furthermore, optimize the distribution based on the network distance in order to enable short communication paths.

Chapter 6

Deployment of operators onto IoT environments

This chapter introduces contribution C3, which handles the deployment of operators onto IoT environments. This contribution contains two approaches to realize the deployment: (i) an *automatic deployment approach*, in which the operators of a DSPM are deployed automatically based on the *mapping plan* generated in contribution C2, and (ii) a *semi-automatic deployment approach*, which is also based on the mapping plan, however, manual actions, called *human tasks* [OAS10], are supported. The concepts of contribution C3 employ the TOSCA standard [OAS13] and established software deployment tools, such as Shell scripts.

The deployment approaches are explained in detail in Sections 6.1 and 6.2. In Section 6.3 the Topic Description Language for the IoT (TDLIoT) is introduced, which is used in this thesis by the deployment approaches. In Section 6.4, the architecture component responsible for the deployment is described. Finally, Section 6.5 presents related work to contribution C3.

6.1 Automatic deployment approach

The *mapping plan* resulting from contribution C2 is defined as a *solution* for the DSPM operator placement problem (cf. Definition 5.1). It consists of the *operator mapping* assigning operators of a DSPM to IoT objects of the IoTEM. Based on the operator mapping, operators are iterated and each of them is deployed automatically using the TOSCA-based deployment engine OpenTOSCA [BBH+13]. Each operator can reach several states, which reflect the current deployment state of an operator. These states are explained in the following.

6.1.1 Deployment states of an operator

This thesis defines six deployment states of an operator: (i) unconfigured, (ii) configured, (iii) started, (iv) published, (v) stopped, and (vi) terminated. These operator states are depicted in Figure 6.1.

The initial state of an operator is *unconfigured*. Once the deployment starts, the first task is to *configure* the assigned IoT object for the execution of the operator. For this, the operator in the form of software artifacts is copied to the assigned IoT object. Furthermore, required software dependencies (e.g., MQTT client, Python interpreter) for the operator are installed and configured on the assigned IoT object. After the configuration, the operator transitions to the state *configured*.

A configured operator can be *started*, which is the main goal of the automatic deployment. Once all operators in the operator mapping reach the state *started*, the automatic deployment is considered as finished and, consequently, the operators of the IoT application are set up and running. Furthermore, a configured operator can be directly terminated as a rollback mechanism in case the operator could not be started.

Once an operator is started, its output, e.g., extracted sensor data, can be made available for external applications. This task is conducted by domain analysts, which decide if and which operators should be *published*. For this, descriptions of how to access and parse the output of the operators are

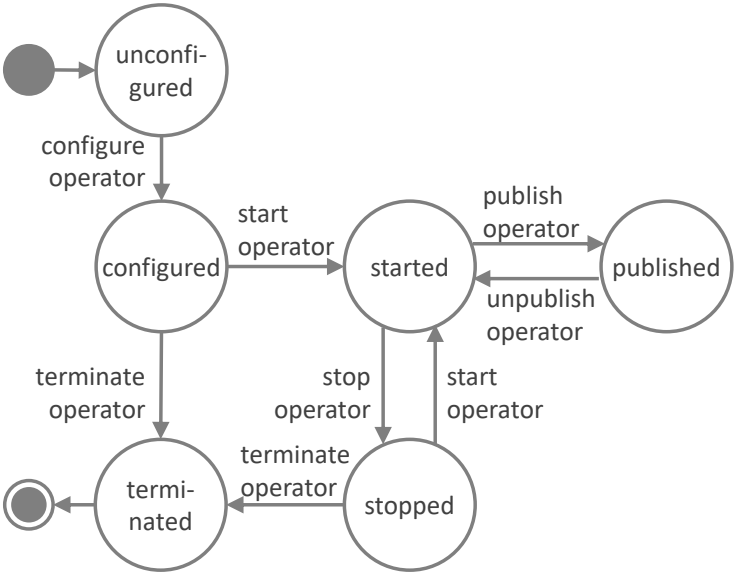


Figure 6.1: Life cycle of an operator

automatically created. Such descriptions and corresponding concepts for their management are explained in detail in Section 6.3.

Finally, when the data stream processing is retired, e.g., by domain analysts, each of the previously deployed operators needs to be retired. For this, each started operator is *stopped* and, subsequently, *terminated*. If the operator was published, it is unpublished, stopped and terminated. By terminating an operator, all software artifacts that were copied to the IoT object are deleted and dependencies are uninstalled.

6.1.2 TOSCA-based operator deployment

In [FBH+17; FBK+16; FHB+17], we show how to use the Winery modeling tool [KBBL13] to model TOSCA topologies containing several operators. Furthermore, we also show how to use the OpenTOSCA runtime [BBH+13] to automatically deploy the modeled operators onto IoT objects.

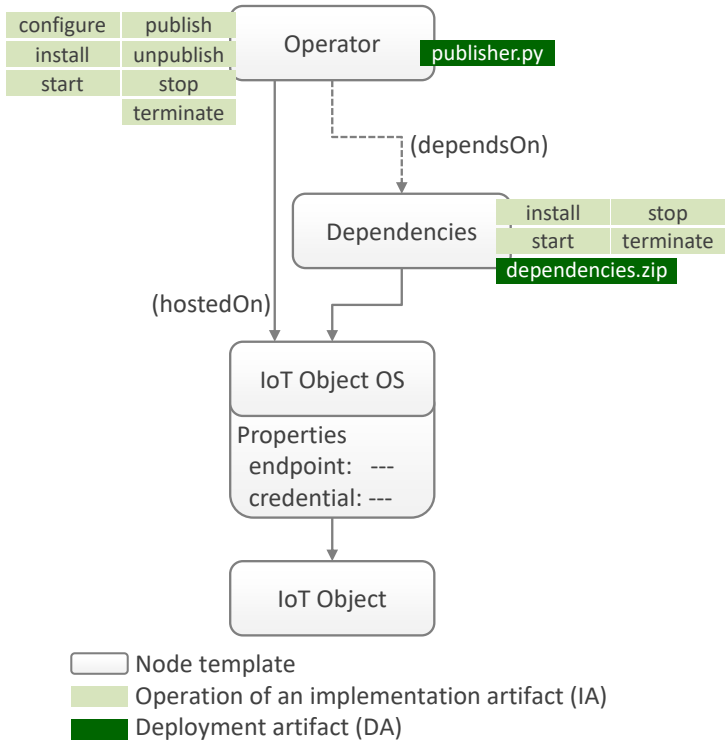


Figure 6.2: TOSCA topology model for an operator

Figure 6.2 depicts a TOSCA topology model, which is used in this thesis as a basis template for the deployment of each operator in a mapping plan. This topology does not yet contain all necessary information for the deployment, and needs to be extended with specific information of each operator in the mapping plan during the deployment. The node template *Operator* defines a life-cycle interface containing operations corresponding to the operator state transitions defined in this thesis (cf. Figure 6.1). These operations are *configure*, *install*, *start*, *publish*, *unpublish*, *stop*, and *terminate*. Furthermore, the node template *Dependencies* also defines a life-cycle interface for the dependencies of the modeled operator.

To deploy the operators with the OpenTOSCA runtime, the operators in the mapping plan are iterated and this topology is copied and extended with the specific details of each operator. Software artifacts and dependencies of an operator are programmatically added to the topology as *implementation artifacts* and *deployment artifacts* of the node templates *Operator* and *Dependency*. Furthermore, specific properties of the mapped IoT object are also added to the topology. Finally, the extended topologies can be deployed sequentially or in parallel by several OpenTOSCA engine instances.

6.2 Semi-automatic deployment approach

As introduced in Section 6.1.2 and in [FBH+17; FBK+16; FHB+17], OpenTOSCA can be used to deploy operators onto IoT environments in a fully automated manner based on topologies. These approaches assume that the IoT objects in the IoT environment are already configured for the deployment of the operators. However, manual actions, called *human tasks*, are usually required to be conducted, in order to deploy operators of IoT applications, e.g., plugging in sensors.

In this thesis, the concept of human tasks is based on the OASIS specification *WS-HumanTask* [OAS10]. Human tasks are activities that need to be conducted by people and, therefore, cannot be conducted automatically. Examples of human tasks include the approval of certain processes, e.g., for buying expensive hardware or to grant a loan. Further examples in the context of IoT environments are plugging in sensors to devices or flashing microcontrollers. In conclusion, a human task is a request to a person to perform a specific activity.

To be more tailored for the challenges presented in IoT environments, this thesis provides a semi-automatic deployment approach, in which human tasks are also considered for the deployment. For this, this thesis enables domain experts to define human tasks and add these to mapping plans before the deployment starts. Furthermore, by supporting human tasks, it increases the reusability of mapping plans for different, however, similar

IoT environments, e.g., different smart buildings composed of identical IoT objects. In this case, specific property values of IoT objects, such as IP addresses, do not need to be modeled in the IoTEM, and can be filled by a human task during the deployment.

When the deployment of a DSPM is triggered, the *mapping plan* is received as input for the deployment. This mapping plan is searched for *human task definitions* and for implicit *patterns* implying that human tasks are required. An example of such a pattern is an empty property value in the description of an IoT object. In this case, a human task definition is automatically created to fill empty property values.

In case the mapping plan is considered complete, i. e., the mapping plan does not contain human task definitions or patterns, the deployment is performed as explained in Section 6.1.

Finally, when the search is done, a list of human task definitions is sent to a middleware component, called *Human task manager*. This component notifies users (e.g., domain experts) about the human tasks to be conducted and also notifies deployment engines about the status of human tasks. Within the scope of this thesis, a bachelor thesis was conducted by Kutger [Kut18], which presents an approach for enabling the OpenTOSCA engine to support human tasks.

6.3 Topic Description Language for the IoT

The following section is mostly based on [FHB+18].

In [FHB+18], we introduce the *Topic Description Language for the Internet of Things* (TDLIoT) approach, which provides simple means to describe and find topics for public sensors and actuators, and furthermore, for published operators as defined in this thesis. The TDLIoT notation is derived from an extensive literature review and experiences on different IoT platforms, protocols, and applications. The TDLIoT approach is used in this thesis to create descriptions that aim to ease the access to the output data of published operators (e. g., extracted sensor data) by further IoT applications.

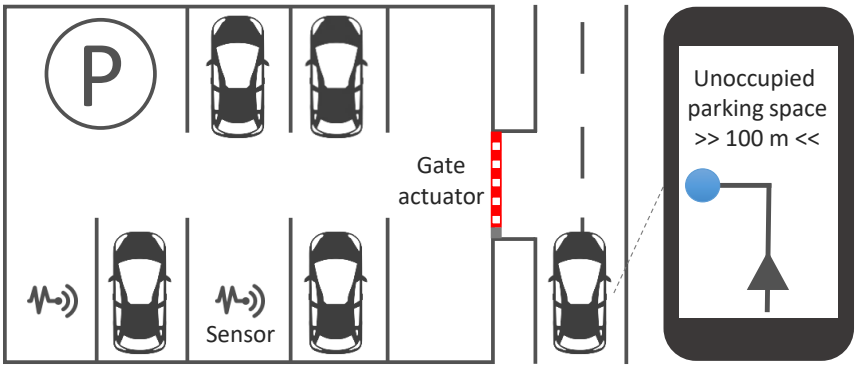


Figure 6.3: Case scenario: parking in smart cities (based on [FHB+18])

A case scenario for such an IoT application is the live detection of unoccupied parking spaces in a smart city, so that drivers nearby are notified on their smart phones about unoccupied parking spaces. This case scenario is depicted in Figure 6.3. It can be based, for example, on parking space sensors, which detect the availability of parking spaces. Furthermore, once parking spaces are available, drivers could trigger actuators through their smart phone applications, for example, to open gates to the parking areas.

However, in order to develop such an IoT application, developers needs to know (i) which sensors and actuators (abstracted as published operators) exist in the context of the application, (ii) what data sensors provide and in which format, (iii) which actions can be triggered for actuators, and (iv) how sensors and actuators can be accessed to be used.

In the IoT, the access to sensors and actuators is usually realized through the publish-subscribe communication model [HW03] based on *topics*. These topics are used to provision sensor values or enable access to actuators. In the scope of this thesis, a topic is an entity that allows to receive and send data in a uniform manner. Topics could be realized through various communication models, such as publish-subscribe or request-response, using different protocols. In the topic-based publish-subscribe model, e. g., realized through MQTT [BG14], subscribers register on topics relevant for their

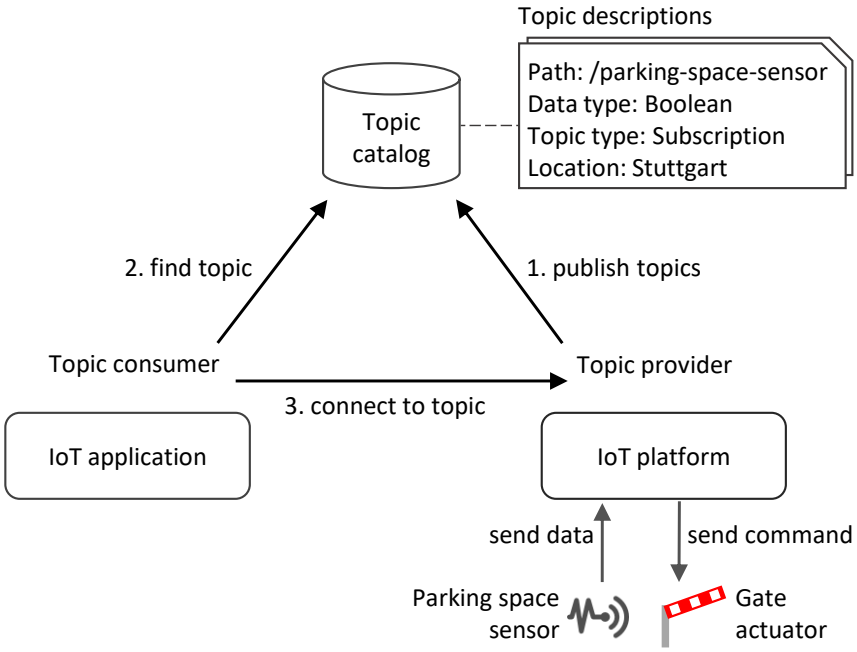


Figure 6.4: Overview of the TDLIoT approach for topic description and retrieval [FHB+18]

context to get asynchronously notified when publishers send messages to these topics [EFGK03]. In the request-response model, instead of getting notified, applications need to request the data, e.g., through HTTP requests.

The management of subscriptions and delivery of messages is usually realized by IoT platforms, such as FIWARE [RGSE14], Mosquitto [Lig17], OpenMTC [WMS12], or RMP [HWBM16a]. For example, given that sensor values are provided through topics hosted on such IoT platforms, using the publish-subscribe model, a smart city parking application subscribes to topics of all sensors monitoring parking spots and gets notified once a parking spot is detected as *unoccupied*.

Application developers depend on knowing everything about the topics

they can use to build such IoT applications. This information, which includes, for example, the data structure or how it can be accessed, is usually only known if the application developers own the involved IoT objects. Other available topics, which could provide additional information about the environment, are oftentimes not considered but could lead to a significant improvement of the application, e. g., through a higher coverage by additional sensors. For instance, there exist publicly accessible IoT data available on platforms, such as dweet.io[1], however, details about the data content, e.g., the data structure and how to interpret them, are not provided. This highly increases the effort to develop IoT applications using such IoT data.

Therefore, the TDLIoT approach provides simple means to describe and find topics of published operators, and furthermore, data sources and sinks abstracted from sensors and actuators. The TDLIoT approach provides (i) a holistic description of the topics, (ii) a topic catalog to browse the topic descriptions, and (iii) an effective way to find suitable topics that offer access to sensors and actuators abstracted as data sources and sinks. In this way, IoT application development can be eased through an abstraction from specific IoT platforms. The topics in the catalog can be searched by, for example, a specific location or sensor type.

The TDLIoT approach, which is depicted in Figure 6.4, is composed of three main roles: the *topic provider*, the *topic consumer*, and the *topic catalog*. The topic provider creates *topic descriptions* based on the TDLIoT and publishes them to the topic catalog. The topic consumer searches for interesting topic descriptions in the topic catalog, and directly connects to topic providers to either publish data or receive the data published to subscribed topics. The TDLIoT approach has been prototypical implemented and is available as an open-source GitHub project[2].

In the scope of this thesis, a TDLIoT topic catalog is provided, in which descriptions for operators are stored. This catalog can, therefore, be searched by external developers in order to create further IoT applications requiring publicly accessible data, e.g., in the smart city domain [FHB+18].

[1] http://dweet.io
[2] https://github.com/IPVS-AS/TDLIoT

A TDLIoT-based topic description must contain at least the following attributes, however, it can be extended with further attributes when necessary to describe a topic in more detail:

- **data type**: the type of the values provided by the topic, e.g., Boolean.
- **hardware type** (optional): the type of hardware represented by the topic, i.e., a specific sensor or actuator.
- **location**: the location of the sensor or actuator represented by the topic. It contains the *location type*, e.g., GPS or a specific city name, as well as the *location value*, e.g., specific *GPS coordinates*.
- **message format**: the format of the message provided by the topic, e.g., JSON, YAML, or XML.
- **message structure**: the structure of the message defined as metamodel to understand its content. It contains the *metamodel type*, e.g., *JSON schema* or *XML schema* and the specific *metamodel*.
- **platform endpoint**: the endpoint of the IoT platform hosting the topic, e.g., the endpoint of a message broker running on a server.
- **owner**: the name of the topic provider.
- **path**: path of the topic.
- **protocol**: the communication protocol being used, e.g., MQTT or HTTP.
- **topic type**: the type of the topic, i.e., *subscription* or *command*.
- **unit** (optional): the unit of the data provided through the topic, e.g., *Celsius* if the topic provides temperature values in °C.

In Figure 6.5, the aforementioned attributes are visualized in the TDLIoT metamodel as entity-relation model in the notation of Chen [Che76]. This metamodel describes the relations between the entities of TDLIoT descriptions. With exception of the *hardware type* and *unit* attributes, all of its entities must occur only once within a single TDLIoT description. However,

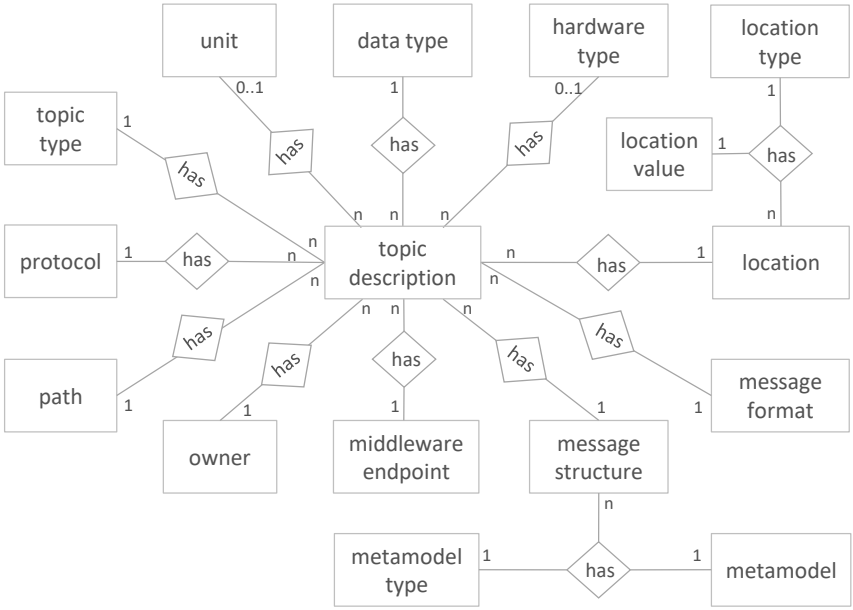


Figure 6.5: Data model associated with the TDLIoT [FHB+18]

these entities can occur in an arbitrary amount of descriptions. In case topics provide aggregated data, for example, originating from different sensors, or data that do not come from hardware at all, the *hardware type* attribute is not required.

To prevent conflicts regarding the values of the TDLIoT attributes, ontologies can be used to detail the semantic description of these values, such as a specific hardware type having more than one name. For example, the type *parking-space-sensor* could also be named *parking sensor* or *PSensor* in different topic descriptions.

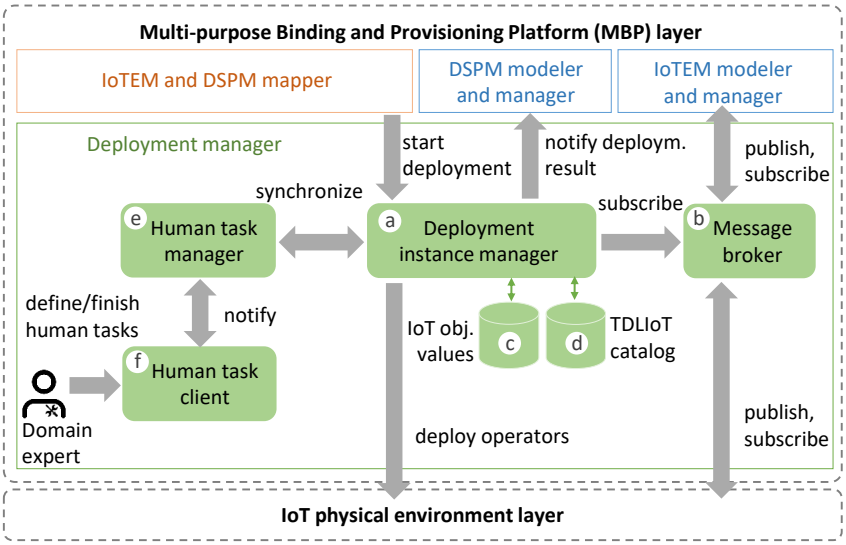
Grangel-Gonzalez et al. [GHC+16] introduce such an ontology-based vocabulary for the IoT based on ontologies, which can be used as foundation for the TDLIoT. Furthermore, other semantic relations can be expressed through the ontology, for example, whether the represented hardware is a sensor

```
1  { "data_type": "boolean",
2    "hardware_type": "parking-space-sensor",
3    "location": {
4        "location_type": "city_name",
5        "location_value": "Stuttgart"
6    },
7    "message_format": "JSON",
8    "message_structure": {
9      "metamodel_type": "JSON_schema",
10     "metamodel":"{"title": "provider_schema",
11                   "type": "object",
12                   "properties": {
13                      "value": {"type": "boolean"},
14                      "timestamp": {"type": "integer"},
15                   "required": ["value", "timestamp"]}"
16   },
17   "platform_endpoint": "http://example.com",
18   "owner": "university_of_stuttgart",
19   "path": "parking-space/operator/id7321/output",
20   "protocol": "MQTT",
21   "topic_type": "subscription"
22 }
```

Listing 6.1: Example of a TDLIoT description in JSON (based on [FHB+18])

or actuator. Using such a vocabulary enhances the TDLIoT with semantics, i.e., the meaning of attribute values can be understood through reasoning approaches. With this semantic information, finding topic descriptions for specific use cases can become easier.

Listing 6.1 shows an example in JSON format of a TDLIoT description to access the output data of an extraction operator. In this example, the output can be accessed through the message topic *parking-space/operator/id7321/output* (line 19) hosted on an MQTT message broker (line 20). The format and structure of the messages provided through this topic are defined in lines 7 to 16. That is, the message is formatted in JSON and the message structure is specified by the JSON schema in lines 10 to 15.



IoTEM: IoT environment model
DSPM: Data stream processing model
TDLIoT: Topic description language for the IoT

Figure 6.6: Architecture component: Deployment manager

6.4 Architecture component and implementation – Deployment manager

In the overall architecture (cf. Section 3.3), the *Deployment manager* component provides the means for the deployment of operators onto IoT objects. This component is depicted in Figure 6.6. The deployment of operators of a DSPM can be started through the *Deployment instance manager* ⓐ, which receives as input the *mapping plan* created via mapping plan generation (cf. Section 5.1) or created manually (cf. Section 5.2). Furthermore, once the deployment of DSPMs is completed, running instances of the DSPMs (dDSPMs) are created and managed by the Deployment instance manager. These running instances are forwarded to the *DSPM modeler and manager* component (cf. Section 4.2.3), which gets informed about the success of

the deployment of DSPMs, in order to provide visual feedback to domain analysts in the DSPM modeling tool.

In the automatic deployment approach, operators and their required software artifacts are installed, configured, and started onto the mapped IoT objects. Further software artifacts to collect empirical values and monitoring information about the operators and the IoT objects are also deployed. Once the execution of operators is started on IoT objects, they publish data, e. g., sensor values or monitoring information, to the *Message broker* (b) component. The Deployment instance manager subscribes to the Message broker, in order to continuously receive IoT object values and monitoring values, which are stored in the *IoT object values* (c) repository. Descriptions of how to access and parse the output of published operators are stored in the *TDLIoT catalog* (d). This catalog also provides a REST API to search for publicly accessible operators, e.g., which extract values of public sensors.

In the semi-automatic deployment approach, the *Human task manager* (e) infers manual actions (i.e., human tasks) from the mapping plan, and notifies the *Human task client* (f) application about their existence. Once the human tasks are completed by domain experts input in the Human task client, the Deployment instance manager is notified to either start or continue the deployment of the operators.

The *Deployment manager* component, has been partly implemented in the MBP as a prototype and also uses existent external implementations. The OpenTOSCA container [BBH+13] is used as an external *Deployment instance manager*, which is able to automatically deploy operators of mapping plans based on the TOSCA standard. This external tool is available in the GitHub project of the OpenTOSCA Ecosystem[1]. Within the scope of this thesis, Kutger [Kut18] conducted a bachelor thesis, which implemented the *Human task manager* and the *Human task client*, in order to enable the OpenTOSCA container to support the deployment of operators involving human tasks. The *Human tasks manager* is based on the implementation from Wagner [Wag10], which corresponds to a Java web application running

[1] https://github.com/OpenTOSCA

on Apache Tomcat[1]. The *Human task client* was implemented as an Android-based mobile application[2].

Additionally, the MBP provides several ready-to-use extraction and control operators as Python and Shell scripts[3], which can then be registered in the MBP and be deployed onto IoT objects by the MBP. Once operators are deployed, they send values to the MBP by connecting and publishing to the *Message broker*, which is implemented using the MQTT broker Mosquitto. The MBP can visualize sensor values as line diagrams in the MBP dashboard. The data management and processing in the MBP implements a lambda-based architecture [MW15], in which both live and historical data can be managed and processed. To store measurement data, i.e., timestamp-based sensor data, the time-series database InfluxDB is used. Further metadata, IoT environment models, and data stream processing models are stored in the NoSQL database MongoDB, to clearly separate measurement data and metadata. The *TDLIoT catalog* has been prototypically implemented outside the MBP as an open-source GitHub project[4]. It is implemented in Java and uses a MongoDB database to store TDLIoT descriptions. It provides a web interface implemented in JavaScript and a REST API for interactions with the TDLIoT catalog. Finally, Section 8.1 presents the integration architecture of all architecture components and shows how they fit together.

6.5 Related work

This section describes related work regarding software deployment onto IoT environments. Li et al. [LVCD13] propose to employ TOSCA to specify the basic components of IoT applications (e.g., gateways, micro controllers) and their configuration, in order to automate IoT application deployment in heterogeneous environments. Extensions of this work were presented by Vögler et al. [VSI+15; VSID16]. The authors propose the framework

[1] https://github.com/IPVS-AS/Human-Task-Manager
[2] https://github.com/IPVS-AS/Human-Task-Client
[3] https://github.com/IPVS-AS/MBP/tree/master/resources/operators
[4] https://github.com/IPVS-AS/TDLIoT

LEONORE for the deployment and execution of custom application logic directly on IoT gateways. However, for the framework to know the available IoT gateways for provisioning, the IoT gateways must have a pre-installed local provisioning agent. This agent registers itself to the framework by providing its unique identifier and profile data of the gateway (e.g., MAC address, instruction set, and memory consumption). In contrast, the approach of this thesis does not require pre-installed components on IoT objects, however, it requires activated remote access to IoT objects, so that necessary components can be installed automatically.

Hur et al. [HCJL15] propose a Semantic Service Description (SSD) ontology and a system architecture to automatically deploy IoT devices to heterogeneous IoT middlewares, aiming to solve interoperability problems between them. This thesis also aims to tackle interoperability problems, however, it proposes a standard-based approach using TOSCA to automatically deploy computing operators onto heterogeneous IoT environments.

Hirmer et al. [HBF+16; HWBM16a] introduce an approach for automated binding of IoT devices using a middleware called *Resource Management Platform* (RMP). The RMP enables an easy registration of IoT devices and their binding through adapters. An adapter is a piece of code containing the logic to read sensor values of IoT devices, to send the sensor values to the RMP, and to invoke actuators. For the binding, adapter scripts are automatically deployed onto the IoT devices. An extension of the RMP, which uses TOSCA to deploy the adapters, is also described by Hirmer et al. [HWBM16b]. In contrast to this thesis, Hirmer et al. concentrate on the binding of hardware devices whereas the approach of this thesis additionally deals with the deployment of computing operators onto whole IoT environments.

The automated deployment onto IoT environments can be realized by several approaches, e.g., by using Shell scripts, Chef, or Puppet. However, in contrast to these approaches, the standard-based approaches in this thesis are using TOSCA and enable a generic approach based on topology models and a corresponding graphical notation [BBK+12]. These topology models are highly adaptable in a way that single software components of a topology

can be easily interchanged, and furthermore, extended. In other approaches, this requires a large adaptation effort, e.g., when editing Chef scripts. In addition, through the concepts of node and relationship types, TOSCA offers a high abstraction level that supports reusability for software deployment.

Chapter 7

Monitoring of deployed DSPMs

This chapter introduces contribution C4, which is the monitoring of deployed data stream processing models (dDSPM). For this, contribution C4 provides the means to continuously monitor IoT objects in IoT environments and dDSPMs, in order to recognize disturbances. This monitoring is realized using complex event processing (CEP) techniques [Luc01], which are well-established for the processing of data streams to timely recognize situations [BD15; BK09; FHWM16].

The modeling of disturbance recognition is explained in Section 7.1. How the execution of disturbance recognition is realized is explained in Section 7.2. In Section 7.3, the architecture component and implementation for the monitoring of dDSPMs is described. Finally, Section 7.4 presents related work to contribution C4.

7.1 Modeling of disturbance recognition

Context-aware applications have gained a lot of attention in recent years, especially in the IoT domain. Deriving high-level context from low-level, raw sensor data enables automated adaptation and realization of self-organized IoT environments. However, there are a lot of challenges to be addressed regarding the acquisition, modeling and management of context information [GBH+05; LCG+09]. A further challenge is how to achieve efficient processing of large amounts of sensor data and, consequently, good quality of the derived high-level context knowledge. For this, well-established techniques, such as complex event processing (CEP), have been introduced to process large amount of data in a timely fashion [BD15; BK09].

This thesis uses the definition provided by Dey and Abowd [Dey01] describing *context* as *"any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or object"*. As a consequence, such context information can be used to derive high-level context information, called situations. In the scope of this thesis, a *disturbance* is considered as a *situation*, which is defined as an occurrence that might require correcting actions [Luc01]. Therefore, disturbance recognition is also referred to as *situation recognition* in this thesis.

Hirmer et al. [HWS+15] propose *SitRS*, a cloud-ready situation recognition service that enables situation recognition based on raw sensor data. In their approach, sensors are bound dynamically and the sensor data is received in a pull-based manner. The situation recognition is executed in fixed time intervals by pulling the sensor data and deriving situations and, thus, is suitable only for simple situation recognition scenarios. In [FHWM16], we introduce *SitRS XT*, an efficient, scalable situation recognition service. SitRS XT extends the approach from Hirmer et al. [HWS+15] by enabling a continuous processing of sensor data through a stream-based approach using CEP technologies. Thereby, the modeling and execution of more complex situation recognition scenarios are enabled.

This thesis employs SitRS XT to recognize disturbances on deployed DSPMs onto IoT environments. SitRS XT processes monitoring data on different

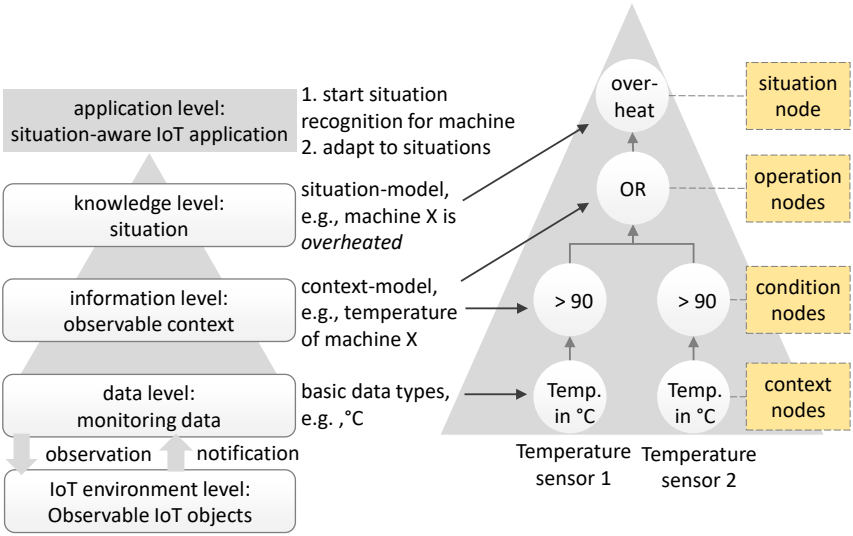


Figure 7.1: Data processing levels (based on [FHWM16])

levels, which are depicted in Figure 7.1 on the left. On the *data level*, only raw monitoring data (e.g., temperature sensor values) is available. This data is pushed to the *information level*, to be enhanced with information about relations of raw data to real-world things (e.g., a production machine), becoming, in this way, information about the IoT environment. Based on the observable context, monitoring data is aggregated and interpreted in order to derive situations, which leads to *knowledge* about the IoT environment. This high-level knowledge is crucial for situation-aware applications, since it can be processed on a higher-level of abstraction.

In SitRS XT, situations are modeled as *situation templates* [HHL+10], a domain-specific model that abstracts from complex, technical details. This model contains the monitored IoT objects and the conditions that have to match for a certain situation to be recognized. Situation templates are based on situation aggregation trees (SAT), which are directed, cohesive graphs as introduced by Zweigle et al. [ZHKL09]. In SATs, sensors correspond to leaf

nodes called *context nodes* and branches are aggregated bottom-up through a combination of so-called *condition nodes* and *operation nodes* until the root node (i.e., situation node) is reached.

A simple example of a situation template is depicted in Figure 7.1 on the right. It defines the conditions to recognize when the temperature of a production machine passes the threshold of 90 Celsius degrees. The nodes of a situation template reflect the aforementioned processing levels: *context nodes* represent the sensors monitoring a certain IoT object, which correspond to the data level. Context nodes are connected to *condition nodes*, which establish the relations of sensor data to IoT objects (information level). Furthermore, condition nodes filter monitoring data based on the defined conditions. Condition nodes can be aggregated by *operation nodes* using logical operations until the situation node is reached, which represents the situation to be recognized. The combination of condition, operation and situation nodes corresponds to the knowledge level, where sensor data is aggregated, interpreted, and derived to situations.

Defining situations using situation templates releases domain experts of creating complex, executable representations, such as CEP queries. Such representations are, however, still required for the deployment in execution environments. The manual creation of such complex representations requires expert knowledge and, thus, is time-consuming and error-prone. Therefore, SitRS XT reduces this complexity by automatically transforming situation templates onto the required executable representations, i.e., CEP queries. The transformation of situation templates onto CEP queries, and consequently, the execution of disturbance recognition based on these executable representations are explained in Section 7.2.

In the aforementioned SitRS XT approach for disturbance recognition, it is assumed that the resulting CEP queries are to be executed only on monolithic IT infrastructures. However, in order to process data more efficiently, further approaches are required to distribute the CEP queries within the IoT environment, so that a distributed data processing with short communication paths and reduced network traffic is enabled. Therefore, we introduced

in [FHKM18] an approach for CEP query shipping onto IoT environments, so that the execution of all required CEP queries only on monolithic IT infrastructures can be avoided.

In the following, a case scenario based on the work of Hoos et al. [HHM17] is presented, in which several CEP queries can be shipped (i.e., deployed) onto different execution locations. The goal of this case scenario is the monitoring of a production step on the shop floor of a manufacturing company, in order to recognize disturbances in the production step as soon as possible.

In this production step, a shop floor worker inserts a metal part into a machine, which cuts it into the required form. In this process, two problems could occur: (i) the tool cutting the metal gets decayed over time, or (ii) the metal part is wrongly placed into the machine, so that the metal cannot be cut correctly. Both these cases lead to an erroneous end product, which sometimes is discovered late in the production process. This could cause high costs because many steps need to be repeated or products even need to be thrown away. Consequently, those error cases need to be recognized immediately in order to save costs.

In this case scenario, there are two data sources, (i) a position sensor and (ii) a tool condition sensor. The position sensor returns a Boolean value indicating whether the metal part is correctly placed into the machine. In case of *true*, the metal part is in the correct position. In case of *false*, the metal part is in a wrong position. The tool condition sensor returns the cutting tool's condition as percentage value, where *0%* means the tool is fully decayed, and *100%*, the tool is unused. Furthermore, to realize this case scenario, three different CEP queries can be created, so that they can be executed on different locations. However, the CEP engine that will execute these CEP queries needs to be either distributed, such as presented by Cugola et al. [CM13], or be provided as different instances of the same CEP engine. In this case scenario, the second option is assumed.

In Listing 7.1, the CEP query Q1 is depicted in the Event Processing Language (EPL) syntax [Esp06b]. This query uses the values of a position sensor as its input and creates an output event if the sensor produces the value *false*. For this, a schema for an input event stream is defined with

name *PositionSensorStream* in line 1. Likewise, a schema for an output event stream with name *ProductionErrorStream* is defined in line 2. Subsequently, the CEP query Q1 is defined in lines 4 and 5.

```
1 create schema PositionSensorStream(value boolean);
2 create schema ProductionErrorStream(value boolean);
3
4 @Name('Q1') insert into ProductionErrorStream
5   select * from PositionSensorStream(value=false);
```

Listing 7.1: CEP query Q1 in EPL syntax [FHKM18]

Another CEP query (Q2), depicted in Listing 7.2, uses the values of the tool condition sensor as input and produces an output event when the tool condition becomes less than *80%*.

```
1 create schema ConditionSensorStream(value integer);
2 create schema ProductionErrorStream(value boolean);
3
4 @Name('Q2') insert into ProductionErrorStream
5   select * from ConditionSensorStream(value<80);
```

Listing 7.2: CEP query Q2 in EPL syntax [FHKM18]

The CEP query Q3, depicted in Listing 7.3, serves as an aggregation query, which uses the output events, produced by the queries Q1 and Q2, as input.

```
1 create schema ProductionErrorStream(value boolean);
2
3 @Name('Q3') select count(*) from
4   ProductionErrorStream.win:time(10 sec)
5   having count(*) > 1
```

Listing 7.3: CEP query Q3 in EPL syntax [FHKM18]

Q3 checks whether Q1 or Q2 produce more than one event for a time window of 10 seconds. The time window is used to avoid detection of false-positive events due to outliers. Only if multiple events are produced during

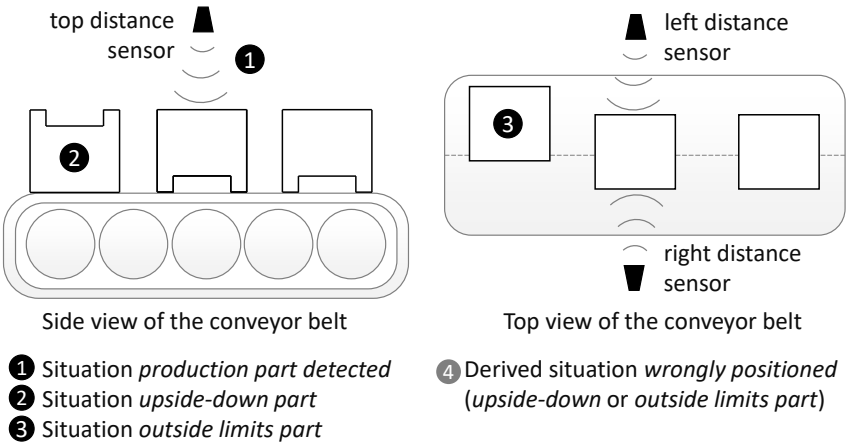


Figure 7.2: Case scenario: monitoring production parts on a conveyor belt [FHWM16]

this time window, it can be ensured that a production error really occurred.

Moreover, there is a notification service acting as data sink, which receives events of Q3 and notifies the person responsible to cope with the occurred error, for example, a maintenance engineer, who can replace the affected production part, or a shop floor worker that can remove the erroneous part from the production process. Finally, the software for executing the CEP queries are deployed as described in Chapter 6, which automatically starts the disturbance recognition.

7.2 Executing disturbance recognition

This section explains how to execute disturbance recognition based on the concepts explained in Section 7.1. For this, situations previously modeled as *situations templates* are first transformed onto executable, event-based representations, i.e., CEP queries. These representations are then executed by CEP engines, which process input monitoring data and generate notifications when disturbances are recognized based on the executable representations.

In Figure 7.2, an example abstracted from real-world production scenarios is depicted, which timely recognizes disturbances in production processes. In this example, production parts on a conveyor belt are monitored in order to recognize when they are wrongly positioned.

The disturbance recognition is based on the data generated by several distance sensors, which are attached to the conveyor belt. In this scenario, four situations are defined.

The situation *production part detected* ❶ indicates the presence of a production part on the conveyor belt. This situation alone does not necessarily indicate a disturbance, but its occurrence is required for the following defined situations. The situation *upside-down part* ❷ indicates that a production part is upside-down and, thus, a disturbance in the production process occurred. The situation *outside limits part* ❸ indicates that the production part is positioned outside the allowed limits either left or right and, therefore, corresponds to a disturbance in the production process.

The derived situation *wrongly positioned* ❹ is a composition of the aforementioned situations and indicates that a production part is either upside-down or positioned outside the allowed limits and, therefore, wrongly positioned on the conveyor belt.

In Figure 7.3, a situation template combining the aforementioned situations is depicted. This situation template is automatically transformed to an executable representation, which is depicted at the bottom. In this example, the executable representation is defined using the CEP query language *EPL* that can be executed by the *Esper* CEP engine. However, a wide range of CEP-based execution formats exist that could be used by this approach.

For the automated transformation of a situation template onto a CEP query, the nodes of the situation template are traversed in order to formulate a complex event pattern, which is composed of pattern expressions combined through logical operators (e.g., or, and). In this approach, the complex event pattern is built based on the set of condition nodes, which are aggregated by operation nodes. Each condition node corresponds to a pattern expression, while an operation node corresponds to a logical operator. The example in Listing 7.4 depicts such a complex event pattern in a pseudo

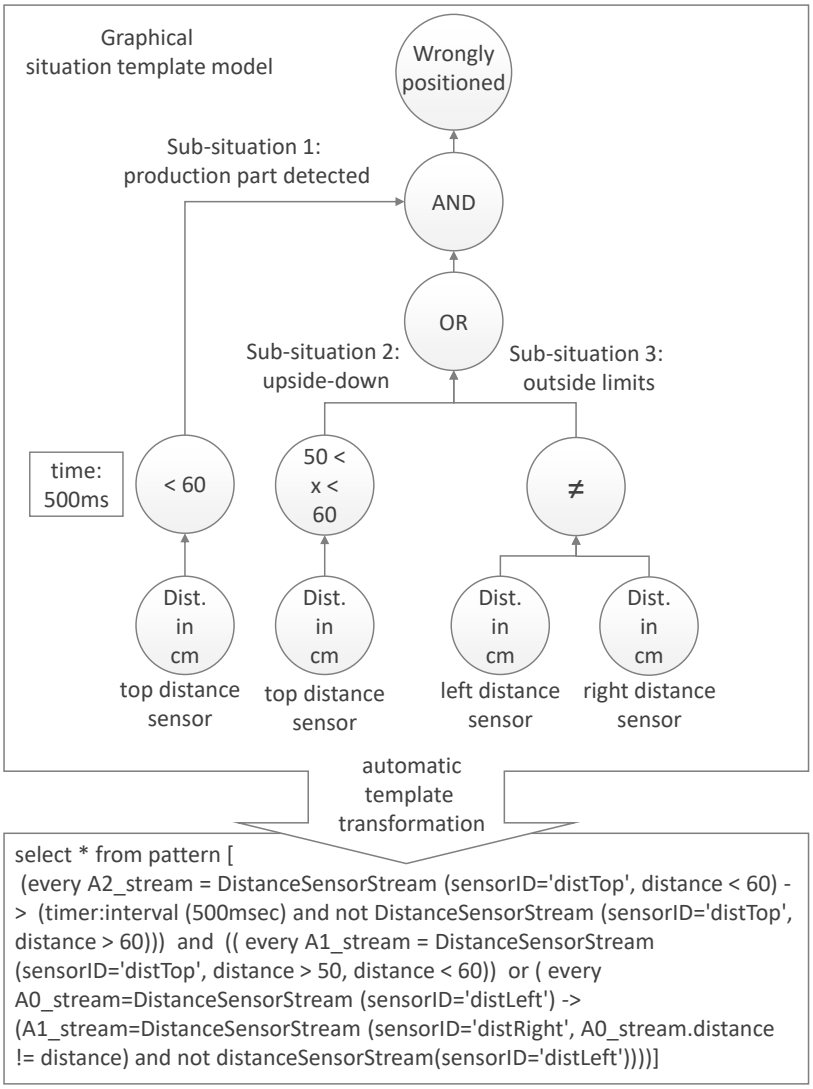


Figure 7.3: Exemplary situation template and its transformation to Esper CEP queries (based on [FHWM16])

query language, which is roughly based on the Esper CEP query language.

```
select * from pattern [
  <conditionNode_patternExpression>
  <operationNode_type>
  <conditionNode_patternExpression>
]
```

Listing 7.4: Complex event pattern in pseudo-code

To build the pattern expression for a condition node, the monitored sensor identifier and the sensor type are retrieved from the IoTEM. To map which physical sensor is used for a specific condition in the situation template, the *sensor role*, e.g., "top distance sensor", is specified in the context node and the transformation retrieves the registered sensor by this role. Listing 7.5 shows the structure of a pattern expression based on a condition node.

```
conditionNode_patternExpression = <sensor_type_stream>(
  sensor_id = '<monitored_sensor_id>',
  sensor_role = '<contextNode_sensor_role>',
  <conditionNode_condition>)
```

Listing 7.5: Complex event pattern expression in pseudo-code

In Figure 7.3 at the bottom, the CEP query to recognize the situation *wrongly positioned* is shown. This CEP query is complex and, therefore, difficult to be created manually. Furthermore, CEP queries may become verbose depending on the complexity of the situation to be recognized. By providing automatic transformation from situation templates onto CEP queries, domain experts are released from the burden of creating such complex CEP queries themselves.

Finally, the disturbance recognition is started by deploying the resulting CEP queries on the CEP execution engine. Within the scope of this thesis, a master thesis was conducted by Mahmoodi [Mah18], in which several CEP engines (e. g., Esper, Apache Flink, and Odysseus) were investigated. This master thesis provides an abstract query language, which expresses common

CEP features among the different investigated CEP engines. Consequently, besides using situation templates for situation recognition, this thesis also enables the modeling using this abstract query language. Similar to situation templates, queries modeled using the designed abstract query language are also transformed onto CEP queries that can be executed by the specific CEP engines. This enables the support of different execution environments, avoiding dependency on a specific CEP execution engine (vendor lock-in).

7.2.1 Customization and provisioning of CEP engines

In [FBH+17], we present an approach based on the TOSCA standard, in order to enable the customization and provisioning of CEP engines.

Many software solutions providing CEP functionalities have been developed, such as Esper [Esp06a], flowthings.io [flo10], FIWARE CEP GE [FIW16], or Odysseus [Uni07]. Furthermore, there are many approaches to provision CEP engines in a generic manner, such as Docker, Amazon AWS, Ansible, or Vagrant. However, generic approaches enabling provisioning of standard CEP engines lack the customization required by the IoT domain. More precisely, IoT environments are highly heterogeneous in regard to deployed IoT objects, applications, and continuous queries to process data.

Consequently, using standard cloud services for such CEP engines requires a high customization effort. These customization steps comprise: (i) configuration of the CEP engine, (ii) binding of data sources and sinks by writing and deploying complex operator code, and (iii) deploying CEP queries. When conducted manually, these steps are tedious and error-prone.

To cope with these issues, we present the approach in [FBH+17] for customization and provisioning of CEP engines, including their IoT environments, required operators, and CEP queries. This approach leads to a large reduction of customization effort when applying CEP to IoT environments. Furthermore, such a customized approach enables increased data security through the creation of fixed, non-changeable instances of CEP engines, similar to views in databases.

This approach builds on a self-contained TOSCA topology model, i. e.,

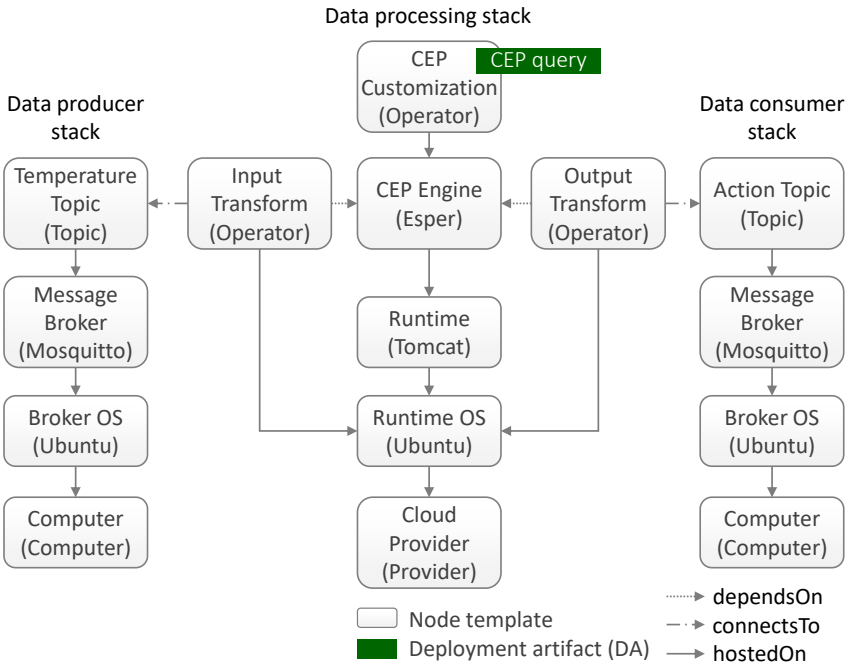


Figure 7.4: TOSCA topology model for a CEP engine (based on [FBH+17])

the topology model must contain all necessary software components to set up the CEP engine. Such a topology model is depicted in Figure 7.4. In this example, a simplified HVAC system scenario is modeled, in which the temperature of a room is continuously monitored to recognize and react when the temperature exceeds or goes below the thermal comfort zone.

The customization and provisioning of CEP engines require at least the following components: (i) a *CEP engine* to process and aggregate input data, (ii) *CEP queries* in a compatible query language, which define how the CEP engine processes input data to compute higher-level situations based on low-level data, (iii) *data producers*, which provide low-level data to the CEP engine, and (iv) *data consumers*, which receive and process the higher-level situations derived by the CEP engine. Therefore, the topology is divided in

three stacks: the data producer stack, the data processing stack containing the CEP engine and CEP queries, and the data consumer stack.

The *data producer stack* contains an IoT device, a temperature sensor, and the infrastructure providing access to sensor values through a message broker. For instance, temperature values are measured by a sensor plugged into a Raspberry Pi, which executes an operator to extract these values and to send them to a specific topic hosted on a message broker. In Figure 7.4, the node templates for the IoT object and extraction operator are omitted for simplification reasons. An example of a data producer stack including these node templates is shown in Figure 5.2.

The *data processing stack* depicted in the middle of Figure 7.4 contains the infrastructure and the CEP engine that monitors and processes sensor values. This stack can be created in two ways: (i) modeling the CEP engine self-hosted on own infrastructure and platform components or (ii) as a service, e. g., offered by an external cloud provider, such as Amazon AWS or Microsoft Azure. In the first case, the topology stack usually consists of seven node templates, as depicted in Figure 7.4. Starting from the bottom, a *Provider* node template corresponds to the hardware resource provider, which creates and runs virtual machines. Furthermore, the *Runtime OS* node template models the operating system of the virtual machine, while the *Runtime* node template models the necessary runtime, e. g., a web server for the CEP engine to be deployed in. In this thesis, the self-hosted approach is assumed in order to provide an approach that is generic and not dependent on a specific cloud provider.

In the following, the node templates of the customized CEP engine are described in detail. The *CEP engine* node template represents the CEP system itself, e.g., Esper. This node template, which is *hosted on* the Runtime node template, must contain: (i) an implementation artifact (IA) for installing, configuring and starting the CEP engine, and (ii) a property providing the API endpoint of the CEP engine to be used for customization, i.e., to push events, deploy queries, and define event types. Usually, (iii) a deployment artifact (DA) needs to be provided that contains the binaries of the engine's installers. If this DA is not provided, these binaries need to be dynamically

retrieved by the IA.

Once the CEP engine is set up, the customization of the CEP engine is realized by the *CEP Customization* node template, which defines the event types, i. e., input and output data events. More precisely, data is provided or consumed based on the modeled topic node template, e. g., the node templates Temperature Topic and Action Topic in the topology model depicted in Figure 7.4. Furthermore, the CEP Customization node template contains tailor-made CEP queries, which define how the CEP engine must process incoming data of the defined input events. Both event type definitions and CEP queries are represented as deployment artifacts, i.e., monolithic objects, such as text files, which are deployed into the CEP engine. Furthermore, the CEP Customization node template contains properties that define additional system-specific customizations, such as the (de-)activation of query optimizations.

To bind the *data producer stack* to the CEP engine through the publish-subscribe pattern, an operator is required that (i) subscribes to data producer topics, (ii) transforms the received data to a format the CEP engine understands, and (iii) pushes this transformed data to the CEP engine through its provided interface and API endpoint. For that, the *Input Transformation* node template is provided, which contains such an operator as a deployment artifact. This node template *connects to* the *Temperature Topic* node template and *depends on* the CEP engine node template, since it requires the API endpoint of the CEP engine being provided to push events to it. In addition, the node template contains an implementation artifact that deploys these deployment artifacts onto a corresponding runtime. In the depicted example, this is the runtime the CEP engine is hosted on. However, if the Input Transformation node template requires a different type of runtime, an additional node template can be modeled.

Similar to the binding of the data producer stack, a further operator is required to realize the binding of the data consumer stack to the CEP engine that (i) receives the resulting situation computed by the CEP engine based on the CEP queries, (ii) transforms it to the data format the message broker understands, and (iii) publishes it to the data consumer topic. Thus, the

Output Transformation node template is provided, which contains such an operator as a deployment artifact. This node template may also *connect to* several Topic node templates (e. g., Action Topic) and *depends on* the CEP engine node template, because it also requires the API endpoint of the CEP engine from which the operator receives derived situations. Similar to the Input Transformation node template, the Output Transformation also contains a corresponding implementation artifact that handles the deployment of this operator into a suitable runtime.

Modeling the CEP engine as a service offered by an external provider differs slightly from the model in Figure 7.4. The Runtime and Runtime OS node templates do not need to be modeled, however, the infrastructure for the Input Transformation and Output Transformation node templates still needs to be provided. As before, they can communicate with the CEP engine by using the API endpoint of the CEP Engine node template.

In summary, in order to create the data processing stack in the TOSCA topology model, containing the customized CEP engine, the above-mentioned components need to modeled and connected as described.

The *data consumer stack* works in a similar fashion as the data producer stack, since it also builds on the publish-subscribe pattern. Data consumers can be, for example, applications reacting on situations computed by the CEP engine. This stack contains an IoT device, an actuator, and furthermore, the infrastructure composed of a message broker, through which the actuator can be controlled. For instance, a ventilator actuator is plugged into a Raspberry Pi. On this IoT device, a control operator is hosted, which subscribes to the specific action topic of a message broker to extract on/off commands for the ventilator actuator. Furthermore, this operator sends the corresponding action commands to the ventilator actuator.

In Figure 7.4 on the right, the infrastructure for the message broker hosting the *Action Topic* node template is depicted. The node templates for the IoT device and control operator are omitted for simplification reasons. An example of a data consumer stack including these node templates is shown in Figure 5.2.

Furthermore, the introduced approach for customization and provisioning

of CEP engines enables another important feature: data security. Similar to the concept of *read-only views* [UGW02] in relational database management systems, this approach is able to limit the degree of exposure of underlying data through this customization approach. This can be realized by forbidding additional query and data input into the CEP engine as well as additional output through encapsulating the CEP engine (e.g., through port blocking) after its provisioning and only allowing communication through the deployed topics. In addition, by doing so, consumers of data processed by the CEP engine cannot retrace how the results have been achieved, i.e., the concrete structure of the CEP query is hidden.

7.2.2 Disturbance classes

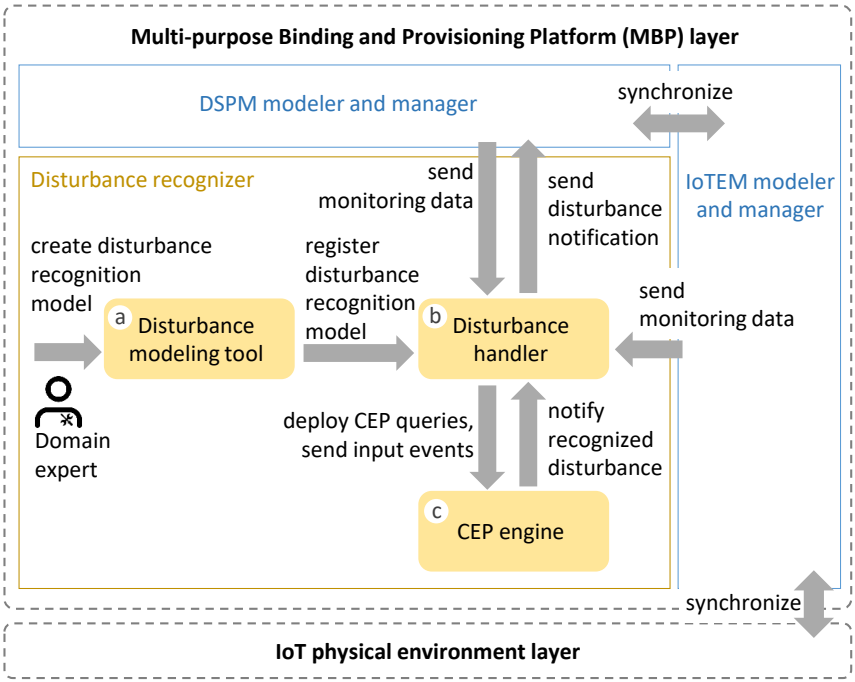
Due to the dynamic nature of IoT environments, reconfiguration of operators of a deployed DSPM might be required when disturbances in the IoT environment occur. Section 7.1 presents the means to model disturbance recognition. This section presents an overview of which disturbances are relevant to be modeled and recognized in the scope of this thesis.

This thesis takes into considerations two main classes for disturbances: (i) changes in the network infrastructure of an IoT environment, and (ii) changes in the DSPM executed on this IoT environment. The following list represents a subset of disturbances to be recognized, and hence, is not complete. These disturbances can occur through changes in the IoT environment or in the DSPM of an IoT application.

- **IoT object breakdown**. In case an IoT object stops working, this can cause a disturbance if the corresponding IoT object belongs to the deployed DSPM (dDSPM), i.e., if the IoT object is executing an operator or if it is a data source. For this, condition monitoring of IoT objects is conducted. Gupta [Gup15] gives an overview about monitoring mechanisms, such as Big Brother [Mac97] and Zenoss [Bad08], which can be applied for the condition monitoring of IoT objects connected to a network infrastructure. Such monitoring mechanisms generate notifications about critical values reaching threshold or condition failures

on IoT objects.

- **IoT object update**. Changes in the capabilities and properties of an IoT object might lead to a disturbance if the updated capabilities do not fulfill any requirement of the DSPM anymore. Furthermore, if an IoT object changes, for example, a crucial property, such as its IP address, the IoT object might become unreachable or the data flow might be compromised.
- **IoT object running low on resources**. If an IoT object becomes overloaded over time or if its battery state is low, this situation might lead to a disturbance if, as a consequence, this processing node stops working. In this case, the IoT object can be given more resources if possible, or the operator needs to be remapped to another IoT object.
- **IoT object added to the IoTEM**. When a new IoT object is added to the IoTEM in which a DSPM was already deployed, consequently, the IoT environment provides additional resources that can be taken into consideration the next time a mapping of operators onto IoT objects is realized. Within the scope of this thesis, a bachelor thesis was conducted by Fouskas [Fou17b], which designed an automated discovery service for IoT devices using different communication technologies, such as Wi-Fi or Bluetooth.
- **Processing operator removal**. In case a processing operator, a data source, or a data sink is removed from a deployed DSPM, the allocated resources of the IoT environment need to be released and the data flow needs to be checked to ensure that it is not compromised.
- **Processing operator update**. Changes in the requirements of a processing operator can lead to a disturbance if the new or updated requirements cannot be met anymore by the mapped IoT object.
- **Processing operator added to the DSPM**. When a new processing operator, data source, or data sink is added to a deployed DSPM, the new element is then required to be mapped onto a suitable IoT object.



IoTEM: IoT environment model
DSPM: Data stream processing model
dDSPM: deployed data stream processing model

Figure 7.5: Architecture component: Disturbance recognizer

Once such a change is recognized, in case this is a disturbance, possible correcting actions can be executed by domain experts or domain analysts. An example of a correcting action is the redeployment of necessary processing operators (i.e., operator migration). The resolution and execution of correcting actions for disturbances recognition is part of the future work of this thesis. For this, a complete remapping of processing operators and IoT objects will need to be conducted. The previous mapping plan of a dDSPM can then be compared to the new mapping plan and the difference can be extracted. Based on this difference, processing operators can be

redeployed, and the dDSPM can be updated accordingly. The redeployment of stateless operators is a simpler task, since such operators are only required to be redeployed. By redeploying stateful operators, however, the state of operators needs to be migrated. A survey about state management in data stream processing is presented by To et al. [TSM18]. Moreover, Cardellini et al. [CLNR18] investigate operator placement decisions for the effective runtime management of data stream processing applications in geographically distributed environments. Their work aims to select the optimal adaptation strategy that minimizes migration costs but still satisfies application QoS requirements.

7.3 Architecture component and implementation – Disturbance recognizer

In the overall architecture (cf. Section 3.3), the *Disturbance recognizer* component provides the means to monitor IoT objects and deployed DSPMs in order to recognize disturbances during the data processing. This component is depicted in Figure 7.5. The modeling of disturbances to be recognized is conducted by domain experts in the *Disturbance modeling tool* (a), which registers the modeled disturbances to the *Disturbance handler* (b).

The Disturbance handler transforms disturbance recognition models onto CEP queries (cf. Figure 7.3), which are deployed onto the *CEP engine* (c). Furthermore, the disturbance handler receives monitoring data about the IoT environment and processing operators. These monitoring data are transformed onto CEP input events and forwarded to the CEP engine. Once the CEP engine recognizes a disturbance, it notifies the Disturbance handler, which sends notifications about the disturbances to the *DSPM modeler and manager* component (cf. Section 4.2.3) and to the MBP dashboard.

The *Disturbance recognizer* component has been prototypically implemented as part of the MBP. Section 8.2 gives an overview on the MBP functionalities. The *Disturbance modeling tool*[1] has been implemented in

[1] https://github.com/IPVS-AS/MBP/wiki/Rules

JavaScript and uses the library jQuery. Created disturbance recognition models are stored in a MongoDB database. The *Disturbance handler* transforms disturbance recognition models into CEP queries, which can be processed by the Esper *CEP engine*[1]. The Disturbance handler is implemented in Java in the server side of the MBP. Section 8.1 presents the integration architecture of all architecture components and shows how they fit together.

7.4 Related work

This section describes related work regarding situation recognition within IoT environments. Many approaches exist that employ ontologies for situation recognition [WZGP04]. However, these approaches are either focused on specific use case scenarios [BMK+00] or cannot provide the efficiency suitable for real-time critical scenarios [DMM+13; WZGP04], e.g., in smart factories. These limitations regarding efficiency also occur in machine learning approaches [ASRH13]. In contrast, the approach in this thesis offers high efficiency by recognizing situations in milliseconds instead of seconds or even minutes as reported in [DMM+13; WZGP04]. This enables applicability in time-critical real-world scenarios, such as smart factories [LCW08], in which fast recognition times are of vital importance.

Several situation recognition systems using complex event processing were proposed in [GHM+13; HCBO11; TL11]. Taylor and Leidinger [TL11] propose the use of ontologies to specify and recognize complex events, whose occurrence can be detected in digital messages streamed from multiple sensor networks. The developed ontology is accessed through a user interface, where the user specifies the events of interest. The specification of an event of interest is then processed in order to generate configuration commands for a CEP system. The CEP system monitors the specified data streams and generates notifications, which can be delivered to clients when the event occurs [TL11]. In this article, complex events are not directly specified, but rather abstracted as situations. The approach in this thesis also realizes

[1] http://www.espertech.com/esper/esper-downloads

transformations of the event specifications (i.e., the modeled situations) into CEP queries for a CEP system. The difference is that this thesis does not use ontologies to model situations of interest.

Hasan et al. [HCBO11] propose to use CEP along with a dynamic enrichment of sensor data in order to realize situation-awareness. In this approach, the situations of interest are directly defined in the CEP engine, i.e., the user formulates the situations of interest using CEP query languages. A dynamic enrichment component processes and enriches the sensor data before the CEP engine evaluates them. This approach and the one in this thesis differ as follows: No complex dynamic enrichment of the sensor data is done in this thesis. The necessary information about the sensor for the situation recognition (e.g., the sensor identification) is kept at a minimum. This information together with the sensor reading are made available directly to the CEP engine, dispensing any further processing step of the sensor data. Furthermore, instead of defining situations directly as CEP queries, which can be long and complicated depending on the situation, this thesis defines the situations of interest as situation templates. The abstraction provided through situation templates enables the employment of CEP engines as well as the use of other technologies for situation recognition. Besides that, the use of situation templates also facilitates the modeling step of situations for the user, so that the user does not have to deal with the complexity of formulating CEP queries. The formulation of CEP queries is taken care of by transformations, which automatically create the necessary CEP queries for a given situation template.

Glombiewski et al. [GHM+13] present a similar approach integrating context from a wide range of sources for situation recognition using event processing technologies as well. However, they do not provide any abstraction, i.e., the users of the situation recognition have to create CEP queries themselves. This proves difficult, especially for domain experts, e.g., in factories, who do not have extensive computer science knowledge. In this thesis, an abstraction by situation templates [HHL+10] and a graphical interface are provided, which enable the usage by domain experts without necessary knowledge of technical details, such as event processing queries.

Chapter 8

Evaluation

This chapter describes and evaluates the Multi-purpose Binding and Provisioning Platform (MBP), which has been developed as a proof-of-concept of this thesis' contributions (cf. Chapters 4 to 7). The MBP is an open-source IoT platform and its prototypical implementation can be found as open-source projects on GitHub[1,2,3].

Section 8.1 presents the integration architecture for the architecture components of each contribution and describes its prototypical implementation. Section 8.2 gives an overview of the main functionalities of the MBP IoT platform and explains how it can be employed to realize a simple IoT scenario for the smart office domain. The concepts of the MBP IoT platform have been published as a demonstration paper in [FHS+20][4], therefore, are approved for feasibility. Finally, Section 8.3 evaluates the MBP architecture against the IEEE standard 1934-2018 for fog computing, which comprehends the OpenFog reference architecture [Ope17].

[1] https://github.com/IPVS-AS/MBP
[2] https://github.com/IPVS-AS/MBP-Docker
[3] https://github.com/IPVS-AS/MBP2Go
[4] Best Paper Award, Percom Demos 2020

8.1 Integration architecture and prototype

Figure 8.1 depicts the detailed resulting architecture of this thesis, which integrates the components of each contribution, and shows how they fit together. The contributions are highlighted by color and the steps of the methodical approach are also indicated. The architecture components and corresponding implementations for each contribution are explained in detail in Chapters 4 to 7, respectively.

In the overall architecture (cf. Section 3.3), contribution C1 encompasses two architecture components, the *IoTEM modeler and manager* (cf. Section 4.1.3), and the *DSPM modeler and manager* (cf. Section 4.2.3). These components are further divided in smaller components and are indicated in the color blue in Figure 8.1. The *IoTEM modeler and manager* component is the entry point of the methodical approach (step ❶) and supports domain experts during the creation and management of the IoTEM. The *DSPM modeler and manager* component supports domain analysts by providing the means to create and manage the DSPM (step ❷), and later to retire the deployed DSPM onto the IoT environment (step ❻).

Contribution C2 encompasses the architecture component *IoTEM and DSPM mapper*, which supports domain analysts on the decision about where processing operators should be deployed. This architecture component is further divided in smaller components and is indicated in the color orange in Figure 8.1. This component communicates with the architecture components of contribution C1 to retrieve the IoTEM and the DSPM. It provides the means to realize the IoTEM and DSPM mapping (step ❸) automatically through algorithms or manually by domain analysts.

Contribution C3 encompasses the architecture component *Deployment manager*, which provides the means for the deployment of operators onto IoT objects (step ❹). This component is indicated in the color green in Figure 8.1. This component receives the mapping plan from the architecture component of contribution C2 and starts the deployment based on this mapping plan. The current deployment status of an operator can be visualized in the *Dashboard*. In case manual actions are required by the deployment (e. g.,

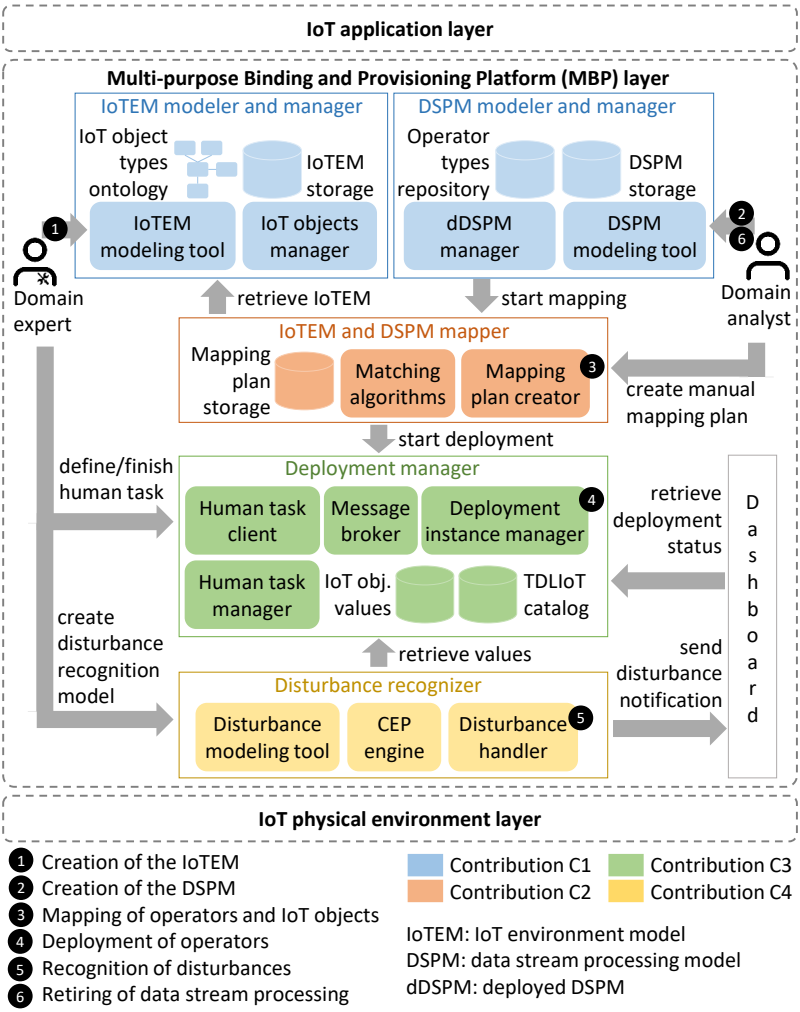


Figure 8.1: Integration architecture of this thesis

plugging sensors), this component provides the means for domain experts to define human tasks (cf. Section 6.2), to get notified about human tasks to be executed, and to finish human tasks after they are manually executed.

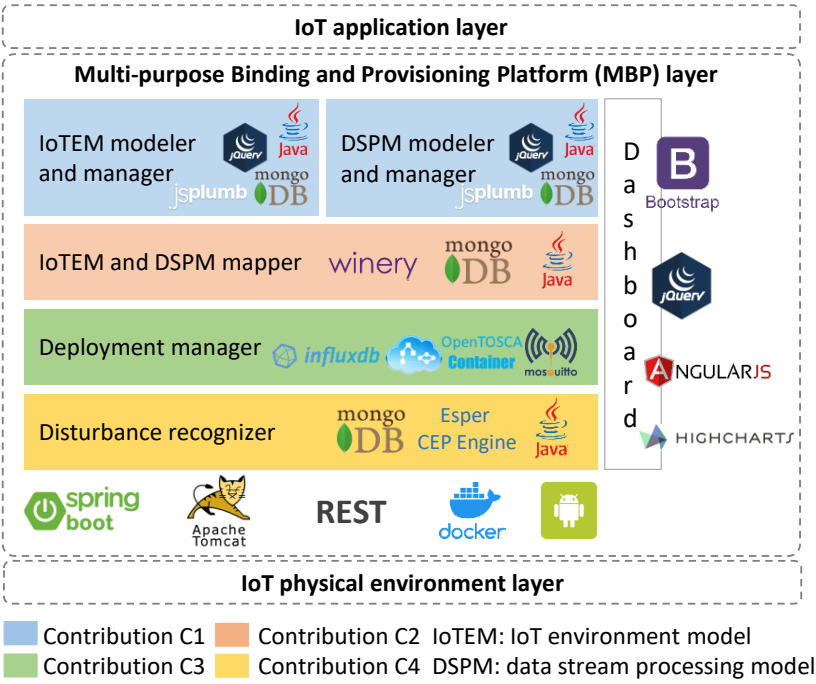


Figure 8.2: Overall detailed architecture of this thesis

Contribution C4 encompasses the architecture component *Disturbance recognizer*, which is indicated in the color yellow in Figure 8.1. This component provides the means to monitor IoT objects and deployed DSPMs, in order to recognize disturbances during the data processing (step ❺). It provides a disturbance modeling tool, which domain experts can model and start the recognition of disturbances. This component continuously retrieves output values of deployed operators and monitoring values of IoT objects from the architecture component of contribution C3. These values are forwarded to the CEP engine, where they are processed based on disturbance recognition models. In case disturbances are recognized, notifications about the disturbances are sent to the *Dashboard*, where they can be depicted to either domain experts or domain analysts.

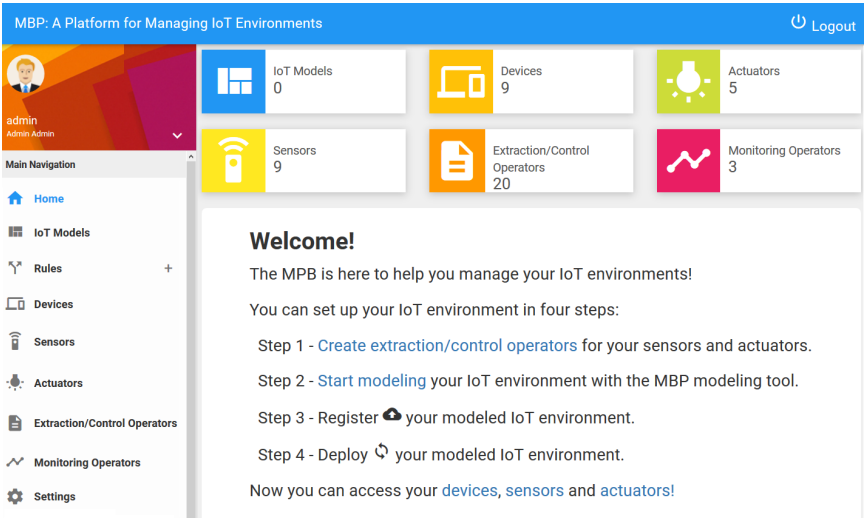


Figure 8.3: The MBP user interface [FHS+20]

In Figure 8.2, the software technologies employed in the prototypical implementation of the MBP architecture are depicted. The MBP implementation uses mainly JavaScript and Angular for the MBP user interface and Java for the MBP server. The REST API is built on the Spring boot framework, and the whole server component is provided as a web application running on the Java application server Apache Tomcat. The MBP can be installed on Windows, Mac or Linux operating systems by installing its required software components individually, or by an installation script on Linux. In addition, the MBP can be run as a Docker container. After installation, the MBP user interface, which is depicted in Figure 8.3, can be accessed through an Internet browser (e.g., Chrome or Firefox). The MBP user interface is based on Bootstrap templates and uses the JavaScript framework Angular JS. Icons are extracted from Google's Material Design library. Furthermore, the library Highcharts was employed in the Dashboard to visualize live and historical output values of deployed operators and to show monitoring data of IoT objects. A REST API reflecting the same functionalities provided by the MBP

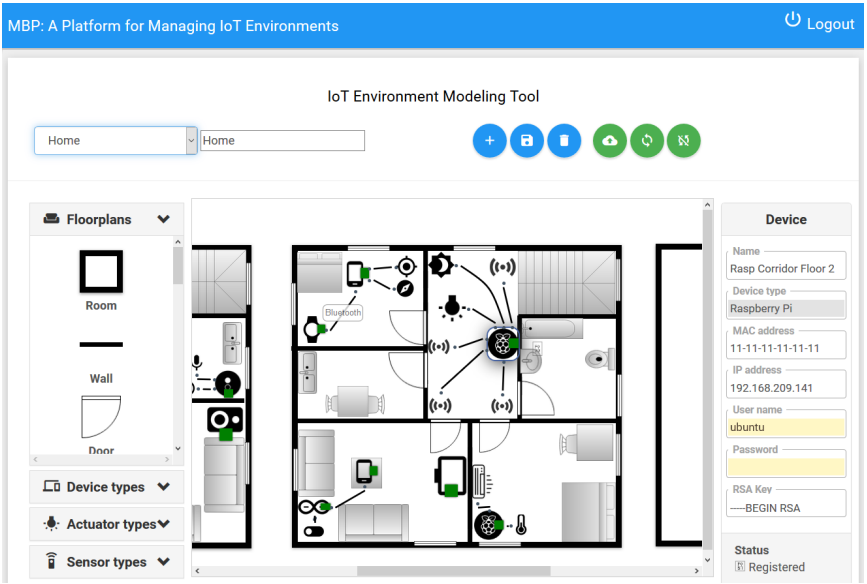


Figure 8.4: The MBP IoTEM modeling tool [FHS+20]

user interface has been implemented in the MBP server as well.

Regarding the *IoTEM modeler and manager* component, the *IoTEM modeling tool* has been implemented in JavaScript and uses the JavaScript library jQuery. The jsPlumb Toolkit has been used to implement a drag-and-drop editor for modeling IoT environments, which is depicted in Figure 8.4. The *IoT object manager* is implemented in the MBP server using Java. The MongoDB database is used to store *IoT object types* and modeled IoTEMs.

Regarding the *DSPM modeler and manager* component, the FlexMash tool [Hir18] is used as the *DSPM modeling tool*, which was developed by Hirmer et al. In the scope of this thesis, the FlexMash graphical interface and its JSON-based underlying model were extended within the master thesis conducted by Chaudhry [Cha18]. The extensions include enabling the annotation of requirements on processing operators and on edges between operators. Furthermore, FlexMash was extended to connect to the MBP

and enable the import of sources and sinks from IoTEMs. The FlexMash graphical interface was implemented in JavaScript and uses jQuery and jsPlumb. Modeled DSPMs are stored in the MBP in a MongoDB database. The *dDSPM manager* is implemented in the MBP server using Java. The operator types repository corresponds to a file system, which is configured in the installation location of the MBP. Several operators have been implemented in Python and Shell, which can be found in the MBP GitHub project[1].

Concerning the *IoTEM and DSPM mapper* component, Winery [KBBL13] is used as an external *mapping plan creator* for the manual mapping approach. In this case, the mapping plans are based on the TOSCA standard and are stored in a configured file system for Winery. For the automatic mapping approach, the mapping plan creator and two matching algorithms, a greedy variant and a backtracking variant, were implemented in the MBP server in Java within the scope of the bachelor thesis conducted by Schneider [Sch18]. In this case, automatically generated mapping plans are stored in a MongoDB database.

Concerning the *Deployment manager* component, the OpenTOSCA container [BBH+13] is used as an external *deployment instance manager*, which is able to automatically deploy operators of mapping plans based on the TOSCA standard. Within the scope of this thesis, a bachelor thesis was conducted by Kutger [Kut18], which implemented the *Human task manager* and the *human task client*, to enable the OpenTOSCA container to support human tasks. The *Human tasks manager* is based on the implementation from Wagner [Wag10], which corresponds to a Java web application running on Apache Tomcat. The *Human tasks client* was implemented as an Android-based mobile application. Additionally, the MBP provides several ready-to-use extraction and control operators as Python and Shell scripts[1], which can then be registered in the MBP and be deployed onto IoT objects by the MBP. Once operators are deployed, they send values to the MBP by connecting and publishing to the *Message broker*, which is implemented using the MQTT broker Mosquitto. The MBP can then visualize sensor values

[1] https://github.com/IPVS-AS/MBP/tree/master/resources/operators

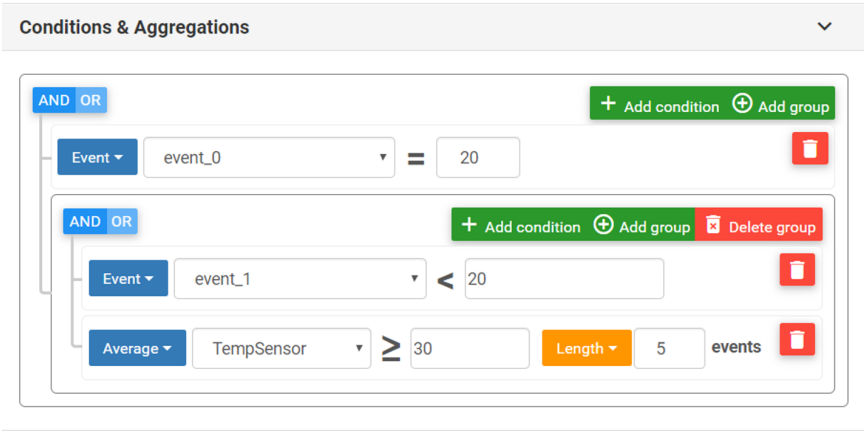


Figure 8.5: The MBP disturbance modeling tool

as line diagrams in the Dashboard. The data management and processing in the MBP implements a lambda-based architecture [MW15], in which both live and historical data can be managed and processed. To store measurement data, i.e., sensor data and timestamp-based data, the time-series database InfluxDB is used. Further metadata, IoT environment models, and data stream processing models are stored in the NoSQL databases MongoDB, to clearly separate measurement data and corresponding metadata. The *TDLIoT catalog* has been prototypically implemented outside the MBP as an open-source GitHub project[1]. It is implemented in Java and uses a MongoDB database to store TDLIoT descriptions. It provides a web interface implemented in JavaScript and a REST API for interactions with the TDLIoT catalog.

Concerning the *Disturbance recognizer* component, the *Disturbance modeling tool*, which is depicted in Figure 8.5, is implemented in JavaScript and uses the library jQuery. Created disturbance recognition models are stored in the MongoDB database. The *Disturbance handler* transforms disturbance recognition models into CEP queries, which can be processed by the Esper

[1] https://github.com/IPVS-AS/TDLIoT

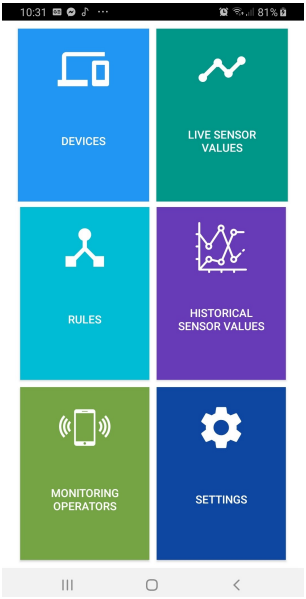


Figure 8.6: The MBP mobile client application [FHS+20]

CEP engine. The Disturbance handler is implemented in Java.

Within the scope of this thesis, the MBP2Go, an Android-based mobile client application implemented in Java to connect to the MBP, has been developed. The MBP2Go is depicted in Figure 8.6 and was partially implemented within the bachelor thesis conducted by Ulusal [Ulu19]. The MBP2Go provides an overview of registered IoT objects in the MBP. It can register and remove IoT objects, and visualize current and historical sensor values. The modeling of IoT objects with the MBP2Go can be automated by scanning *QR code templates* provided in the MBP2Go GitHub repository[1]. By scanning such QR code templates, the MBP2Go automatically fills properties that are possible to be inferred for the IoT object type being modeled.

[1] https://github.com/IPVS-AS/MBP2Go

8.2 MBP overview

The following section is mostly based on [FHS+20].

Since the popularization of the Internet of Things [VF13], many commercial and non-commercial IoT platforms were developed to help non-expert users with the management of IoT objects within their IoT environments. However, the binding and provisioning of IoT objects to these IoT platforms are still very challenging tasks for non-expert users.

In the context of this thesis, *binding* means enabling IoT applications to access IoT objects on a higher level of abstraction, so that the required hardware expertise is kept at a minimum.

In [HBF+16], we primarily designed the MBP to ease the management of IoT environments, however, we also moved a step deeper into supporting users already during the binding and provisioning of IoT environments. In many IoT platforms, such as FIWARE [RGSE14], IBM Watson IoT [Nel16], OpenMTC [CCE+12], or Microsoft Azure IoT [Kle17], IoT objects are registered, bound, and provisioned to IoT platforms in a manual fashion. Such tasks are complex and require technical knowledge about the IoT objects. That is, operators (i.e., software code) to extract and provision sensor data to IoT applications, as well as to receive actuator control commands from IoT applications are required. Such operators need to be created and deployed for each sensor and actuator manually. Furthermore, monitoring functionality needs to be implemented and deployed manually as well.

Deploying operators manually is error-prone and time-consuming, since a hardware expert has to configure IoT objects, install necessary operators for the specific sensors and actuators, bind them, and provide accessible interfaces to IoT applications. In real-world scenarios, e.g., for situation recognition [FHWM16], efficiency and accuracy requirements are crucial. However, these requirements cannot be met through manual binding and provisioning. To tackle the aforementioned issues, the MBP was developed to support users in the entire life cycle of IoT environments, so that the amount of manual tasks are kept to the possible minimum. This includes the automated deployment, management, and monitoring of IoT environments.

The main functionalities of the MBP are explained in the following.

8.2.1 Modeling IoT environments

IoT objects can be registered to the MBP either separately or as part of a specific IoT environment. The second option additionally enables the user to model the connections among IoT objects within the IoT environment. In the MBP IoTEM modeling tool, IoT objects including their properties and connections are specified. These properties describe specific information about the IoT objects, such as IoT object type, identifier, or MAC address.

To automate the modeling of IoT objects, the MBP provides *QR code templates* and an Android-based smartphone application to scan these QR codes and automatically fill properties that are possible to be inferred for the IoT object being modeled. Furthermore, following the same fashion, IoT objects can be automatically discovered by connecting to the *MBP registration Wi-Fi hotspot*. That is, the MBP listens for new connections to this configured hotspot and automatically creates models with inferred properties for the connected IoT objects.

Once an IoT environment is modeled and saved, it can automatically be registered through the MBP user interface. At this point, basic monitoring information about IoT objects can be already visualized in dashboards, such as network accessibility, CPU usage, and CPU temperature of a device.

8.2.2 Deploying operators onto IoT environments

The MBP provides several ready-to-use extraction and control operators in the form of scripts, which can be found in the MBP GitHub project. One exemplary extraction operator reads measurement values of an analog temperature sensor (TMP36 module) connected to a Raspberry Pi. This operator sends the extracted values to the MBP through MQTT, a publish-subscribe communication protocol. Such operators can be simply linked to registered devices and, in sequence, be deployed onto these devices. Through this approach, sensors are bound automatically to the MBP and measured

sensor values can be live-visualized and are also available as historical data.

The MBP also enables the users to provide their own operators, which can be implemented in any programming language. The MBP only requires the existence of specific life-cycle management scripts (e.g., install, start, stop, or terminate) for the operator, in order to be able to automate the deployment of the user-defined operators. The application logic of these management scripts can be still defined by the user, e.g., specifying necessary software that needs to be installed. In this way, users can create operators fulfilling their specific requirements, such as the use of specific hardware types or programming libraries.

8.2.3 Monitoring IoT environments

Once IoT objects are bound to the MBP, they can be monitored by the users through provided dashboards. These dashboards show status and statistical information about IoT devices, sensors, and actuators. Furthermore, both live sensor data and historical sensor data can be visualized.

To realize an automatic monitoring based on sensor data and to trigger actuators automatically based on recognized disturbances, the MBP provides the means to recognize disturbances based on user-defined rules. Such rules, which are based on the event-condition-action pattern [KRRS96], can be defined in the MBP disturbance modeling tool. These high-level rules, i. e., disturbance recognition models, are then internally transformed into complex event processing (CEP) queries and evaluated using corresponding CEP systems [Luc01]. By employing CEP systems, the MBP has the advantage of achieving an efficient and timely sensor data processing.

8.2.4 Demonstration: smart office

This section explains how a simple IoT scenario, a smart office, can be realized with the MBP. An office is defined as a room in which people conduct their work tasks. In order to make an office *smart*, it can be enhanced with computing power and sensing and acting capabilities through IoT devices,

Figure 8.7: Lego smart offices [FHS+20]

sensors, and actuators. A common practice in real-world scenarios is to integrate a heating, ventilation, air conditioning (HVAC) system into rooms. The goal of such a system is to keep the indoor environment comfortable for its occupants, while taking over tasks that can be executed automatically. For example, the room's temperature can be automatically regulated for its occupants based on user-defined goals and continuous sensor measurements.

In the Lego IoT environment depicted in Figure 8.7, several Lego miniature offices were enhanced with diverse IoT objects, in which many IoT applications, including a HVAC system, can be realized. To implement the HVAC system, two IoT devices can be used: (i) a *Bosch XDK* device equipped with eight sensors, including humidity and temperature sensors, and (ii) a *Raspberry Pi*, which is connected to a relay actuator controlling a cooler fan.

To realize the IoT scenario, the IoT devices, the temperature and humidity sensors, and the relay actuator are modeled in the IoTEM modeling tool

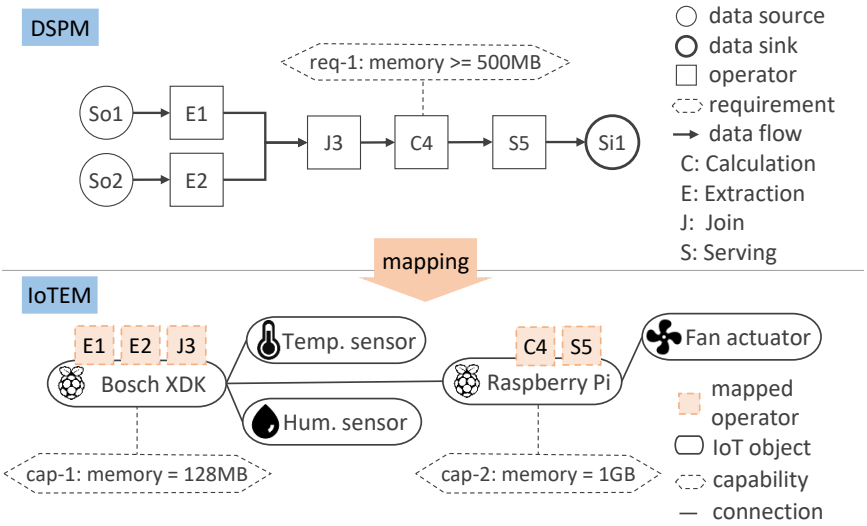


Figure 8.8: Mapping in smart office scenario

(step ❶ in Figure 8.1). The modeled IoTEM is depicted in Figure 8.8. After this, the DSPM modeling tool (FlexMash tool) imports the modeled IoTEM and abstracts the temperature sensor, the humidity sensor, and the relay actuator as two data sources and one data sink. A DSPM containing the processing logic for the HVAC system is then modeled for this IoTEM (step ❷ in Figure 8.1). The DSPM consist of two extraction operators, a join operator, a calculation operator and a serving operator. The modeled DSPM is depicted in Figure 8.8. The concrete implementation of these operators are Python and Shell scripts, which are stored in the MBP. Extraction operators for the specific temperature and humidity sensors of the XDK, and for the relay actuator are provided in the MBP GitHub repository. The join operator combines a temperature value and a humidity value into one measurement based on a time-window. The calculation operator checks if the temperature value is under or above the thermal comfort zone inside the office, e.g., from 20 to 22 Celsius degrees. This operator also calculates, based on temperature

and humidity values, if the current mold level has reached an unhealthy zone. If the temperature is above the thermal comfort zone or the mold level is unhealthy, this operator generates a command indicating that the cooler fan should be turned on. The serving operator receives turn on and off commands from the calculation operator and switches the relay actuator controlling the cooler fan accordingly. For simple scenarios, event-condition-action rules as provided by the Disturbance Recognition could be used. For more complex scenarios, such as the Smart Office scenario, modeling a DSPM is required, which provides a more sophisticated means to model data processing among several devices, sensors and actuators.

Through the FlexMash tool, the automated mapping algorithm is started and a mapping plan is created (step ❸ in Figure 8.1). The greedy variant of the matching algorithm is employed to decide where to deploy the operators. A possible mapping is depicted in Figure 8.8. The extraction operators E1 and E2 are mapped to the IoT device, to which the sensors are physically attached to. Similarly, the serving operator S5 is mapped to the IoT device, to which the relay actuator is physically connected to. Afterwards, the join operator J3 is mapped to the device nearest to the deployed extraction operators. The calculation operator C4 is mapped to the device matching the operator's requirement, i.e., at least 500 MB memory. Based on the mapping plan, the deployment of the operators is started (step ❹ in Figure 8.1). When the deployment is completed, basic monitoring information about the devices can be visualized in dashboards, such as network accessibility and CPU temperature of the devices. Sensor values can be also visualized in the detailed sensor view of the MBP.

To assure that the processing of the deployed data stream processing model (dDSPM) stays correct as long as needed by the HVAC system, several high-level rules, i. e., disturbance recognition models, are created for the set up IoT environment, in order to recognized disturbances (step ❺ in Figure 8.1). To recognize if the devices are overheating, a disturbance recognition model is created in the MBP disturbance modeling tool that continuously checks if the CPU temperature of the devices is near to the operating temperature limit (85°C for the Raspberry Pi). In case a distur-

bance is recognized, notifications about the disturbance are shown in the MBP dashboard. Once the HVAC system is not needed anymore, the dDSPM of the HVAC system is stopped and retired through the dDSPM manager (step ❻ in Figure 8.1).

8.3 Further considerations

In 2018, the *OpenFog reference architecture* [Ope17] was adopted as an official standard for fog computing. This standard, known as IEEE 1934-2018, presents eight main principles that IoT platforms need to implement in order to satisfy the data-intensive requirements existing in IoT environments. These principles are security, scalability, openness, autonomy, RAS (reliability, availability, and serviceability), agility, hierarchy and programmability. In this section, the MBP architecture is evaluated against these principles.

The **security** principle aims to achieve safe, trusted transactions within IoT environments. For this, an approach is required to discover, attest, and verify IoT objects before trust can be established. In the MBP, IoT objects are verified through authentication mechanisms, such as passwords, digital credentials, or certificates. Furthermore, the data exchange between IoT objects and the MBP is supported by well-established security mechanisms, such as HTTPS, or OAuth2. Moreover, the MBP provides user management and ownership concepts, so that IoT objects are accessible only to their owners or to other users with granted access permissions.

The **scalability** principle aims to benefit from scaling opportunities provided within IoT environments. These opportunities exist in many dimensions, such as scalable performance, scalable capacity, and scalable software. *Scalable performance* refers to the demanded performance of IoT applications, which should be provided with small latency between sensor measurements and resulting actuator commands, even when the number of IoT objects connected to the IoT platform grows. For this, the MBP employs CEP technologies, which are able to continuously process large amounts of data, and furthermore, to process sensor data efficiently and timely, so that actuator

commands can be issued as soon as possible. *Scalable capacity* refers to the capability of IoT objects to be added (or removed) to IoT environments. The MBP provides mechanisms to update IoT environments, by adding, updating, or removing IoT objects, which include both hardware objects (e.g., devices, sensors, actuators) and virtual objects (e.g., virtual machines). Finally, *scalable software* refers, among other aspects, to the scalability of the management infrastructure of an IoT platform, i.e., its capability to manage lots of IoT objects as demanded by IoT scenarios. Due to the modular architecture of the MBP and to the employment of scalable software components (e.g., scale databases, or message brokers) as described in Section 8.1, the MBP is also able to provide this dimension of scalability.

The **openness** principle aims to avoid vendor lock-in. The MBP is available as a GitHub open-source project [FH+17] and, therefore, is neither proprietary nor a single-vendor solution. Moreover, the MBP is based on established standards, such as MQTT, TOSCA, XML, in order to ensure its long lasting applicability. Furthermore, the MBP does not have any constraint about which IoT objects can be connected to the MBP. Through its dynamic binding concepts [HBF+16], operators for any kind of IoT object can be created and registered in the MBP. The MBP provides several ready-to-use extraction and control operators in the form of scripts, but also enables users to provide their own operators in any desired programming language.

The **autonomy** principle refers to the decision making autonomy of IoT objects once failures occur. The MBP provides the means to deploy operators on IoT objects, so that a basic degree of autonomy of IoT objects is enabled through operators. However, upon recognized failures in IoT environments connected to the MBP, the required decision-making to react to failures is conducted mostly by the MBP itself. Nonetheless, more sophisticated approaches are still required to achieve the goals of this principle and, therefore, will be part of future work based upon the concepts of this thesis.

The **programmability** principle aims to achieve a highly adaptive deployment so that giving new tasks (i.e., operators) to an IoT object occurs in an automated fashion. This principle is achieved in the MBP through its operator placement and dynamic deployment concepts [FHM19], e.g., in

which an operator can be redeployed to another IoT object automatically.

The **agility** principle aims at the transformation of huge amounts of low-level data into compact, high-level information to help users to analyze this data and make business decisions. Furthermore, it refers to the capability of dealing with dynamic IoT environments by responding to changes quickly. The MBP provides several approaches to help users to early recognize disturbances in the IoT environments, and furthermore, to conduct correcting actions as early as possible. Examples for such approaches are high-level information presentation through dashboards, a disturbance modeling tool for user-defined rules, which enables automatic rule-based reaction on disturbance recognition, and timely data processing using CEP technologies.

To realize an automatic monitoring based on sensor data and to trigger actuators automatically based on recognized disturbances, the MBP provides the means to recognize disturbances based on user-defined rules. Such rules, which are based on the event-condition-action pattern [KRRS96], can be defined in the MBP disturbance modeling tool.

The **hierarchy** principle aims at hierarchical architectures composed of several layers, where each layer addresses specific problems in IoT scenarios. As depicted in Figure 8.2, the MBP provides a layered architecture, where from the bottom layer to the top layer, each layer corresponds to a different level of abstraction. For example, the *Disturbance recognizer* processes low-level sensor data, while the *DSPM modeler and manager* deals with high-level application logic.

Finally, the **RAS (reliability, availability, serviceability)** principle is divided into three aspects. The *reliability* is defined as the ability of the IoT platform to provide the designed functionalities even upon adverse circumstances, for example, when an IoT object becomes faulty. The *availability* refers to the continuous management and orchestration of the IoT environment. The *serviceability* refers to the correct operation of IoT applications on the IoT environment, for example, through automated deployment and repair. Throughout the contributions of this thesis (cf. Chapters 4 to 7), it has been shown how the MBP addresses the concerns about reliability, availability and serviceability. To increase reliability, an approach to recog-

nize disturbances as early as possible is presented in Chapter 7. Moreover, to increase availability, the MBP manages (cf. Chapter 4) and monitors (cf. Chapter 7) IoT environments and deployed operators continuously. Finally, Chapters 5 and 6 support serviceability through several concepts to automatically deploy and redeploy operators onto IoT environments.

Chapter 9

Conclusion and future work

In the last decades, the Internet of Things (IoT) vision has become more and more a reality. Advances in, for example, hardware and network technologies have enabled the existence of IoT environments containing devices, sensors, and actuators, which can be employed to realize diverse sophisticated applications, such as smart cities, smart homes, or smart factories.

Within IoT environments, large amounts of data streams are continuously generated, which leads to great challenges in respect to their processing and management. In many approaches, IoT data is transferred to cloud infrastructures, where the processing is executed centralized.

However, depending on the application at hand, this can increase latency and network traffic, and furthermore, provoke delays before and after the processing. Other approaches, in contrast, execute the data processing within the IoT environment, in order to avoid latency and network traffic, and furthermore, to achieve a timely processing of data streams. For this, operator placement is essential. Over the last decades, many approaches

have been proposed that tackle the operator placement problem. Many of them decide the placement location based on the fulfillment of QoS requirements, such as latency and bandwidth.

However, in the IoT domain, further aspects have emerged that additionally need to be taken into consideration while realizing operator placement within IoT environments, such as heterogeneity of processing nodes. Current approaches lack in supporting such additional requirements introduced by the IoT domain while realizing operator placement decisions for IoT environments. They also normally do not take into consideration non-functional and user-defined requirements for IoT applications, such as data privacy and anonymization, security, and interoperability.

Therefore, in this PhD thesis, an approach for the placement of data stream processing operators onto IoT environments was presented, which takes into consideration the characteristics of the IoT domain as well as non-functional and user-defined requirements during the operator placement decision. For this, an IoT environment and its processing capabilities are described by an IoT environment model (IoTEM). Likewise, the business logic of an IoT application and its requirements are defined by a data stream processing model (DSPM). These informational models are then employed to enable the operator placement decision for IoT environments, i.e., to decide where processing operators should be placed onto IoT environments based on the matching of requirements and capabilities. Through the approach of this PhD thesis, data processing of IoT applications can be tailored to particular use cases, supporting the specific requirements of the IoT domain, and furthermore, of IoT application users.

9.1 Summary

This PhD thesis provided four contributions (C), which address the research questions, and furthermore, provide concepts to achieve the goals described in Section 1.2:

C1: Modeling of IoT environment and data stream processing;

C2: Mapping of DSPMs onto IoTEMs;

C3: Deployment of operators onto IoT environments;

C4: Monitoring of the deployed DSPM.

In contribution C1 (cf. Chapter 4), the achievement of goals G2 and G3 was reached by providing the IoTEM for the modeling of IoT environments and the DSPM for the modeling of the business logic of IoT applications. This contribution enables a user-friendly, easy modeling of IoT environments through a graphical modeling tool in which it is possible to model heterogeneous IoT objects, their interconnections, and furthermore, the capabilities of IoT objects and connections. On the other hand, the modeling of data stream processing supporting IoT requirements was achieved through another graphical modeling tool, in which the processing logic of IoT applications for domain-specific use cases, including IoT and user requirements, are modeled.

Furthermore, contribution C2 (cf. Chapter 5) achieves goal G4 by enabling a requirements-based placement of processing operators onto IoT environments. For this, this contribution provides several algorithms to conduct mappings of data stream processing models (DSPMs) onto IoT environment models (IoTEMs), considering the requirements of the processing operators to be fulfilled by the capabilities of the IoT objects.

In contribution C3 (cf. Chapter 6), goal G5 was achieved by providing several deployment approaches based on the TOSCA standard. These approaches are able to deal with the heterogeneous and dynamic nature of IoT environments, achieving in this way, an efficient deployment of operators onto IoT objects.

Moreover, contribution C4 (cf. Chapter 7) achieves goal G6 by continuously monitoring IoT environments and deployed operators by employing well-established CEP techniques, so that disturbances are recognized as early as possible. In summary, all goals are covered by the contributions.

As a proof-of-concept of this thesis' contributions, the IoT platform *Multi-purpose Binding and Provisioning Platform (MBP)* has been developed as a

prototype. The MBP has been developed as an open-source project throughout several student works supervised within the scope of this PhD thesis. Furthermore, the concepts of the MBP as an IoT platform have been published as a demonstration paper in [FHS+20] and, therefore, approved for feasibility. In Chapter 8, an overview on the main functionalities of the MBP was provided and its architecture was evaluated against the OpenFog reference architecture provided by the IEEE standard 1934-2018 for fog computing. In Figure 8.1, the complete resulting architecture including the individual architecture components of each contribution, are depicted. The contributions are highlighted by color and the steps of the methodical approach employing this architecture are also indicated. The architecture components of each individual contribution are explained in detail in Chapters 4 to 7.

The methodical approach is composed of six main steps: ❶ creation of the IoT environment model (IoTEM), ❷ creation of the data stream processing model (DSPM), ❸ mapping of processing operators and IoT objects, ❹ deployment of processing operators onto IoT objects, ❺ recognition of disturbances affecting the data processing, and ❻ retirement of the data processing. In the methodical approach, two main roles are defined, the *domain expert*, which conducts step ❶ and part of step ❺, and the *domain analyst*, which conducts step ❷ and part of step ❻.

Domain experts have technical knowledge about the hardware objects (i.e., devices, sensors, actuators), the virtual objects (i.e., virtual machines), and their network interconnections within an IoT environment. Furthermore, domain experts have the knowledge how to access these IoT objects to, for example, extract sensor data or send control commands to an actuator. Therefore, in step ❶, the main task of domain experts is the creation of IoTEMs, which are directed graphs containing IoT objects as nodes, and their network interconnections as edges.

Domain analysts have domain knowledge about the processing of data generated within the IoT environment, i.e., they have the required knowledge to model different IoT applications for domain-specific use cases. In step ❷, the main task of domain analysts is the creation of DSPMs representing the processing logic of IoT applications. Furthermore, they can retire the data

stream processing of an IoT application in step ❻ .

The contributions altogether enable the achievement of goal G1, i.e., they enable a timely, efficient processing of IoT data within IoT environments. These contributions were based on established standards, e.g., MQTT, TOSCA, XML, in order to ensure their long lasting applicability, and furthermore, they have been published in several journals, national and international conferences. The contributions were further evaluated by integrating them into different research projects, whereby their applicability could be confirmed. These projects include the research projects SitOPT [WSBL15], SmartOrchestra [ABF+19; LAB+18], and IC4F [IC417]. The SitOPT project aimed at the development of concepts and methods to allow situation-based applications, so that they could adapt autonomously to the dynamic environment in which they run. The SmartOrchestra project aimed at the development of an open platform for the safe combination and TOSCA-based orchestration of smart services for cyber-physical applications, and furthermore, for the effective marketing of such smart services. The IC4F project aimed at the development of secure, robust and real-time communication solutions for the manufacturing industry.

Finally, software developed as part of this thesis have been made available as open-source projects in GitHub [1,2,3,4].

9.2 Future work

Currently, when disturbances in the IoT environments are recognized that require a correcting action, new operator placement decisions need to be recalculated, which is realized by the MBP in a central manner. In future work, this approach can be extended to distribute the operator placement decision logic throughout different instances, including IoT objects, in order to avoid a single point of failure (SPOF), and furthermore, to increase

[1] https://github.com/IPVS-AS/MBP
[2] https://github.com/IPVS-AS/MBP-Docker
[3] https://github.com/IPVS-AS/MBP2Go
[4] https://github.com/IPVS-AS/TDLIoT

the autonomy of IoT objects once failures occur. Therefore, IoT objects and deployed operators should be extended with an autonomy concept, as recommended by the OpenFog reference architecture for fog computing (IEEE standard 1934-2018). In this way, IoT objects and deployed operators will be able to autonomously recognize disturbances and partially calculate a new operator placement for themselves.

Furthermore, the concepts of this thesis can be extended to enable predictive analytics for IoT data, in order to, for example, predict when IoT devices, sensors, or actuators might become faulty, require maintenance, or the hardware need to be newly calibrated. By enabling such predictions, the reliability of IoT environments can be increased, and consequently, IoT applications being executed on such IoT environments can highly improve their robustness and quality of processing results. Preliminary work has been already conducted in the scope of this thesis by a student survey of Bacharew et al. [BVV20].

Finally, an important and up-to-date aspect of IoT platforms and IoT objects is how to ensure that they are provided with the minimal necessary security and data privacy. Due to the highly interconnected nature of the IoT, IoT objects are prone to attacks that are not only isolated but can be rather propagated through whole IoT environments. Therefore, the concepts of this thesis should be extended with further privacy and security measures. Preliminary work has been already conducted in the scope of this thesis by a student survey of Glaub et al. [GSU19].

Author publications

The following lists the author publications, which are divided as first author and co-author publications.

First author publications

- A. C. Franco da Silva, P. Hirmer, M. Wieland, B. Mitschang: SitRS XT – Towards Near Real Time Situation Recognition. Journal of Information and Data Management (JIDM), 2016
- A. C. Franco da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann: OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker. In: Proceedings of the 6th International Conference on the Internet of Things, 2016
- A. C. Franco da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke: Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments. In: Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER), 2017
- A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang: Customization and provisioning of complex event processing using

TOSCA. In: Research and Development, Springer, 2017

- A. C. Franco da Silva, P. Hirmer, R. Koch Peres, B. Mitschang: An Approach for CEP Query Shipping to Support Distributed IoT Environments. In: Proceedings of the 14th Workshop on Context and Activity Modeling and Recognition at Percom, 2018
- A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang: TDLIoT: A Topic Description Language for the Internet of Things, In: Proceedings of the 18th International Conference on Web Engineering (ICWE), 2018
- A. C. Franco da Silva, P. Hirmer, B. Mitschang: Model-based Operator Placement for Data Processing in IoT Environments, In: Proceedings of the IEEE International Conference on Smart Computing (SMARTCOMP), 2019
- A. C. Franco da Silva, P. Hirmer, J. Schneider, S. Ulusal, M. Tavares Frigo: MBP: Not just an IoT Platform, In: Proceedings of the 18th IEEE International Conference on Pervasive Computing and Communications (PerCom), 2020
- A. C. Franco da Silva and P. Hirmer: Models for Internet of Things Environments – a Survey, In: Information, Vol. 11, 2020

Co-author publications

- P. Hirmer, U. Breitenbücher, A. C. Franco da Silva, K. Képes, B. Mitschang, M. Wieland: Automating the Provisioning and Configuration of Devices in the Internet of Things, In: Complex Systems Informatics and Modeling Quarterly (CSIMQ), Vol. 9, 28–43, 2016
- C. Stach, F. Steimle, A. C. Franco da Silva: TIROL: The Extensible Interconnectivity Layer for mHealth Applications. In: Proceedings of the 23rd International Conference on Information and Software Technologies (ICIST), 2017

- A. Liebing, L. Ashauer, U. Breitenbücher, T. Günther, M. Hahn, K. Képes, O. Kopp, F. Leymann, B. Mitschang, A. C. Franco da Silva, R. Steinke: The SmartOrchestra Platform: A Configurable Smart Service Platform for IoT Systems. In: Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSOC'18), 2018
- L. Ashauer, U. Breitenbücher, A. C. Franco da Silva, O. G. Gemein, T. Günther, M. Hahn, K. Képes, E. Kleinod, O. Kopp, F. Leymann, A. Liebing, B. Mitschang, P. Niehues, D. Olschewski, K. Semmler, R. Steinke, J. van Well, M. Virtel: Sichere internetbasierte Vermarktung cyber-physischer Systeme mit SmartOrchestra. In: Sichere Plattformarchitekturen - Rechtliche Herausforderungen und technische Lösungsansätze. Begleitforschung Smart Service Welt - Internetbasierte Dienste für die Wirtschaft, 2019
- M. Tavares Frigo, P. Hirmer, A. C. Franco da Silva, L. H. Thom: A Toolbox for the Internet of Things–Easing the Setup of IoT Applications. In: Proceedings of the ER Forum, Demo and Posters 2020 co-located with 39th International Conference on Conceptual Modeling, 2020

Supervised student work

The following lists the student work supervised by me, which collaborated to the implementation of the concepts presented in this PhD thesis.

- A. Hüneburg: Automatische, TOSCA-basierte Provisionierung des Situationserkennungssystems SitOPT, Bachelor thesis, 2016
- A. Blehm, O. Kabierschke, S. Lehmann: Analyse und Vergleich von IoT-Plattformen, Prozessanalyse, 2016
- A. Fouskas: Automatisches Auffinden und Anbinden von IoT-Geräten, Bachelor thesis, 2017
- D. Krüger: Ein Testwerkzeug für das Internet der Dinge, Bachelor thesis, 2017
- D. Krüger, Q. T. Pham, F. Pfeffer: Modellierungstool für IoT Umgebungen, Projekt-INF, 2017
- M. B. Chaudhry: Enhancing data flow models with computing requirements for distributed IoT environments, Master thesis, 2018
- Isabella Kutger: Human Tasks für OpenTOSCA zum Aufsetzen von IoT-Anwendungen, Bachelor thesis, 2018
- S. Mahmoodi: A Canonical Language for Complex Event Processing Systems, Master thesis, 2018

- J. Schneider: Finden einer geeigneten Infrastruktur für Datenoperationen in IoT-Umgebungen, Bachelor thesis, 2018
- S. Lehmann: Policy4TDLIoT - Policies for the Topic Description Language, Master thesis, 2018
- E. Czychon, F. Scheerer, S. Ulusal: Ultimativer Vergleich mobiler IoT-Applikationen, Fachstudie, 2018
- F. Bauer, S. Öney, N. Dörr: Ultimativer Vergleich von Betriebssystemen und Laufzeitumgebungen für das IoT, Fachstudie, 2018
- S. Ulusal: MBP2Go+: Erweiterung der mobilen MBP-Applikation um Monitoring-Funktionalitäten, Bachelor thesis, 2019
- A. Imeri: Modellierung und Deployment von IoT-Umgebungen in der MBP, Bachelor thesis, 2019
- S. Glaub, J. Schneider, S. Ulusal: Eine Sicherheitsanalyse der IoT-Plattform MBP, Master-Fachstudie, 2019
- A. Bacharew, C. Vieira Rocha, M. Vijayaruban: Algorithmen für Maschinelles Lernen auf IoT-Daten, Master-Fachstudie, 2020

Bibliography

[AAB+05] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. “The Design of the Borealis Stream Processing Engine.” In: *Cidr*. Vol. 5. 2005. 2005, pp. 277–289 (cit. on pp. 32, 66).

[AAS13] C. C. Aggarwal, N. Ashish, A. Sheth. “Managing and Mining Sensor Data.” In: ed. by C. C. Aggarwal. Boston, MA: Springer US, 2013. Chap. The Internet of Things: A Survey from the Data-Centric Perspective, pp. 383–428 (cit. on p. 49).

[ABF+19] L. Ashauer, U. Breitenbücher, A. C. Franco da Silva, O. G. Gemein, T. Günther, M. Hahn, K. Képes, E. Kleinod, O. Kopp, F. Leymann, A. Liebing, B. Mitschang, Niehues, D. Olschewski, K. Semmler, R. Steinke, J. van Well, M. Viertel. “Sichere internetbasierte Vermarktung cyber-physischer Systeme mit SmartOrchestra.” In: *Sichere Plattformarchitekturen - Rechtliche Herausforderungen und technische Lösungsansätze*. Begleitforschung Smart Service Welt - Internetbasierte Dienste für die Wirtschaft, 2019 (cit. on p. 157).

[ACÇ+03] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik. “Aurora: a new model and architecture for data stream management.” In: *The VLDB Journal* 12.2 (2003), pp. 120–139 (cit. on pp. 63, 65, 66).

[ACD+03] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, et al. *Business process execution language for web services*. Online. 2003 (cit. on p. 35).

[AGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash. "Internet of things: A Survey on Enabling Technologies, Protocols, and Applications." In: *IEEE communications surveys & tutorials* 17.4 (2015), pp. 2347–2376 (cit. on p. 57).

[AHS06] K. Aberer, M. Hauswirth, A. Salehi. "A middleware for fast and flexible sensor network deployment." In: *Proceedings of the International Conference on Very Large Data Bases (VLDB 2006)*. Seoul, Korea: ACM, 2006 (cit. on p. 60).

[AHS07] K. Aberer, M. Hauswirth, A. Salehi. "Invited Talk: Zero-Programming Sensor Network Deployment." In: *2007 International Symposium on Applications and the Internet Workshops*. IEEE, 2007 (cit. on p. 60).

[AIM10] L. Atzori, A. Iera, G. Morabito. "The internet of things: A survey." In: *Computer networks* 54.15 (2010), pp. 2787–2805 (cit. on p. 30).

[AMMD15] M. B. Alaya, S. Medjiah, T. Monteil, K. Drira. "Toward Semantic Interoperability in oneM2M Architecture." In: *IEEE Communications Magazine* 53.12 (2015), pp. 35–41 (cit. on p. 50).

[App14] Apple Inc. *Apple HomeKit*. Online. 2014. URL: `https://www.apple.com/ios/home` (cit. on p. 30).

[Ard05] Arduino Company. *Arduino*. Online. 2005. URL: `https://www.arduino.cc` (cit. on p. 56).

[Ard13] Arduino Company. *Arduino Yún*. online. 2013. URL: `https://www.arduino.cc/en/Guide/ArduinoYun` (cit. on p. 56).

[ARJ19] Asghari, Parvaneh, Rahmani, Amir Masoud, Javadi, Hamid Haj Seyyed. "Internet of Things applications: A systematic review." In: *Computer Networks* 148 (2019), pp. 241–261 (cit. on pp. 18, 30).

[Ash+09] K. Ashton et al. "That 'internet of things' thing." In: *RFID journal* 22.7 (2009), pp. 97–114 (cit. on p. 29).

[ASRH13] J. Attard, S. Scerri, I. Rivera, S. Handschuh. "Ontology-based Situation Recognition for Context-aware Systems." In: *Proceedings of the 9th International Conference on Semantic Systems*. Graz, Austria: ACM, 2013 (cit. on p. 130).

[Bad08] M. Badger. *Zenoss core network and system monitoring*. Packt Publishing Ltd, 2008 (cit. on p. 126).

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. “Models and Issues in Data Stream Systems.” In: *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, 2002 (cit. on pp. 18, 30, 62, 65).

[BBD+13] M. Bauer, N. Bui, J. De Loof, C. Magerkurth, A. Nettsträter, J. Stefa, J. W. Walewski. “IoT Reference Model.” In: *Enabling Things to Talk: Designing IoT solutions with the IoT Architectural Reference Model*. Springer Berlin Heidelberg, 2013. Chap. 7, pp. 113–162 (cit. on p. 50).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications.” In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (cit. on pp. 35, 88, 94, 95, 106, 139).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. “Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA.” In: *On the Move to Meaningful Internet Systems: OTM 2012: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part I*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 416–424 (cit. on pp. 34, 108).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. “Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA.” English. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*. IEEE Computer Society, Mar. 2014, pp. 87–96 (cit. on p. 35).

[BBK+16] U. Breitenbücher, T. Binz, O. Kopp, K. Képes, F. Leymann, J. Wettinger. “Hybrid TOSCA Provisioning Plans: Integrating Declarative and Imperative Cloud Application Provisioning Technologies.” In: *Communications in Computer and Information Science*. Springer Nature, 2016, pp. 239–262 (cit. on p. 35).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “TOSCA: Portable Automated Deployment and Management of Cloud Applications.” In: *Advanced Web Services*. Ed. by A. Bouguettaya, Z. Q. Sheng, F. Daniel. New York, NY: Springer New York, 2014, pp. 527–549 (cit. on p. 33).

[BD15] R. Bruns, J. Dunkel. *Complex Event Processing: Komplexe Analyse von massiven Datenströmen mit CEP*. Springer-Verlag, 2015 (cit. on pp. 26, 40, 111, 112).

[BDPP16] A. Botta, W. de Donato, V. Persico, A. Pescapé. "Integration of Cloud computing and Internet of Things: A survey." In: *Future Generation Computer Systems* 56 (2016), pp. 684–700 (cit. on p. 33).

[BEBT16] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi, K. Taylor. "IoT-Lite: A Lightweight Semantic Model for the Internet of Things." In: *Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld), 2016 Intl IEEE Conferences*. IEEE. 2016, pp. 90–97 (cit. on pp. 41, 52).

[BEBT17] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi, K. Taylor. "IoT-Lite: a lightweight semantic model for the internet of things and its use with dynamic semantics." In: *Personal and Ubiquitous Computing* 21.3 (June 2017), pp. 475–487 (cit. on pp. 50, 52).

[BG14] A. Banks, R. Gupta. *MQTT Version 3.1.1*. OASIS, 2014 (cit. on p. 99).

[BHM18] V. K. C. Bumgardner, C. Hickey, V. W. Marek. "An Edge-Focused Model for Distributed Streaming Data Applications." In: *Proceedings of the 2018 IEEE International Conference on Pervasive Computing and Communications Workshops*. 2018 (cit. on p. 91).

[BK09] A. Buchmann, B. Koldehofe. "Complex Event Processing." In: *IT-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik* 51.5 (2009), pp. 241–242 (cit. on pp. 26, 30, 31, 40, 111, 112).

[BL12a] M. Blackstock, R. Lea. "IoT mashups with the WoTKit." In: *2012 3rd IEEE International Conference on the Internet of Things*. IEEE. 2012, pp. 159–166 (cit. on p. 69).

[BL12b] M. Blackstock, R. Lea. "WoTKit: A Lightweight Toolkit for the Web of Things." In: *Proceedings of the Third International Workshop on the Web of Things*. ACM. 2012, p. 3 (cit. on p. 69).

[BL14] M. Blackstock, R. Lea. “Toward a distributed data flow platform for the web of things (distributed node-red).” In: *Proceedings of the 5th International Workshop on Web of Things*. ACM. 2014, pp. 34–39 (cit. on p. 69).

[BMK+00] B. Brumitt, B. Meyers, J. Krumm, A. Kern, S. Shafer. “EasyLiving: Technologies for Intelligent Environments.” English. In: *Handheld and Ubiquitous Computing*. Springer Berlin Heidelberg, 2000 (cit. on p. 130).

[BMP13] P. Balamuralidhara, P. Misra, A. Pal. “Software platforms for internet of things and M2M.” In: *Journal of the Indian Institute of Science* 93.3 (2013), pp. 487–498 (cit. on pp. 60, 70).

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-oriented Software Architecture – A System of Patterns*. Vol. 1. Wiley, 1996 (cit. on pp. 62, 63).

[BVV20] A. Bacharew, C. Vieira Rocha, M. Vijayaruban. *Algorithmen für Maschinelles Lernen auf IoT-Daten*. Universität Stuttgart. Master-Fachstudie. 2020 (cit. on p. 158).

[Can+13] A. Cantino et al. *Huginn – Your agents are standing by!* Online. 2013. URL: `https://github.com/huginn/huginn` (cit. on p. 69).

[CCE+12] M. Corici, H. Coskun, A. Elmangoush, A. Kurniawan, T. Mao, T. Magedanz, S. Wahle. “OpenMTC: Prototyping Machine Type communication in carrier grade operator networks.” In: *Globecom Workshops (GC Wkshps), 2012 IEEE*. IEEE. 2012, pp. 1735–1740 (cit. on pp. 49, 60, 142).

[Cha18] M. B. Chaudhry. *Enhacing data flow models with computing requirements for distributed IoT environments*. Universität Stuttgart. Master thesis. 2018 (cit. on pp. 68, 138).

[Che09] Chef. *Chef Infra: Infrastructure Automation for Hardened, Consistent Configuration at Any Scale*. Online. 2009. URL: `https://www.chef.io/products/chef-infra/` (cit. on p. 71).

[Che76] P. P.-S. Chen. “The Entity-Relationship Model – Toward a Unified View of Data.” In: *ACM Transactions on Database Systems (TODS)* 1.1 (1976), pp. 9–36 (cit. on p. 102).

[Cip14] N. Cipriani. “Flexible processing of streamed context data in a distributed environment.” PhD thesis. Universität Stuttgart, 2014 (cit. on p. 90).

[CKE+15] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. “Apache flink: Stream and batch processing in a single engine.” In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36.4 (2015) (cit. on pp. 31, 32).

[CLM10] N. Cipriani, C. Lübbe, B. Mitschang. “Exploiting constraints to build a flexible and extensible data stream processing middleware.” In: *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*. Apr. 2010, pp. 1–8 (cit. on pp. 54, 90).

[CLNR18] V. Cardellini, F. Lo Presti, M. Nardelli, G. Russo Russo. “Optimal operator deployment and replication for elastic distributed data stream processing.” In: *Concurrency and Computation: Practice and Experience* 30.9 (2018), e4334 (cit. on p. 129).

[CLRS09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to Algorithms*. MIT press, 2009 (cit. on p. 74).

[CM12] G. Cugola, A. Margara. “Processing Flows of Information: From Data Stream to Complex Event Processing.” In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15 (cit. on pp. 18, 19, 30–32, 62, 65).

[CM13] G. Cugola, A. Margara. “Deployment strategies for distributed complex event processing.” In: *Computing* 95.2 (2013), pp. 129–156 (cit. on pp. 32, 66, 115).

[CSM11] N. Cipriani, O. Schiller, B. Mitschang. “M-TOP: Multi-target Operator Placement of Query Graphs for Data Streams.” In: *Proceedings of the 15th Symposium on International Database Engineering & Applications*. IDEAS ’11. ACM, 2011, pp. 52–60 (cit. on p. 90).

[CT12] M. Chinosi, A. Trombetta. “BPMN: An introduction to the standard.” In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134 (cit. on p. 35).

[Dea15] T. Dean. *Network+ Guide to Networks*. Course Technology Press, 2015 (cit. on pp. 54, 56).

[DEDP15] H. Derhamy, J. Eliasson, J. Delsing, P. Priller. “A survey of commercial frameworks for the internet of things.” In: *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. IEEE. 2015, pp. 1–8 (cit. on p. 30).

[Dey01] A. K. Dey. “Understanding and Using Context.” In: *Personal and Ubiquitous Computing* (2001) (cit. on p. 112).

[DMM+13] W. Dargie, J. Mendez, C. Mobius, K. Rybina, V. Thost, A.-Y. Turhan, et al. “Situation Recognition for Service Management Systems Using OWL 2 Reasoners.” In: *Proceedings of the 10th IEEE Workshop on Context Modeling and Reasoning 2013*. IEEE Computer Society, 2013, pp. 31–36 (cit. on p. 130).

[EBB+11] M. Eckert, F. Bry, S. Brodt, O. Poppe, S. Hausmann. “A CEP Babelfish: Languages for Complex Event Processing and Querying Surveyed.” In: *Reasoning in Event-Based Distributed Systems*. Springer, 2011, pp. 47–70 (cit. on p. 66).

[Ecl17] Eclipse. *Vorto*. online. 2017. URL: `https://github.com/eclipse/vorto` (cit. on p. 50).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. “The Many Faces of Publish/Subscribe.” In: *ACM Computing Surveys (CSUR)* 35.2 (June 2003), pp. 114–131 (cit. on pp. 34, 100).

[EN10] O. Etzion, P. Niblett. *Event Processing in Action*. Manning Publications Co., 2010 (cit. on p. 31).

[Esp06a] EsperTech Inc. *Esper*. Online. 2006. URL: `http://www.espertech.com/esper` (cit. on pp. 31, 32, 121).

[Esp06b] EsperTech Inc. *Esper Reference*. 2006. URL: `http://www.espertech.com/esper/release-5.3.0/esper-reference/html` (cit. on p. 115).

[Esp14] Espressif. *ESP8266*. online. 2014. URL: `https://www.esp8266.com` (cit. on p. 56).

[FBH+17] A. C. Franco da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. “Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments.” In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress. SciTePress Digital Library, 2017, pp. 358–367 (cit. on pp. 61, 70, 84, 85, 95, 97, 121, 122).

[FBK+16] A. C. Franco da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. “OpenTOSCA for IoT: Automating the Deployment of IoT Applications Based on the Mosquitto Message Broker.” In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT’16. Stuttgart, Germany: ACM, 2016, pp. 181–182 (cit. on pp. 33, 61, 84, 95, 97).

[FH+17] A. C. Franco da Silva, P. Hirmer, et al. *Multi-purpose Binding and Provisioning Platform (MBP)*. Online. Institute for Parallel and Distributed Systems/Applications of Parallel and Distributed Systems (IPVS/AS), University of Stuttgart. 2017. URL: `https://github.com/IPVS-AS/MBP` (cit. on p. 149).

[FH20] A. C. Franco da Silva, P. Hirmer. “Models for Internet of Things Environments—A Survey.” In: *Information* 11.10 (2020). URL: `https://www.mdpi.com/2078-2489/11/10/487` (cit. on pp. 48–50).

[FHB+17] A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang. “Customization and provisioning of complex event processing using TOSCA.” In: *Computer Science - Research and Development* (2017), pp. 1–11 (cit. on pp. 61, 69, 84, 95, 97).

[FHB+18] A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang. “TDLIoT: A Topic Description Language for the Internet of Things.” In: *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer International Publishing, 2018, pp. 333–348 (cit. on pp. 50, 98–101, 103, 104).

[FHKM18] A. C. Franco da Silva, P. Hirmer, R. Koch Peres, B. Mitschang. “An Approach for CEP Query Shipping to Support Distributed IoT Environments.” In: *Proceedings of the 2018 IEEE International Conference*

on Pervasive Computing and Communication Workshops. 2018 (cit. on pp. 115, 116).

[FHM19] A. C. Franco da Silva, P. Hirmer, B. Mitschang. "Model-based Operator Placement for Data Processing in IoT Environments." In: *Proceedings of the IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2019 (cit. on pp. 52, 62, 75, 78, 79, 149).

[FHS+20] A. C. Franco da Silva, P. Hirmer, J. Schneider, S. Ulusal, M. Tavares Frigo. "MBP: Not just an IoT Platform." In: *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom)*. 2020 (cit. on pp. 25, 133, 137, 138, 141, 142, 145, 156).

[FHST20] M. Frigo, P. Hirmer, A. C. F. da Silva, L. H. Thom. "A Toolbox for the Internet of Things–Easing the Setup of IoT Applications." In: *Proceedings of the ER Forum, Demo and Posters 2020 co-located with 39th International Conference on Conceptual Modeling*. 2020 (cit. on p. 71).

[FHWM16] A. C. Franco da Silva, P. Hirmer, M. Wieland, B. Mitschang. "SitRS XT – Towards Near Real Time Situation Recognition." In: *Journal of Information and Data Management* 7.1 (Apr. 2016), pp. 4–17 (cit. on pp. 26, 40, 74, 111–113, 117, 119, 142).

[Fit09] Fitbit, Inc. *Fitbit Trackers*. online. 2009. URL: `https://www.fitbit.com` (cit. on p. 56).

[FIW16] FIWARE. *Complex Event Processing (CEP) - Proactive Technology Online*. Online. 2016. URL: `https://github.com/ishkin/Proton` (cit. on pp. 32, 121).

[flo10] flowthings.io. *flowthings.io*. Online. 2010. URL: `https://flowthings.io` (cit. on pp. 32, 121).

[Fou17a] O. Foundation. *OPC Unified Architecture Specification. Part 5: Information Model*. Release 1.04. OPC Foundation, 2017 (cit. on p. 50).

[Fou17b] A. Fouskas. *Automated discovery and binding of IoT devices*. Universität Stuttgart. Bachelor thesis. 2017 (cit. on p. 127).

[GAW+08] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, M. Doo. “SPADE: The System s Declarative Stream Processing Engine.” In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 1123–1134 (cit. on p. 32).

[GBH+05] M. Großmann, M. Bauer, N. Hönle, U.-P. Käppeler, D. Nicklas, T. Schwarz. “Efficiently Managing Context Information for Large-Scale Scenarios.” In: *Proc. of the Third IEEE Intl. Conf. on Pervasive Computing and Communications*. 2005 (cit. on p. 112).

[GBMP13] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. “Internet of Things (IoT): A vision, architectural elements, and future directions.” In: *Future Generation Computer Systems* 29.7 (2013), pp. 1645–1660 (cit. on pp. 18, 49).

[GHC+16] I. Grangel-González, L. Halilaj, G. Coskun, S. Auer, D. Collarana, M. Hoffmeister. “Towards a Semantic Administrative Shell for Industry 4.0 Components.” In: *Proceding of the 10th International Conference on Semantic Computing (ICSC)*. IEEE, 2016, pp. 230–237 (cit. on p. 103).

[GHM+13] N. Glombiewski, B. Hoßbach, A. Morgen, F. Ritter, B. Seeger. “Event Processing on your own Database.” In: *BTW workshops*. 2013, pp. 33–42 (cit. on pp. 130, 131).

[Goo17] Google. *Google Cloud IoT Solutions*. Online. 2017. URL: `https://cloud.google.com/solutions/iot` (cit. on p. 30).

[GSU19] S. Glaub, J. Schneider, S. Ulusal. *Eine Sicherheitsanalyse der IoT-Plattform MBP*. Universität Stuttgart. Master-Fachstudie. 2019 (cit. on p. 158).

[Gup15] U. Gupta. “Monitoring in IOT enabled devices.” In: *CoRR* abs/1507.03780 (2015) (cit. on p. 126).

[HA94] N. Haller, R. Atkinson. “On Internet Authentication.” In: (1994) (cit. on p. 55).

[HB17] P. Hirmer, M. Behringer. “FlexMash 2.0 – Flexible Modeling and Execution of Data Mashups.” In: *Rapid Mashup Development Tools* 696 (2017), pp. 10–29 (cit. on pp. 67, 69).

[HBF+16] P. Hirmer, U. Breitenbücher, A. C. Franco da Silva, K. Képes, B. Mitschang, M. Wieland. “Automating the Provisioning and Configuration of Devices in the Internet of Things.” In: *Complex Systems Informatics and Modeling Quarterly* 9 (2016), pp. 28–43 (cit. on pp. 56, 108, 142, 149).

[HCBO11] S. Hasan, E. Curry, M. Banduk, S. O’Riain. “Toward Situation Awareness for the Semantic Sensor Web: Complex Event Processing with Dynamic Linked Data Enrichment.” In: *SSN* 839 (2011), pp. 69–81 (cit. on pp. 130, 131).

[HCJL15] K. Hur, S. Chun, X. Jin, K.-H. Lee. “Towards a Semantic Model for Automated Deployment of IoT Services Across Platforms.” In: *Proceedings of the 2015 IEEE World Congress on Services*. SERVICES ’15. IEEE, 2015, pp. 17–20 (cit. on p. 108).

[HHL+10] K. Häussermann, C. Hubig, P. Levi, F. Leymann, O. Simoneit, M. Wieland, O. Zweigle. “Understanding and designing situation-aware mobile and ubiquitous computing systems.” In: *Proc. of intern. Conf. on Mobile, Ubiquitous and Pervasive Computing* (2010), pp. 329–339 (cit. on pp. 113, 131).

[HHM17] E. Hoos, P. Hirmer, B. Mitschang. “Towards Context-Aware Decision Information Packages to Improve Problem Resolving on the Shop Floor.” In: *Proceedings of the 29th International Conference on Advanced Information Systems Engineering (CAiSE)*. 2017 (cit. on p. 115).

[Hir18] P. Hirmer. “Anforderungsbasierte Modellierung und Ausführung von Datenflussmodellen.” PhD thesis. Universität Stuttgart, 2018 (cit. on pp. 20, 67–69, 138).

[HM16] P. Hirmer, B. Mitschang. “FlexMash - Flexible Data Mashups Based on Pattern-Based Model Transformation.” In: vol. 591. Rapid Mashup Development Tools. Springer International Publishing, 2016, pp. 12–30 (cit. on pp. 67, 69).

[Hor13] B. Horan. *Practical Raspberry Pi*. Apress, 2013 (cit. on p. 57).

[HW03] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc., 2003 (cit. on pp. 86, 99).

[HWBM16a] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Automated Sensor Registration, Binding and Sensor Data Provisioning.” In: *Proceedings of the CAiSE 2016 Forum at the 28th International Conference on Advanced Information Systems Engineering*. 2016 (cit. on pp. 100, 108).

[HWBM16b] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. “Dynamic Ontology-based Sensor Binding.” In: *Proceedings of the 20th East-European Conference on Advances in Databases and Information Systems (ADBIS)*. 2016 (cit. on pp. 48, 108).

[HWS+15] P. Hirmer, M. Wieland, H. Schwarz, B. Mitschang, U. Breitenbücher, F. Leymann. “SitRS - A Situation Recognition Service Based on Modeling and Executing Situation Templates.” In: *Proceedings of the 9th Symposium and Summer School On Service-Oriented Computing (SummerSOC)*. 2015 (cit. on p. 112).

[IC417] IC4F Consortium. *IC4F Research Project*. Online. 2017. URL: `https://www.ic4f.de` (cit. on pp. 50, 157).

[IEE10] IEEE. *Information technology – Smart transducer interface for sensors and actuators – Common functions, communication protocols, and Transducer Electronic Data Sheet (TEDS) formats*. Standard. IEEE, 2010. URL: `http://standards.ieee.org/findstds/standard/21450-2010.html` (cit. on p. 50).

[IFT11] IFTTT. *IFTTT: Put the internet to work for you*. online. 2011. URL: `https://ifttt.com` (cit. on p. 69).

[Jaz14] N. Jazdi. “Cyber physical systems in the context of Industry 4.0.” In: *IEEE International Conference on Automation, Quality and Testing, Robotics*. May 2014, pp. 1–4 (cit. on p. 49).

[JS 13] JS Foundation. *Flow-based programming for the Internet of Things*. online. 2013. URL: `https://nodered.org` (cit. on p. 69).

[JSA+18] C. Jennings, Z. Shelby, J. Arkko, A. Keranen, C. Bormann. *Sensor measurement lists (SenML)*. Internet Engineering Steering Group (IESG), 2018 (cit. on p. 50).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery – A Modeling Tool for TOSCA-based Cloud Applications.” In: *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dec. 2013, pp. 700–704 (cit. on pp. 35, 88, 95, 139).

[KBF+15] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, S. Taneja. “Twitter Heron: Stream Processing at Scale.” In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 239–250 (cit. on p. 31).

[Kle17] S. Klein. *IoT Solutions in Microsoft’s Azure IoT Suite*. Springer, 2017 (cit. on pp. 49, 60, 142).

[KRRS96] G. Kappel, S. Rausch-Schott, W. Retschitzegger, M. Sakkinen. “From Rules To Rule Patterns.” In: *Advanced Information Systems Engineering*. Ed. by P. Constantopoulos, J. Mylopoulos, Y. Vassiliou. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 99–115 (cit. on pp. 144, 150).

[Kut18] I. Kutger. *Human Tasks für OpenTOSCA zum Aufsetzen von IoT-Anwendungen*. Universität Stuttgart. Bachelor thesis. 2018 (cit. on pp. 98, 106, 139).

[LAB+18] A. Liebing, L. Ashauer, U. Breitenbücher, T. Günther, M. Hahn, K. Képes, O. Kopp, F. Leymann, B. Mitschang, A. C. Franco da Silva, R. Steinke. “The SmartOrchestra Platform: A Configurable Smart Service Platform for IoT Systems.” In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC)*. 2018 (cit. on p. 157).

[LCG+09] R. Lange, N. Cipriani, L. Geiger, M. Großmann, H. Weinschrott, A. Brodt, M. Wieland, S. Rizou, K. Rothermel. “Making the World Wide Space Happen: New Challenges for the Nexus Context Platform.” English. In: *Proceedings of the 7th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom ’09). Galveston, TX, USA. March 2009*. 2009 (cit. on p. 112).

[LCW08] D. Lucke, C. Constantinescu, E. Westkämper. “Manufacturing Systems and Technologies for the New Frontier: The 41st CIRP Conference on Manufacturing Systems May 26–28, 2008, Tokyo, Japan.” In: ed. by M. Mitsuishi, K. Ueda, F. Kimura. London: Springer London, 2008. Chap. Smart Factory - A Step towards the Next Generation of Manufacturing, pp. 115–118 (cit. on p. 130).

[LF98] D. C. Luckham, B. Frasca. “Complex Event Processing in Distributed Systems.” In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998), p. 16 (cit. on p. 66).

[LFWW16] F. Leymann, C. Fehling, S. Wagner, J. Wettinger. “Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!” In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science*. SciTePress, Apr. 2016, pp. 7–15 (cit. on p. 32).

[Lig17] R. A. Light. “Mosquitto: server and client implementation of the MQTT Protocol.” In: *The Journal of Open Source Software* 2.13 (2017), p. 265 (cit. on pp. 87, 100).

[Lin17] Linux Foundation Collaborative Project. *IoTivity*. online. 2017. URL: `https://www.iotivity.org` (cit. on pp. 30, 50).

[LL15] I. Lee, K. Lee. “The Internet of Things (IoT): Applications, investments, and challenges for enterprises.” In: *Business Horizons* 58.4 (2015), pp. 431–440 (cit. on p. 18).

[LLS08] G. T. Lakshmanan, Y. Li, R. Strom. “Placement Strategies for Internet-Scale Data Stream Systems.” In: *IEEE Internet Computing* 12.6 (2008), pp. 50–60 (cit. on p. 32).

[Lo88] V. M. Lo. “Heuristic Algorithms for Task Assignment in Distributed Systems.” In: *IEEE Transactions on Computers* 37.11 (Nov. 1988), pp. 1384–1397 (cit. on pp. 72, 73).

[Luc01] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001 (cit. on pp. 18, 26, 30, 31, 40, 111, 112, 144).

[Luc11] D. C. Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011 (cit. on p. 31).

[Luc19] D. Luckham. *What's the Difference Between ESP and CEP?* Online. June 2019. URL: http://www.complexevents.com/2019/07/15/whats-the-difference-between-esp-and-cep-2 (cit. on pp. 30, 31).

[LVCD13] F. Li, M. Vögler, M. Claeßens, S. Dustdar. "Towards Automated IoT Application Deployment by a Cloud-Based Approach." In: *Proceedings of the 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. SOCA '13. IEEE Computer Society, 2013, pp. 61–68 (cit. on p. 107).

[Mac97] S. MacGuire. "Big Brother: A Web-based UNIX System and Network Monitor." In: *Sys Admin* 6.3 (Mar. 1997), pp. 43–54 (cit. on p. 126).

[Mah18] S. Mahmoodi. "A Canonical Language for Complex Event Processing Systems." MA thesis. Universität Stuttgart, 2018 (cit. on p. 120).

[Man+13] B. Mandler et al. "COMPOSE–A Journey from the Internet of Things to the Internet of Services." In: *27th International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2013, pp. 1217–1222 (cit. on p. 69).

[MC13] A. McEwen, H. Cassimally. *Designing the Internet of Things*. 1st. Wiley Publishing, 2013 (cit. on p. 54).

[Meu95] R. Meunier. "The pipes and filters architecture." In: *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co. 1995, pp. 427–440 (cit. on pp. 41, 62).

[MIV+14] S. Mayer, N. Inhelder, R. Verborgh, R. Van de Walle, F. Mattern. "Configuration of smart environments made simple: Combining visual modeling with semantic metadata and reasoning." In: *2014 International Conference on the Internet of Things (IOT)*. Oct. 2014, pp. 61–66 (cit. on p. 60).

[MIVV14] S. Mayer, N. Inhelder, R. Verborgh, R. Van de Wallet. "User-friendly Configuration of Smart Environments." In: *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*. Mar. 2014, pp. 163–165 (cit. on p. 60).

[MMST16] J. Mineraud, O. Mazhelis, X. Su, S. Tarkoma. “A gap analysis of Internet-of-Things platforms.” In: *Computer Communications* 89 - 90 (2016). Internet of Things: Research challenges and Solutions, pp. 5–16 (cit. on pp. 49, 60, 70).

[MNH+13] H. McDonald, C. Nugent, J. Hallberg, D. Finlay, G. Moore, K. Synnes. “The homeML suite: shareable datasets for smart home environments.” In: *Health and Technology* 3.2 (2013), pp. 177–193 (cit. on pp. 50, 60).

[MS17] S. T. March, G. D. Scudder. “Predictive maintenance: strategic use of IT in manufacturing organizations.” In: *Information Systems Frontiers* (2017), pp. 1–15 (cit. on p. 51).

[MVD+14] M. Maksimović, V. Vujović, N. Davidović, V. Milošević, B. Perišić. “Raspberry Pi as Internet of things hardware: performances and constraints.” In: *Proceedings of 1st International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*. Vol. 3. 8. 2014 (cit. on p. 57).

[MW15] N. Marz, J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015 (cit. on pp. 18, 107, 140).

[Nel16] R. Nelson. “IBM Watson takes to the road.” In: *EE-Evaluation Engineering* 55.7 (2016), pp. 32–33 (cit. on pp. 49, 60, 142).

[NFD+07] C. D. Nugent, D. D. Finlay, R. J. Davies, H. Y. Wang, H. Zheng, J. Hallberg, K. Synnes, M. D. Mulvenna. “homeML – An Open Standard for the Exchange of Data Within Smart Environments.” In: *Pervasive Computing for Quality of Life Enhancement: 5th International Conference On Smart Homes and Health Telematics, ICOST 2007, Nara, Japan, June 21-23, 2007. Proceedings*. Springer, 2007. Chap. Pervasive Computing for Quality of Life Enhancement, pp. 121–129 (cit. on p. 60).

[NPP+17] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, R. H. Campbell. “Samza: stateful scalable stream processing at LinkedIn.” In: *Proceedings of the VLDB Endowment* 10.12 (2017), pp. 1634–1645 (cit. on p. 31).

[NSM17] D. Nicklas, T. Schwarz, B. Mitschang. “A Schema-Based Approach to Enable Data Integration on the Fly.” In: *International Journal of Cooperative Information Systems* 26.01 (2017), p. 1650010 (cit. on p. 50).

[OAS10] OASIS. *Web Services – Human Task (WS-HumanTask) Specification Version 1.1*. Online. 2010. URL: http://docs.oasis-open.org/bpel4people/ws-humantask-1.1-spec-cs-01.html (cit. on pp. 26, 93, 97).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0*. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (cit. on pp. 21, 26, 33, 39, 61, 69, 71, 93).

[OGC14] OGC. *Sensor Model Language (SensorML)*. online. 2014. URL: http://www.opengeospatial.org/standards/sensorml (cit. on p. 50).

[one18] oneM2M Partners. *oneM2M Base Ontology*. oneM2M, 2018. URL: http://www.onem2m.org/technical/latest-drafts (cit. on pp. 50, 61).

[Ope16] Open Connectivity Foundation. *AllJoyn Open Source Project*. Online. 2016. URL: https://openconnectivity.org/developer/reference-implementation/alljoyn (cit. on p. 30).

[Ope17] OpenFog Consortium Architecture Working Group. “OpenFog Reference Architecture for Fog Computing.” In: *OPFRA001* 20817 (2017), p. 162 (cit. on pp. 18, 21, 26, 133, 148).

[Pea84] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984 (cit. on p. 80).

[PLS+06] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer. “Network-Aware Operator Placement for Stream-Processing Systems.” In: *22nd International Conference on Data Engineering (ICDE’06)*. Apr. 2006, pp. 49–49 (cit. on p. 32).

[Pru07] M. Pruett. *Yahoo! pipes*. O’Reilly, 2007 (cit. on p. 69).

[Pup05] Puppet. *Unparalleled infrastructure automation and delivery*. Online. 2005. URL: https://puppet.com/# (cit. on p. 71).

[PVC+14] J. L. Pérez, Á. Villalba, D. Carrera, I. Larizgoitia, V. Trifa. “The COMPOSE API for the internet of things.” In: *Proceedings of the 23rd International Conference on World Wide Web*. ACM. 2014, pp. 971–976 (cit. on p. 69).

[Ran+18] R. Ranjan et al. “The Next Grand Challenges: Integrating the Internet of Things and Data Science.” In: *IEEE Cloud Computing* 5.3 (2018), pp. 12–26 (cit. on p. 18).

[Ras09] Raspberry Pi Foundation. *Raspberry Pi*. Online. 2009. URL: `https://www.raspberrypi.org` (cit. on p. 56).

[RGSE14] F. Ramparany, F. Galan Marquez, J. Soriano, T. Elsaleh. “Handling smart environment devices, data and services at the semantic level with the FI-WARE core platform.” In: *2014 IEEE International Conference on Big Data (Big Data)*. IEEE. 2014, pp. 14–20 (cit. on pp. 49, 60, 100, 142).

[Riz+10] S. Rizou et al. “Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks.” In: *Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, 2010 (cit. on p. 90).

[Rob16] Robert Bosch GmbH. *Bosch XDK Node*. Online. 2016. URL: `https://www.arduino.cc` (cit. on p. 56).

[SBH16] F. Samie, L. Bauer, J. Henkel. “IoT Technologies for Embedded Computing: A Survey.” In: *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES ’16. Pittsburgh, Pennsylvania: ACM, 2016, 8:1–8:10 (cit. on p. 57).

[SBS+17] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, C. Gniady. “Flow-based programming for IoT leveraging fog computing.” In: *2017 IEEE 26th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE. 2017, pp. 74–79 (cit. on p. 69).

[Sch18] J. Schneider. *Finden einer geeigneten Infrastruktur für Datenoperationen in IoT-Umgebungen*. Universität Stuttgart. Bachelor thesis. 2018 (cit. on pp. 90, 139).

[SGL+11] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, V. Nanayakkara. “Siddhi: A second look at complex event processing architectures.” In: *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM. 2011, pp. 43–50 (cit. on p. 32).

[SI02] X. Su, L. Ilebrekke. “A Comparative Study of Ontology Languages and Tools.” In: *Advanced Information Systems Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 761–765 (cit. on p. 62).

[SK17] K. J. Singh, D. S. Kapoor. “Create Your Own Internet of Things: A survey of IoT platforms.” In: *IEEE Consumer Electronics Magazine* 6.2 (Apr. 2017), pp. 57–68 (cit. on pp. 54, 57, 60, 70).

[SKH+15] J. Soldatos, N. Kefalakis, M. Hauswirth, et al. “OpenIoT: Open Source Internet-of-Things in the Cloud.” English. In: *Interoperability and Open-Source Solutions for the Internet of Things*. Springer International Publishing, 2015 (cit. on p. 60).

[Sma12] SmartThings Inc. *Samsung SmartThings*. Online. 2012 (cit. on p. 30).

[Sma16] SmartOrchestra Consortium. *SmartOrchestra Research Project*. Online. 2016. URL: `http://smartorchestra.de/en` (cit. on p. 50).

[SMP09] N. P. Schultz-Møller, M. Migliavacca, P. Pietzuch. “Distributed complex event processing with query rewriting.” In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. ACM. 2009, p. 4 (cit. on p. 66).

[SSF17] C. Stach, F. Steimle, A. C. Franco da Silva. “TIROL: The Extensible Interconnectivity Layer for mHealth Applications.” In: *Information and Software Technologies*. Ed. by R. Damaševičius, V. Mikašytė. Springer International Publishing, 2017, pp. 190–202 (cit. on p. 30).

[TCZN15] F. Tao, Y. Cheng, L. Zhang, A. Y. C. Nee. “Advanced manufacturing systems: socialization characteristics and trends.” In: *Journal of Intelligent Manufacturing* (2015), pp. 1–16 (cit. on p. 18).

[Thr14] Thread Group. *Thread – a low-power wireless mesh networking protocol for IoT*. Online. 2014. URL: `https://www.threadgroup.org` (cit. on p. 30).

[TK07] J. Travis, J. Kring. *LabVIEW for everyone: graphical programming made easy and fun*. Prentice-Hall, 2007 (cit. on p. 69).

[TL11] K. Taylor, L. Leidinger. "Ontology-driven complex event processing in heterogeneous sensor networks." In: *The Semanic Web: Research and Applications*. Springer, 2011, pp. 285–299 (cit. on p. 130).

[TSM18] Q.-C. To, J. Soto, V. Markl. "A survey of state management in big data processing systems." In: *The VLDB Journal—The International Journal on Very Large Data Bases* 27.6 (2018), pp. 847–872 (cit. on p. 129).

[TTS+14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, D. Ryaboy. "Storm@Twitter." In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD '14. ACM, 2014, pp. 147–156 (cit. on p. 31).

[UGW02] J. D. Ullman, H. Garcia-Molina, J. Widom. *Database Systems: The Complete Book*. Upper Saddle River, 2002 (cit. on p. 126).

[Ulu19] S. Ulusal. *MBP2Go: Erweiterung der mobilen MBP-Applikation um Monitoring-Funktionalitäten*. Universität Stuttgart. Bachelor thesis. 2019 (cit. on p. 141).

[Uni07] Universität Oldenburg. *Odysseus – the event processing system*. Online. 2007. URL: `http://odysseus.informatik.uni-oldenburg.de/` (cit. on pp. 32, 121).

[VF13] O. Vermesan, P. Friess. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013 (cit. on pp. 18, 49, 142).

[VFG+13] O. Vermesan, P. Friess, P. Guillemin, H. Sundmaeker, M. Eisenhauer, K. Moessner, F. Le Gall, P. Cousin. "Internet of Things Strategic Research and Innovation Agenda." In: *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers, 2013, pp. 7–152 (cit. on pp. 29, 62).

[VSI+15] M. Vögler, J. M. Schleicher, C. Inzinger, S. Nastic, S. Sehic, S. Dustdar. "LEONORE–Large-Scale Provisioning of Resource-Constrained IoT Deployments." In: *Symposium on Service-Oriented System Engineering (SOSE)*. IEEE Computer Society. 2015, pp. 78–87 (cit. on p. 107).

[VSID16] M. Vögler, J. M. Schleicher, C. Inzinger, S. Dustdar. "A Scalable Framework for Provisioning Large-Scale IoT Deployments." In: *ACM Transactions on Internet Technology (TOIT)* 16.2 (Mar. 2016), 11:1–11:20 (cit. on p. 107).

[W3C05] W3C. *Semantic Sensor Network Ontology*. online. 2005. URL: `https://www.w3.org/2005/Incubator/ssn/ssnx/ssn` (cit. on pp. 50, 52, 61).

[W3C15] W3C. *IoT-Lite Ontology*. online. 2015. URL: `https://www.w3.org/Submission/2015/SUBM-iot-lite-20151126` (cit. on p. 60).

[Wag10] S. Wagner. *A Concept of Human-oriented Workflows*. Universität Stuttgart. Diplomarbeit. 2010 (cit. on pp. 106, 139).

[WK96] G. Widmer, M. Kubat. "Learning in the Presence of Concept Drift and Hidden Contexts." In: *Machine Learning* 23.1 (Apr. 1996), pp. 69–101 (cit. on p. 26).

[WMS12] S. Wahle, T. Magedanz, F. Schulze. "The OpenMTC framework – M2M solutions for smart cities and the internet of things." In: *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. June 2012, pp. 1–3 (cit. on p. 100).

[WSBL15] M. Wieland, H. Schwarz, U. Breitenbücher, F. Leymann. "Towards Situation-Aware Adaptive Workflows." In: *Proceedings of the 11th Workshop on Context and Activity Modeling and Recognition (COMOREA) IEEE Conference on Pervasive Computing (PerCom)*. 2015 (cit. on p. 157).

[WSO13] WSO2. *Siddhi – Stream Processing and Complex Event Processing Engine*. `https://github.com/siddhi-io/siddhi`. 2013 (cit. on p. 32).

[WZGP04] X. Wang, D. Q. Zhang, T. Gu, H. Pung. “Ontology Based Context Modeling and Reasoning Using OWL.” In: *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*. IEEE Computer Society, 2004 (cit. on p. 130).

[ZCB10] Q. Zhang, L. Cheng, R. Boutaba. “Cloud computing: state-of-the-art and research challenges.” In: *Journal of Internet Services and Applications* 1.1 (2010), pp. 7–18 (cit. on p. 32).

[ZHKL09] O. Zweigle, K. Häussermann, U.-P. Käppeler, P. Levi. “Supervised Learning Algorithm for Automatic Adaption of Situation Templates Using Uncertain Data.” In: *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. 2009 (cit. on p. 113).

All URLs were last followed on 08.11.2020.

List of Figures

List of Tables

List of Algorithms

List of Definitions