

Automated Application Deployment in heterogeneous IoT Environments by OpenTOSCA

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Claeßens

Matrikelnummer 0726649

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.Prof. Dipl.-Ing. Mag. Dr. Schahram Dustdar

Mitwirkung: Proj.Ass. Fei Li, Ph.D.

Univ.Ass. Dipl.-Ing. Michael Vögler

Wien, 22. April 2014

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Automated Application Deployment in heterogeneous IoT Environments by OpenTOSCA

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Claeßens

Registration Number 0726649

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.Prof. Dipl.-Ing. Mag. Dr. Schahram Dustdar
Assistance: Proj.Ass. Fei Li, Ph.D.
Univ.Ass. Dipl.-Ing. Michael Vögler

Vienna, 22. April 2014

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Markus Claeßens
Märzstraße 36, 1150 Wien

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

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgements

First of all I want to express my sincere gratitude to my advisor Prof. Schahram Dustdar for giving me the opportunity to conduct this thesis at the Distributed Systems Group. Many thanks also to my co-advisors Fei Li and Michael Vögler who had always taken the time to provide me with their expertise and valuable feedback to improve this work.

Furthermore I want to thank my fellow students for making the experience of working together in several project groups a very pleasant one. Finally I would like to give my sincere thanks to my family for their great support during the years of study.

Abstract

Today's Internet of Things (IoT) landscape is fragmented by proprietary solutions offered by different vertical industries. Gateway application environments were introduced to integrate the domain- and vendor-specific communication protocols, data models, and control devices with management level business processes. But rather than reducing the heterogeneity, they introduced additional, often proprietary communication protocols. Furthermore, the automation of application deployment in IoT framework environments is barely supported and the management procedures are not portable due to the lack of standardization. Thus, the operation of IoT solutions requires time consuming manual configuration and detailed knowledge about each gateway framework in use. Interestingly, quite similar problems appeared in the domain of cloud applications. The absence of standardized APIs led to heterogeneous cloud environments making vendor-specific application life-cycle management procedures necessary. Porting these procedures to a new environment resulted in a cost and time intensive task. The portability problem in cloud environments was perceived by industry and academics, leading to the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA), which provides a meta-model to define the topology of applications and their life-cycle management in a portable way.

The goal of this work is to investigate how TOSCA can be applied to the IoT domain to automate application deployment and life-cycle management in a portable and reusable way. The structure of IoT applications is composed of reusable components which capture detailed management knowledge. Their behaviour is defined by life-cycle management procedures which are defined in a portable and interoperable way. Furthermore we show how the device virtualization proposed in the IoT PaaS architecture can be accomplished by applying TOSCA.

To show the feasibility of this approach, a prototype that implements a building automation use case is developed, based on the OpenTOSCA runtime environment. The prototypical application consists of two Air Handling Unit (AHU) instances which are deployed onto two distinct gateway frameworks – Sedona and Niagara.

Kurzfassung

Die heutige Internet of Things (IoT) Landschaft wird durch proprietäre Lösungen, welche von unterschiedlichen Wirtschaftsbranchen entwickelt werden, fragmentiert. Gateways wurden eingeführt, um die domänen- und herstellerspezifischen Kommunikationsprotokolle, Datenmodelle und Steuergeräte zu integrieren. Aber anstatt die Heterogenität der Systeme zu verringern, wurden weitere, oftmals proprietäre Kommunikationsprotokolle eingeführt. Des Weiteren wird das automatische Deployment von Anwendungen von IoT Frameworks kaum unterstützt und Management-Prozeduren sind aufgrund fehlender Standardisierung nicht portabel. Somit erfordert der Betrieb von IoT Lösungen eine zeitaufwändige manuelle Konfiguration sowie detailliertes Wissen über jedes verwendete Gateway-Framework. Interessanterweise sind ähnliche Probleme im Gebiet der Cloud-Anwendungen entstanden. Das Fehlen von standardisierten APIs führte zu heterogenen Cloud-Umgebungen, welche herstellerspezifische Management-Prozeduren zur Steuerung von Anwendungslebenszyklen nötig gemacht haben. Das Portieren dieser Prozeduren in eine neue Umgebung war eine zeitaufwändige und teure Aufgabe. Das Portabilitätsproblem in Cloud-Umgebungen wurde von Wirtschaft und Wissenschaft wahrgenommen, was zur Entwicklung der OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) geführt hat. Diese Spezifikation stellt ein Meta-Modell zur Definition der Topologie von Anwendungen und den dazu gehörenden Management-Prozeduren zur Verfügung.

Das Ziel dieser Arbeit ist es zu untersuchen, wie TOSCA auf die IoT Domäne angewendet werden kann, um das Deployment von Anwendungen und das Lifecycle-Management auf portable und wiederverwendbare Art und Weise zu automatisieren. IoT Anwendungen sind aus wiederverwendbaren Komponenten zusammengesetzt, welche detailliertes Wissen über deren Management beinhalten. Das Verhalten ist durch Management-Prozeduren zur Steuerung des Lifecycles definiert, welche auf portable und kompatible Art und Weise umgesetzt werden. Des Weiteren wird gezeigt, wie die Virtualisierung, welche in der IoT PaaS Architektur konzipiert wurde, durch die Anwendung von TOSCA erreicht werden kann.

Um die Machbarkeit dieses Ansatzes zu zeigen, wird ein Prototyp auf Basis der OpenTOSCA Laufzeitumgebung entwickelt, welcher einen Anwendungsfall auf dem Gebiet der Gebäudeautomatisierung implementiert. Die prototypische Anwendung besteht aus zwei Klimagerät-Instanzen, welche auf zwei verschiedene Gateway Frameworks – Sedona und Niagara – deployed werden.

Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	Aim of the Work	2
1.3	Methodological approach	3
1.4	Organization	4
2	State of the Art	5
2.1	Deployment Languages & Frameworks	5
2.2	IoT integration approaches	8
2.3	Research based on TOSCA	10
2.4	Towards Automated IoT Application Deployment	12
3	Background & Analysis	13
3.1	IoT PaaS	13
3.1.1	Architecture	13
3.1.2	Providing Control Applications	15
3.2	Building Automation System Integration Levels	16
3.2.1	The data model	17
3.2.2	Centralized integration approaches at the management level	17
3.2.3	Decentralized integration approaches	19
3.3	Gateway Application Environments	19
3.3.1	Niagara Framework	20
3.3.2	Sedona Framework	23
3.3.3	Comparison	26
3.4	TOSCA	26
3.4.1	TOSCA Service Roles & Benefits	26
3.4.2	Use Cases	27
3.4.3	Service Templates	27
3.4.4	Example Topology Template	29
3.4.5	TOSCA Processing Environment	29
3.4.6	Roles Involved in Modeling a Cloud Application	30
3.5	OpenTOSCA Runtime Environment	30
3.5.1	Architecture & Processing Sequence	30

3.5.2	OpenTOSCA Ecosystem	32
4	Use Case Definition	33
5	Design	35
5.1	Definition of Life-cycle States & Management Operations	36
5.2	Node and Relationship Models by TOSCA	37
5.2.1	Hierarchical Node Model	39
5.2.2	Life-cycle Interfaces	40
5.2.3	Hierarchical Relationship Model	41
5.3	Architecture	42
5.3.1	OpenTOSCA Environment	42
5.3.2	Interfacing with Gateways & Control Applications	42
5.3.3	Sedona Environment	43
5.3.4	Niagara Environment	43
5.4	Life-cycle Management Procedures	46
5.4.1	Application Deployment	46
5.4.2	Application Termination	47
5.4.3	Application Management	48
6	Implementation	51
6.1	Structure of the Cloud Service Archive (CSAR)	51
6.2	Node & Relationship Type Definitions	53
6.2.1	Node Type Definitions	53
6.2.2	Node Type Implementations	56
6.2.3	Relationship Type Definitions	58
6.3	Implementing the IoT Application	59
6.3.1	Topology Template	60
6.4	AHU Controller Gateway Applications	65
6.4.1	Sedona Interface	65
6.4.2	Niagara Interface	67
6.5	Implementation Artifacts	69
6.5.1	Sedona	70
6.5.2	Niagara	70
6.5.3	Niagara Proxy	71
6.6	Implementing the Life-cycle Management Procedures	73
6.6.1	Application Deployment	74
6.6.2	Termination Plan	78
6.6.3	Management Plans	81
7	Deployment & Demonstration	85
7.1	Use Case Setup	85
7.2	Application Deployment & Management	86
7.3	Result	89

7.4	Possible Integration with IoT PaaS	90
8	Conclusion & Future Work	91
8.1	Conclusion	91
8.2	Future Work	92
	Bibliography	93

Introduction

1.1 Problem Statement

Today’s Internet of Things (IoT) landscape is fragmented by proprietary solutions offered by different vertical industries as depicted in Figure 1.1. Their domain specific network protocols and control devices are mainly used for local control scenarios where the majority of them rely on field buses rather than on IP networks [1]. Kastner et al. [2] defines modern Building Automation Systems (BAS) as “distributed systems where the control functionality is spread across a three level hierarchy”. Figure 1.2 shows a subset of BAS standards and their associated layers. Granzer et al. [3] explains this hierarchy as follows. Direct interaction with control devices like collecting measurement data or changing configuration parameters happens on the field level, whereas the execution of control loops and sequences happens on the automation level. Finally, global configuration and management tasks like visualization happen at the management level. Gateways (cf. Section 3.3) were introduced to deal with the resulting heterogeneity of hardware, communication protocols and data models by implementing required communication standards as well as integrate legacy systems. But these efforts again have led to many proprietary application runtime environments lacking standardized service management procedures. For example,

Residential				Commercial		Lighting	Industrial				Automotive	Metering	Verticals
X10	Zigbee	Z-Wave	Konnex	BACnet	Lonworks	DALI	Modbus	ProfieBus	DeviceNet	ControlNet	CAN-Bus	M-Bus	Proprietary

Figure 1.1: Vertically oriented solutions using field buses [1].

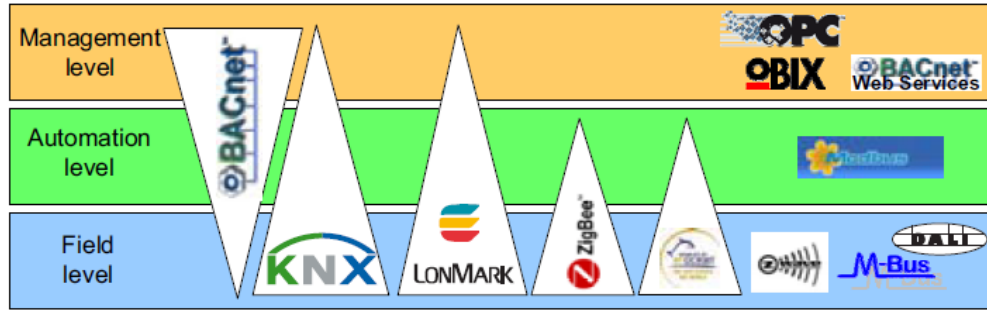


Figure 1.2: Technologies in BAS [3].

the Niagara^{AX} [4] framework provides middleware functionality to integrate BAS systems with management level business processes, but lacks the capability of automating the deployment process. Instead, a tool for semi-automatic device provisioning, specific to the framework and its proprietary protocols, is offered to system integrators. In contrast, the open source platform Sedona [5] - focusing on constrained devices - offers a protocol for low-level device provisioning and component management. The Internet of Things integration middleware (IoTSyS¹) [6] is a transparent multi-protocol gateway that integrates various sensor and actuator systems, which can be found in current home and building automation systems. The integration middleware provides a stack of communication protocols for embedded devices based on various standards to support interoperability that gets directly deployed on 6LoWPAN devices. But this framework again focuses on device integration rather than the application deployment aspect. In summary, the automation of application deployment in IoT framework environments is barely supported and the management procedures are not portable due to the lack of standardization.

Interestingly, the same problems appeared in the domain of cloud applications. The absence of standardized APIs led to heterogeneous cloud environments making vendor-specific application life-cycle management procedures necessary. Thus, porting these procedures to a new environment resulted in a cost and time intensive task. The portability problem in cloud environments was perceived by industry and academics, leading to the OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [7, 8] (cf. Section 3.4). It defines a meta-model for defining service topologies and facilitates the interoperable deployment and life-cycle management of application layer services. Although TOSCA was primarily developed to facilitate Cloud portability, an application to the IoT domain seems promising.

1.2 Aim of the Work

The goal of this thesis is to investigate how TOSCA can be applied to the IoT domain to improve the re-usability of service management processes and facilitate the automation of IoT application deployment. To show that the proposed mechanism works in practice, a prototypical building automation use case scenario consisting of a Sedona and a Niagara gateway will be developed.

¹<https://code.google.com/p/iotsys/>

Furthermore, this prototype will serve as practical proof to the approach described in [9] and thus will extend IoT PaaS [10] in future work (cf. Section 2).

The most important concepts of the prototypical implementation are:

- A hierarchical model of nodes and relationships describing components typically occurring in the BAS domain. These node and relationship definitions extend the current TOSCA node model.
- A *Topology Template* specifying the internal topology of the components of the complete use case and its properties. This application model is composed of reusable node definitions.
- Plans to deploy and un-deploy the complete use case respectively. These high-level life-cycle management procedures are composed of lower-level management operations of the affected nodes. Additional plans are used to perform life-cycle operations on the nodes of the application model.

The actual implementation of the life-cycle operations of the nodes used in the Topology Template will yield valuable information about the extent to which the details of TOSCA match the requirements of the IoT domain, e.g. cloud configuration management makes extensive use of scripts in contrast to IoT application deployment. Potentially necessary modifications to the TOSCA environment or the IoT frameworks will be discussed in this work as well as required interfaces and constraints which applications on specific IoT frameworks need to provide and satisfy respectively. This domain-specific application of the proposed approach will prove that the heterogeneity of IoT environments can be mitigated by applying TOSCA.

1.3 Methodological approach

The methodological approach consists of the following steps:

- First, current application approaches of the TOSCA specification are analyzed.
- Second, the deployment processes and management procedures of the IoT frameworks Sedona and Niagara are reviewed. A comparison of the open-source Sedona framework to the proprietary Niagara framework will bring up valuable knowledge to support generalization considerations regarding interfaces and the definition of the TOSCA type models. Additionally, a basic application will be developed for each framework to show case the application deployment in later steps.
- Third, Web services that provide access to the life-cycle management procedures of each IoT framework will be implemented. If necessary, modifications are made to the respective frameworks and applications to allow Web service access. These constraints caused by e.g. proprietary protocols will be discussed for each framework type.

- Fourth, the OpenTOSCA [11] Ecosystem v1.1² - a TOSCA runtime environment that implements a subset of the TOSCA specification - will be used to deploy and manage IoT framework applications of the use case. A Cloud Service Archive (CSAR) will be developed and processed by the OpenTOSCA Ecosystem. The CSAR will consist of a heterogeneous IoT application setup specifying Sedona and Niagara types. The implementation of the TOSCA types will use the Web services of the third step as Implementation Artifacts and the applications of the second step as Deployment Artifacts. BPEL Plans will be developed to invoke the life-cycle management procedures.
- Finally, the findings gained from the implementation of the CSAR and the modifications made to the environment are consolidated to discuss the portability and automation accomplished by this approach.

1.4 Organization

The remainder of this thesis is structured as follows:

- Chapter 2 discusses several application approaches of TOSCA as well as deployment languages and frameworks.
- Chapter 3 introduces important terms and concepts used in the Building Automation (BA) domain. Additionally, the TOSCA standard and its implementation in the OpenTosca runtime environment are discussed. Finally, the IoT PaaS is presented.
- Chapter 4 defines the prototypical use case scenario and explains its relevance to the BA domain.
- Chapter 5 comprises the high-level design of the prototype's architecture and the interaction of its components. A TOSCA Service Template describes the topology of the prototype.
- Chapter 6 describes the vendor-specific implementations of generic TOSCA nodes together with gateway-specific implementation artifacts needed to integrate the OpenTOSCA environment with the gateways used by the prototype. Finally, plans implementing life-cycle management procedures are presented.
- Chapter 7 evaluates the knowledge gained during the implementation of the prototype with regard to the application of TOSCA concepts. Furthermore, the extent to which these concepts facilitate the automation of application deployment in heterogeneous IoT environments is discussed. Finally, the possible integration of this prototype with IoT PaaS is explained.
- Chapter 8 recapitulates the findings of this thesis and gives an outlook to future research.

²<http://files.opentosca.de/v1.1/>

State of the Art

2.1 Deployment Languages & Frameworks

A Meta-Model Approach for the Deployment of Services-oriented Applications

The *Deployment Language for Services Applications (DLSA)* [12, 13] defines a meta-model to specify a language to describe applications together with deployment activities to face problems like services implementations selection, services dependencies and scheduling, which occur during application deployment in Service-Oriented Computing (SOC). A “services applications meta-model” [12] is used to describe *Services Applications* composed of several *Services*, and their *Interactions*. Each *Service* can contain a *Provided Interface* and several *Required Interfaces* [12]. As this approach is aimed at constrained Execution Environments (EEs), the main goal is to apply a sharing mechanism to re-use already deployed services to minimize the resource consumption of applications and thus maximize the number of installable applications. The DLSA provided by the meta-model is used by a *Deployment Manager for Services Applications (DMSA)* [12]. The deployment manager is realized with a centralized server and an embedded manager in each EE, which together can perform the following deployment activities: *Installation*, *Activation*, *De-Activation*, *De-Installation*, and *Management* [13]. Figure 2.1 describes the manager’s *Installation* activity that uses the service application model (application described by the DLSA) and the configuration of the targeted EE (collected by the embedded manager’s part), to calculate a deployment plan. A *scheduling algorithm* is used to resolve dependencies and check for inconsistencies (taking required interfaces into account) as well as to determine the right activation sequence for the participating services. For each service, the *selection algorithm* then selects a service implementation (e.g. jar file) from the *Repository*, favouring already installed implementations. The resulting deployment plan is then sent to and executed by the *Deployment Plans Manager* on the target platform [13]. Although this approach is quite comprehensive, services sharing is facilitated at the implementation level and thus doesn’t solve the problem induced by sharing of service instances that may comprise an interaction-sensitive internal state posing side-effects between applications [13].

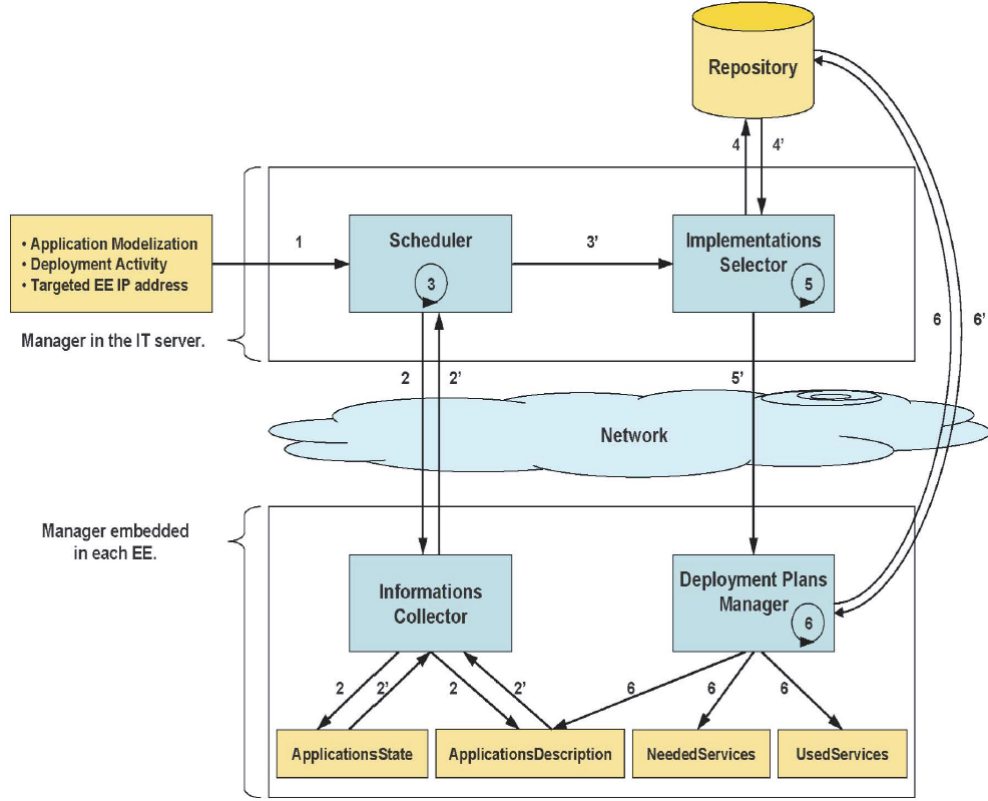


Figure 2.1: Manager's installation activity data exchanges [13].

A Flexible and Extensible Architecture for Device-Level Service Deployment

An approach for a *Deployment and Configuration system* [14] was developed as part of the middleware in the SOCRADES¹ research project. It addresses the challenges arising from remote deployment and configuration of services in constrained environments by dynamically selecting appropriate *Strategies* at runtime with respect to the current configuration of target devices. According to Frenken et al. [14], such a system must support the following use cases: *Service Publication, Updating, Querying, (Re)Mapping, Deployment, Execution, Monitoring* and *(De)Activation*. Additionally, *services, devices, and deployment objectives* are stated as the three dimensions of heterogeneity to be addressed by the proposed system. Taking these requirements into account, the high-level architecture depicted in Figure 2.2 arose. In this Deployment and Configuration system, the *System State* is aggregated from *User Input* (e.g. QoS constraints, services to map) on the one hand and from monitoring of the *Platforms* hosted on devices on the other hand. The *Monitor* keeps track of all nodes available within each *Platform*, the deployed service instances and the topology of the system. The *Mapper* then calculates a deployment plan based on the current *System State*, using concrete implementations selected by the *Strategy*.

¹<http://www.socrades.eu/>

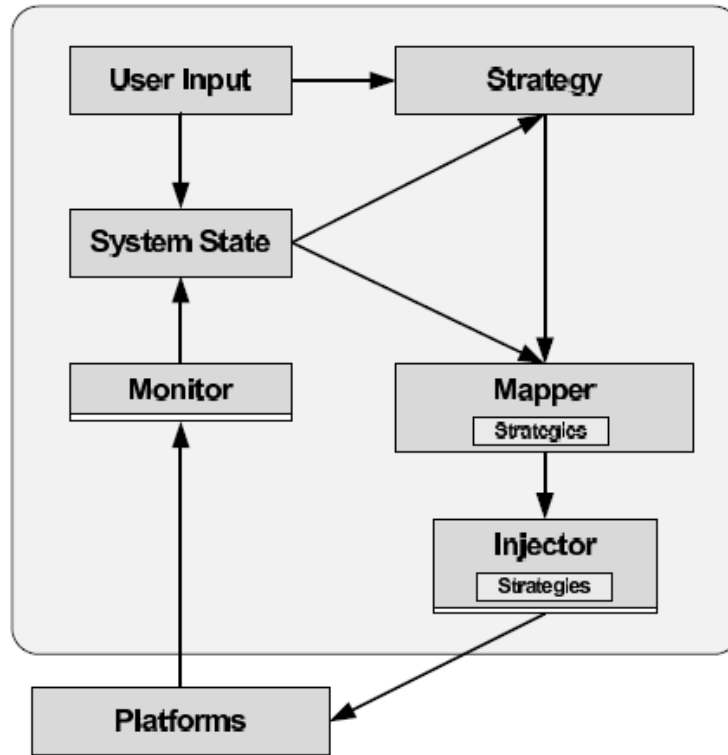


Figure 2.2: High-Level Architecture for Deployment and Configuration [14].

Finally, the *Injector* executes the deployment plan on the targeted *Platform*. The *Injector* and the *Monitor* both consist of a platform-specific and a platform-agnostic part [14]. To gain flexibility, the actual implementation splits the deployment planning phase in a *Node Selection* and a *Matchmaking* phase [14]. In the first phase, node selection strategies enrich the system state by marking each device with the service IDs it is capable to host at the moment. The Matchmaking phase then calculates a deployment plan by mapping the desired services to appropriate devices, taking service dependencies and requirements into account.

Cross-Platform Generative Agent Migration

To support the execution of mobile agents on heterogeneous platforms, Groot et al. [15] investigates the concept of *generative migration* introduced by Brazier et al. [16] that proposes the use of agent blueprints instead of sending complete code and data. A blueprint describes an agent’s structure and functionality in an implementation-independent way using XML syntax. An Agent Factory [16] service provided by target platforms automatically assembles an agent from the blueprint using platform-specific building blocks from a local repository. *Generative agent migration* thus “relies on homogeneity of libraries on different platforms to re-incarnate agents, but does not require homogeneity of platforms” [15].

2.2 IoT integration approaches

openHAB - Open Home Automation Bus

“openHAB² presents an integration platform that operates on a higher level of abstraction. The architecture is based on an event bus in combination with a publish-subscribe pattern, realized on OSGi. To integrate any kind of device of an IoT infrastructure, abstract items are defined to model these devices” [9]. Item Types like *Switch*, *Dimmer* or *Rollershutter* for blinds define Command Types like *OnOff*, *UpDown* or *IncreaseDecrease* to trigger a state change. The *Item Repository* provides state information of items to the *automation logic execution engine* and the user interface. In addition, bindings are used to bind items to concrete hardware, protocols or interfaces. This concept allows the platform to be vendor-neutral and hardware/protocol-agnostic. The openHAB environment can be configured via configuration files defining amongst others the items and automation rules.

oBIX - Open Building Information eXchange

The OASIS standard oBIX³ [17] provides technological-independent information modelling by using a common object model to represent devices in the domain of building automation [3, 4]. In the oBIX object model all devices are represented as objects holding collections of data points with a specified data type [18]. Additionally, the concept of *Contracts* “allow us to tag objects with normalized semantics and structure” [17]. The object types are directly mapped to XML element types in case of XML encoding. Constrained devices can make use of a binary encoding to reduce message size [17]. A simple Web service based protocol aligned to RESTful interactions together with XML encoding of the objects builds an interoperable, platform agnostic interface. A client can interact with an object on an oBIX server by addressing it through its uniform resource identifier (URI) and by using one of three request types, namely *read*, *write* and *invoke*, which map to the HTTP methods GET, PUT and POST [6]. Data and service discovery is provided through a central entry point (*http://server/obix* by convention), called the *Lobby* [17]. There, a client can register objects with the *WatchService* to retrieve real-time information using so called *watches*, which is a model for client polled eventing. Invoking the *pollChanges* operation on the *Watch* URI delivers the events, which occurred since the last poll [17].

A transparent IPv6 multi-protocol gateway to integrate Building Automation Systems in the Internet of Things

Jung et al. [6] presents the Internet of Things integration middleware (IoTSyS⁴) which is a transparent multi-protocol gateway combining the centralized with the decentralized approach (cf. Section 3.2). Management level business processes can control and monitor devices and their data through centralized interfaces, whereas device-level interfaces facilitate M2M communication. Transparent access to legacy devices is guaranteed by assigning IPv6 addresses to

²<http://www.openhab.org/>

³<http://www.obix.org/>

⁴<https://code.google.com/p/iotsys/>

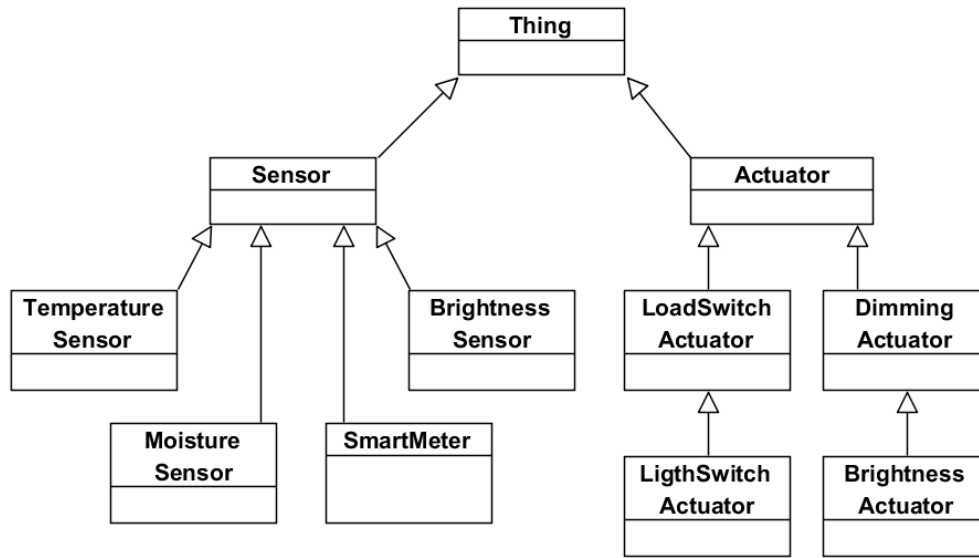


Figure 2.3: Excerpt of IoT contracts (adapted from [18]).

their interfaces [6]. *IoT contracts* (cf. Figure 2.3) define a generic set of custom oBIX contracts that represent functional blocks to model devices occurring in modern BAS systems. The most generic contract is the *Thing*, generalizing the *Sensor* and *Actuator* contract, which themselves group more specific contracts. The *LightSwitchActuator* contract of Listing 2.1 is an example for a specialization of the *Actuator* contract, specifying a writable boolean data point.

Listing 2.1: Light switching actuator contract [18].

```

1 <obj href="iot:LightSwitchActuator" is="iot:Actuator">
2   <bool name="value" href="value" val="false"
3     writable="true"/>
4 </obj>

```

Although these IoT contracts add semantics to the oBIX object model, the “semantic description of oBIX contracts is only intended for human beings but not usable for machine based semantic processing” [6]. Contracts like the *LightSwitchActuator* depicted in Listing 2.1 “can be compared to function blocks of BAS that describe standardized behaviors of sensors, actuators and control devices and define the semantics of input and output data points of devices that implement the function blocks” [6]. Thus, oBIX together with the IoT contracts form the top layer of the “IPv6 multi-protocol gateway stack” [6].

sMAP - a Simple Measurement and Actuation Profile for Physical Information

“Since in IoT Systems most devices use their own proprietary communication stack and interfaces, it is challenging to offer the gathered data in a standardized way. Dawson-Haggerty et

al. [19] proposes sMAP⁵ that tries to overcome this challenge by presenting physical information via RESTful interfaces using a simple JSON⁶ schema. This allows consumers to retrieve data, without the need to access the underlying infrastructure and dealing with proprietary formats” [9]. Additionally, a modified protocol stack is presented to support the execution of sMAP on constrained devices. sMAP voluntarily focuses on data representation rather than interpretation to allow for a wide use in distinct application domains.

BOSS: Building Operating System Services

“Based on sMap, Dawson-Haggerty et al. [20] presents BOSS, a distributed system that provides a collection of crucial, common and reusable services that enable the development of portable and robust applications for heterogeneous physical environment” [9]. A hardware presentation layer (HPL) adds metadata to the sMAP interface, thus making context-aware applications possible. Furthermore, the hardware abstraction layer (HAL) provides an *approximate query language* [21] to retrieve devices without knowing their network address. “The HAL also abstracts the logic used to control building components such as pumps, fans, dampers, chillers, using a set of drivers to provide standard interfaces. Drivers provide high-level methods such as `set_speed` and `set_temperature` that are implemented using command sequences and control loops over the relevant HPL points” [20].

2.3 Research based on TOSCA

“The research and application of TOSCA is still in its infancy. The early works are generally focused on exploring the possibilities of applying TOSCA for various management tasks, thus providing feedback to the standardization efforts and gaining experiences for industrial adoption” [9].

Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA

“Wettinger et al. [22] presents several concepts that integrate both model-driven cloud management and configuration management. The goal of the overall approach is to combine the advantages of these service management paradigms based on TOSCA” [9]. As a result, two implementation artifacts are provided for each life-cycle operation (cf. Figure 2.4). The first one consisting of configuration management tool specific configuration definitions (e.g. a Chef Recipe), whereas the second one programmatically encapsulates the first one in a wrapper script. Now, the TOSCA environment either can use the particular artifact (if it’s type is supported) gaining more control over its execution, or the wrapper script supported by any TOSCA environment. This solution leads to a high degree of portability [22].

⁵<https://code.google.com/p/smap-data/>

⁶<http://www.json.org/>

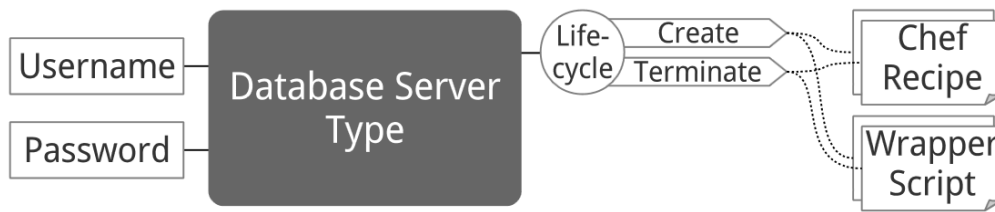


Figure 2.4: Combined integration results in two alternative implementation artifacts for each operation [22].

Improve Resource-Sharing through Functionality-Preserving Merge of Cloud Application Topologies

“Binz et al. [23] uses TOSCA to describe application topologies in a portable and manageable way. Based on this common TOSCA description the authors present an approach that merges two application topologies into one, to save resources by sharing similar components, but preserve the functionality of both applications” [9]. It is important to state that the nodes representing the business logic of a topology are not addressed by this process as there is no generic way to merge them. The merge of the supporting infrastructure nodes consists of an automated node matching phase followed by a manual evaluation phase undertaken by architects and developers which review correspondences yielded by the previous phase. Finally, a resource saving topology is generated by the merging phase. The proposed solution relies on plugins implementing type-specific logic to decide which components can be shared.

Pattern-based Runtime Management of Composite Cloud Applications

“Breitenbücher et al. [24] proposes an approach that enables the management of composite applications and their deployment on a higher level of abstraction. Furthermore the authors show how high and low level management tasks can be implemented separately and fully automated applied to the respective applications, by facilitating the features of TOSCA” [9]. Therefore, application management is divided into three layers of granularity, namely *Management Plans*, *Management Planlets* and *Management Operations*. Management Plans provide high-level management tasks by orchestrating low-level Management Operations which are tightly coupled to the components providing them. Small and recurring management tasks consisting of several Management Operations can be grouped into Planlets, forming generic building blocks for different applications. A globally accessible *Application State Model* keeps track of the application components’ states allowing Planlets to retrieve and modify the properties of components involved in their management tasks.

2.4 Towards Automated IoT Application Deployment

Efficient and scalable IoT service delivery on Cloud

“All introduced frameworks require considerable efforts to understand their application management process, and tedious manual configurations are a norm” [9]. “Thus, rather than proposing another “universal” architecture” [9], we proposed the IoT PaaS [10] architecture (cf. Section 3.1), “on which IoT solutions can be delivered as *virtual verticals* by leveraging computing resources and middleware services on cloud” [10]. Assuming “that IoT infrastructure is heterogeneous and will continue to be so” [9], “a methodology to easily integrate different domain-specific protocols and data models” [9] was developed.

Towards Automated IoT Application Deployment by a Cloud-based Approach

“Even worse than the situation in data exchange protocols, the deployment processes of an application can vary among IoT solutions even if the applications are realizing the same service” [9]. Thus, our latest work [9] demonstrates the feasibility of extending the application scope of TOSCA to IoT applications “to formally describe the internal topology of application components and the deployment process of IoT applications” [9] to manage the “heterogeneity in a coherent way” [9]. It demonstrates that “the node and relationship models can be shared for the same application, and the artifact models can be reused for gateways using the same software framework” [9].

Background & Analysis

In this Section we discuss several frameworks, architectures and protocols which are relevant to the research presented in this thesis. First of all, the IoT PaaS architecture is explained and the process of providing and managing control applications as IoT resources is discussed in detail. In a second step, the integration levels of Building Automation Systems (BAS) are explained. Afterwards, gateway application environments are considered in general followed by a detailed discussion of the two specific gateway frameworks – Niagara and Sedona – which will be part of the prototype. Next, the TOSCA specification will be discussed. TOSCA facilitates the portable definition of the structure as well as the behaviour of applications. Finally we introduce OpenTOSCA, an open-source implementation of the TOSCA processing environment, that we will use to showcase the results of the implementation work of this thesis.

3.1 IoT PaaS

As discussed in Section 1.1, “IoT services are often delivered in physically isolated verticals (often referred to as ‘silos’), in which hardware, middleware and application logics are tightly coupled to fulfill domain or even project-specific requirements” [9]. With IoT PaaS [10] we proposed a “novel IoT service delivery platform that leverages the service delivery model of PaaS cloud. On this architecture, we offer the possibility of providing end-to-end IoT solutions as *virtual verticals* on cloud, opposed to the traditional delivery model of physically-isolated and tightly-coupled vertical solutions” [9]. This means that “each IoT solution customer owns a virtually isolated solution which they can customize to their physical environments and devices” [10].

3.1.1 Architecture

The IoT PaaS architecture depicted in Figure 3.1 is described in a bottom-up manner. “The IoT infrastructure consists of networked tags, sensors, actuators, smart devices and so on” [10],

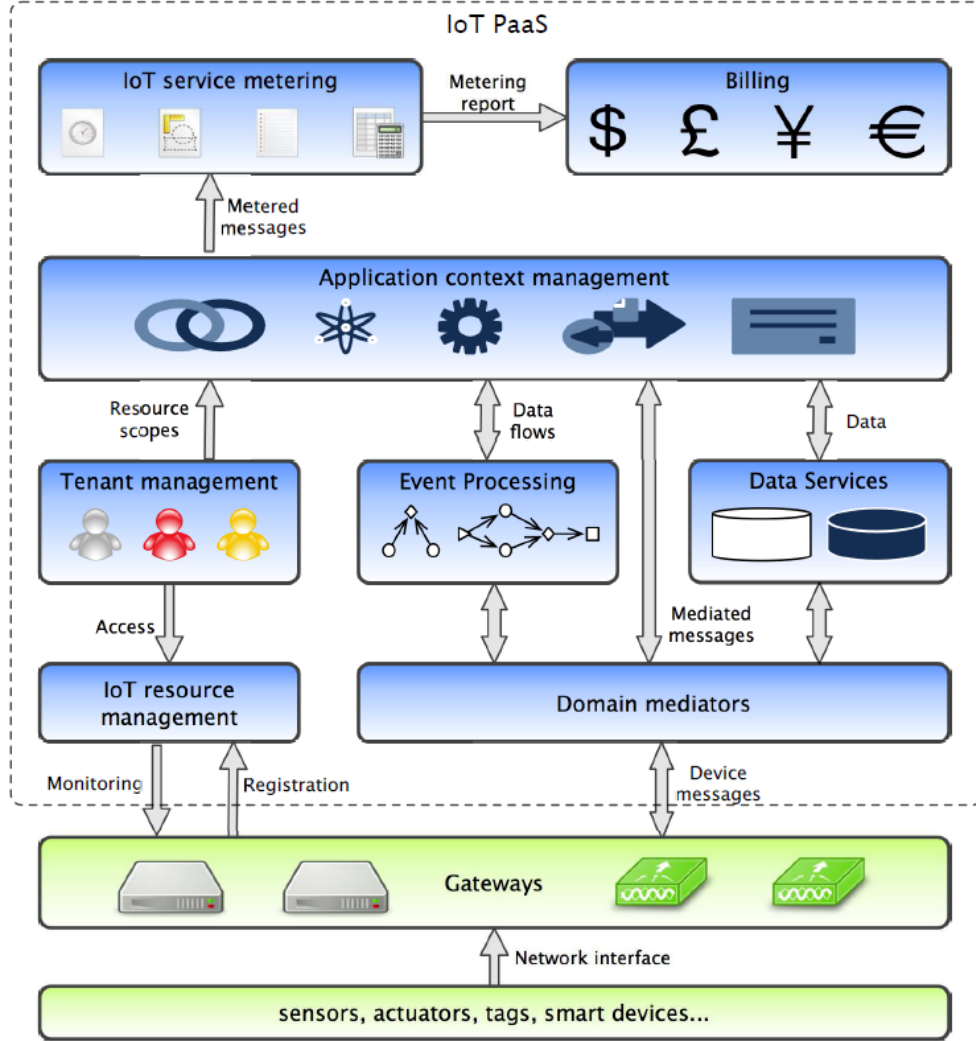


Figure 3.1: The IoT PaaS platform [10].

which are integrated with enterprise applications by Gateways (cf. Section 3.3). “The mechanisms for providing service interfaces for devices are generally referred to as *device virtualization* [25], since they effectively translate device and network interfaces to software interfaces” [10]. “*IoT resource management* provides a registration point for virtualized devices, gateways and control applications. The component monitors the resource status and enforces the access policies through gateways. Although most existing gateway solutions are intended to mitigate lower-level hardware and communication heterogeneity to a certain extent, the diversity of existing domain-specific data models has introduced another layer of heterogeneity. Therefore, we propose *domain mediators* to mediate the interfaces between different gateways in the same application domain. This mechanism allows IoT solutions to conform to the standardization efforts in various domains, such as oBIX for building management or Continua Health

Alliance (CHA)¹ for healthcare” [10]. “*Event processing* is to process and analyze real-time events generated by sensory devices” [10], whereas *Data services* provides access to persistent data. “*Tenant management* provides a consolidated view of the resources that are accessible by each tenant. In the IoT PaaS architecture, the resources include not only cloud resources such as virtual machines and software instances in traditional cloud offerings, but also IoT resources. Device capabilities and control applications can be provided to multiple tenants through virtualization. For instance, fire alarms can be shared between building management and emergency service of a city” [10]. “In the convergence of IoT and cloud, each application is running in a complex and dynamic context, which may encompass available IoT and cloud resources as well as software configurations. Thus, *Application context management* is focused on maintaining the optimal runtime resources and software configurations for applications” [10]. “The tenant management and application context management together give each IoT solution a virtually isolated operational environment, enacting the concept of virtual verticals” [10].

3.1.2 Providing Control Applications

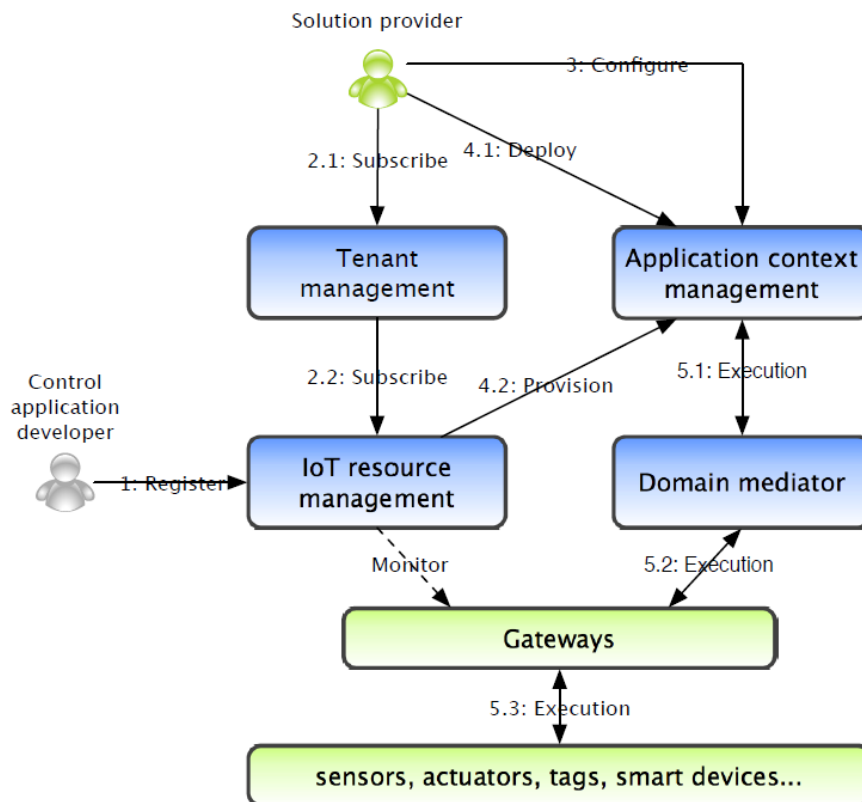


Figure 3.2: Providing control applications on IoT PaaS [10].

¹<http://www.continuaalliance.org/>

The process of providing control applications on the IoT PaaS architecture (cf. Figure 3.2) results in “highly reusable, multi-tenant control applications” [10] which are “managed within the overall service framework, in contrast to the vertical solutions in which applications are managed separately within each solution” [10].

First, a control application developer registers (Step 1 of Figure 3.2) new control applications to the *IoT resource management*. These control applications, posing IoT resources, can be used by solution providers to compose IoT services. Solution providers therefore have to subscribe (Steps 2.1 and 2.2) to the control applications they want to use in their virtual vertical solutions (via *Tenant management*). They define the application context by configuring (Step 3) the application parameters, like device IDs, of each control application. During the deployment step (Step 4.1), the *Application context management* uses this information to deploy the solution with all its subscribed control applications to the application context. Furthermore, *IoT resource management* monitors the availability of IoT resources to then provision (Step 4.2) the solution. Each application is executed (Step 5.1) in its own context using domain mediators to steer the related devices (Steps 5.1 and 5.2).

3.2 Building Automation System Integration Levels

BAS systems can be integrated using a centralized or a decentralized integration approach. The former uses a server to offer a centralized Web service interface for all devices behind it, whereas the latter provides each device with its own Web service interface, which allows the use of Web services to natively interact with the devices. Compared to the centralized approach, such field devices have an increased demand of computational resources, but therefore they can use Web services to communicate with each other. Jung et al. [6] “identifies four possible integration

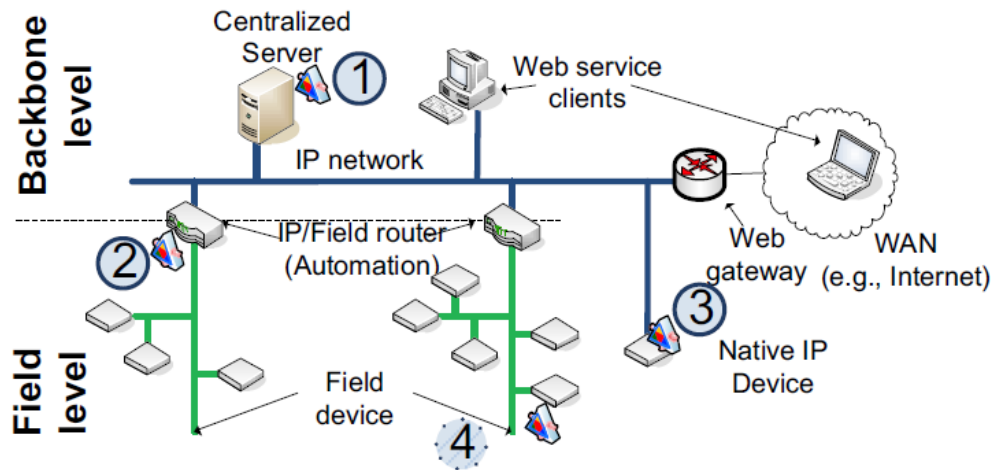


Figure 3.3: Centralized and decentralized integration [6].

approaches” (cf. Figure 3.3). (1) “A centralized server at the IP backbone of a BAS is the state of the art approach and allows the integration into enterprise systems and remote access” [6].

The Niagara Supervisor discussed in Section 3.3.1 follows this approach. (2) “Providing the Web service interface at the automation layer (which bridges the backbone and the field layer) via an IP/field router is a more decentralized approach but from the application layer aspects equivalent to the central deployment” [6]. The Niagara JACE gateway (cf. Section 3.3.1) is utilized at this layer. (3) “IP field devices using IP not only as data link but also as network layer protocol can be equipped with Web services and may offer this interface directly to other devices and Web service clients” [6]. Such devices are also called native IP devices. (4) “Finally, it is even possible to emulate Web service interfaces with a separate IPv6 address either at a centralized server or at an IP/field router acting as transparent gateway. Equipping this gateway with multiple protocol stacks and physical interfaces to different media allows to build a transparent multi-protocol gateway mentioned above” [6].

3.2.1 The data model

An integral part of each BAS is its *application model*. “The application models usually follow a data point approach, meaning that every device is expressed as a collection of input and output data points of well defined data types” [6]. The actual behaviour of field devices (e.g. a thermostat), can then be described by functional blocks assembled from several data points [6]. Although the majority of BAS systems use this approach, each solution comes with its own, usually incompatible implementation of the information model [1, 3, 6]. Standardization efforts are manifested in different data models and protocols discussed next.

3.2.2 Centralized integration approaches at the management level

“Within the domain of home and building automation, integration approaches using Web service technologies like oBIX, OPC UA² or BACnet/WS address this problem by providing a centralized Web service interface at the management tier of a typical BAS” [18].

Web Services for Building Automation and Control Networks (BACnet/WS)

The BACnet/WS standard [26] adds Web service capability to the BACnet standard to integrate data sources at the management tier. A generic data model facilitates the organization of data sources by representing them as hierarchical related nodes which hold their state as a collection of attributes. The concept of *normalized points* [26] allows to expose common point’s state information through common attribute names to allow data retrieval without knowing the exact details of the accessed data source. Additionally, generic access services provide access to nodes and attributes using an URL like path where a “path like ‘/East Wing/AHU #5/Discharge Temp’ identifies a node, and ‘/East Wing/AHU #5/Discharge Temp:InAlarm’ identifies the InAlarm attribute of that node” [26].

²<https://opcfoundation.org/about/opc-technologies/opc-ua/>

Open Building Information eXchange (oBIX)

As already mentioned in Section 2.2, the OASIS standard oBIX provides technological-independent information modelling by using a common object model to represent devices in the domain of building automation [3, 4]. The oBIX object model “defines 17 standard object types ranging from basic data items like `bool`, `int`, `real` and `str` over to more complex object types” [6]. One of the more complex data types is the `op` object, that is used to define operations with input and output object data type specified. The root `obj` element depicted in Listing 3.1 models a simple thermostat [17], where the `real` child elements model the space temperature sensor and the setpoint, whereas the `bool` element represents the furnace state. This oBIX document is clearly identified and accessible through the URI specified in its `href` attribute. In this example, each child element is tagged as `obix:Point` via the `is` attribute which is “a standard contract defined by oBIX for representing normalized point information. By implementing these contracts, clients immediately know to semantically treat these objects as points” [17].

Listing 3.1: oBIX representation of a thermostat [17].

```
1 <obj href="http://myhome/thermostat/">
2
3     <!-- spaceTemp point, current space temperature -->
4     <real name="spaceTemp" is="obix:Point"
5         val="76.0" status="fault"
6         unit="obix:units/fahrenheit"/>
7
8     <!-- setpoint point, desired temperature -->
9     <real name="setpoint" is="obix:Point"
10        val="72.0"
11        unit="obix:units/fahrenheit"/>
12
13     <!-- furnaceOn point, heating -->
14     <bool name="furnaceOn" is="obix:Point" val="true"/>
15 </obj>
```

The open-source *oBIX Toolkit*³ provides a Java software library for implementing oBIX enabled applications. The toolkit contains a data model for object trees, XML encoder/decoder, REST session management, and a Swing diagnostics tool. The oBIX Committee is already working on an improved oBIX version 2.0 [27], which will provide enterprise services based on new contract types. Furthermore, advanced reporting and aggregation to handle large data sets, enterprise alarm logic to allow more advanced alarm queries and enterprise scheduling to schedule interactions with building systems are some of the features planned for oBIX 2.0. “These contracts will be designed for more direct interaction with enterprise systems, able to participate in service oriented architectures (SOA) [...]” [27].

³<http://sourceforge.net/projects/obix/>

3.2.3 Decentralized integration approaches

Decentralized communication [6, 18] relies on per-device IPv6 interfaces used by native IP devices or transparent gateways like the IoT gateway discussed in Section 2.2. As the number of embedded IoT devices increases continuously, the standardization of a lightweight protocol stack is essential to handle the heterogeneity of these embedded devices. The Constrained Application Protocol (CoAP) discussed next is one of them.

Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) [28] is an application protocol that “is designed for machine-to-machine (M2M) applications such as smart energy and building automation” [28], targeting devices in constrained RESTful environments (CoRE). They often use 8-bit microcontrollers with very little memory and rely on lossy and low-power networks like 6LoWPAN [28]. In contrast to heavy-weight SOAP-based web services (WS-*) [25], CoREs come with a flattened communication stack lacking some features of WS-* standards, but therefore allowing the deployment on most constrained devices [6]. CoAP works “on a protocol stack based on IPv6 and UDP” [6] and comes with “built-in discovery of services and resources” [28] facilitating RESTful interaction with resources on field devices [25]. In comparison to HTTP based communication, which “provides reliability but limits the communication to a connection oriented point-to-point communication” [6], CoAP “provides unreliable packet-oriented communication with group communication and asynchronous interaction within the client/server communication model. Due to these differences, CoAP adds facilities for non-confirmed and confirmed message exchange and furthermore extends the regular HTTP protocol with an *observe* verb [29]. The enhancement supports observing a resource and avoids frequent polling of resources such as event streams or alarms” [6].

3.3 Gateway Application Environments

In modern building automation systems (BAS), gateways are used to simplify “the configuration, monitoring, and maintenance of heterogeneous systems” [4] and “enable high-value applications such as energy management and remote monitoring of equipment” [4]. Solutions typically consist of IP/field routers situated at the automation level [6]. These intelligent multi-protocol devices are needed to accommodate the high diversity of proprietary and standards-based communication protocols used by vendor-specific devices available today [4]. Therefore, different fieldbus drivers [4, 30, 31] enable access to commonly used network protocols (cf. Figure 1.1) like BACnet, LonWorks and Modbus, whereas protocols like HTTP and SOAP are typical solutions to facilitate the communications at the enterprise level [4].

First, the widely used Niagara^{AX} framework will be discussed in detail, counting more than 123.450⁴ instances deployed worldwide. Afterwards, the open source platform Sedona will be evaluated.

⁴<http://www.niagaraax.com/>, Accessed: 14.04.2014

3.3.1 Niagara Framework

The Niagara framework is defined as “a universal software infrastructure that allows companies to build custom, web-enabled applications for accessing, automating, and controlling smart devices in real time over the Internet” [32], where the term smart devices refers to sensors, actuators and metering systems. It “integrates diverse systems and devices (regardless of manufacturer or communication protocol) into a unified platform that can be easily managed in real time over the Internet (or intranet) using a standard web browser” [32]. The framework is “fully scalable, meaning that it can run on platforms spanning the range from small, embedded devices to enterprise class servers. Fields of application are energy-services, building-automation, industrial-automation and M2M applications” [32]. “Niagara is targeted for embedded systems capable of running a Java VM. This excludes some very low-end devices that lack 32-bit processors or have only several megabytes of RAM” [32].

Hardware and Protocol interconnectivity

To integrate diverse systems, a physical connection to a device’s network is required. The Java Application Control Engine (JACE) [33] refers to a family of embedded platforms [4], which provide connectivity to common network protocols such as LonWorks, BACnet, and Modbus, along with many proprietary networks. “Scalability and reliability concerns are avoided with the unique distributed architecture that a network of JACE devices creates” [33].

Semantic interoperability

Besides the hardware and protocol interconnectivity provided by JACE devices, “semantic interoperability is essential to allow the same tools to be used for monitoring and configuration—regardless of how this information is encoded or communicated” [4]. The Niagara framework deals with this issue by using a *common object model*, that can be seen as “a uniform, normalized database of objects” [4]. “The object model is a hierarchical composition of concepts in building automation, from the elementary level of simple data types to abstract concepts such as communication sessions and control schedules, [...] through which other applications interact with the various systems” [4]. Based on this object model, “a set of general services such as a real-time control engine, scheduling, alarming and Internet connectivity” [4] is provided. The common object model is referred to as a meta-protocol by [4], which evolved into the oBIX standard discussed in Section 3.2.2.

Architecture

The Niagara software architecture [4, 30, 32] consists of four layers. “The bottom layer [...] is the host platform, either a JACE controller or a PC” [32]. The second layer is a J2ME compliant Java virtual machine (JVM) hosting the Niagara runtime environment (NRE) [32]. A Niagara component application [30] is then executed as a so called Station in the NRE. Figure 3.4 outlines the communication protocols applied within the Niagara environment. The framework “includes a proprietary protocol called *Fox* which is used for all network communication between stations as well as between Workbench and stations. Fox is a multiplexed peer to peer protocol which

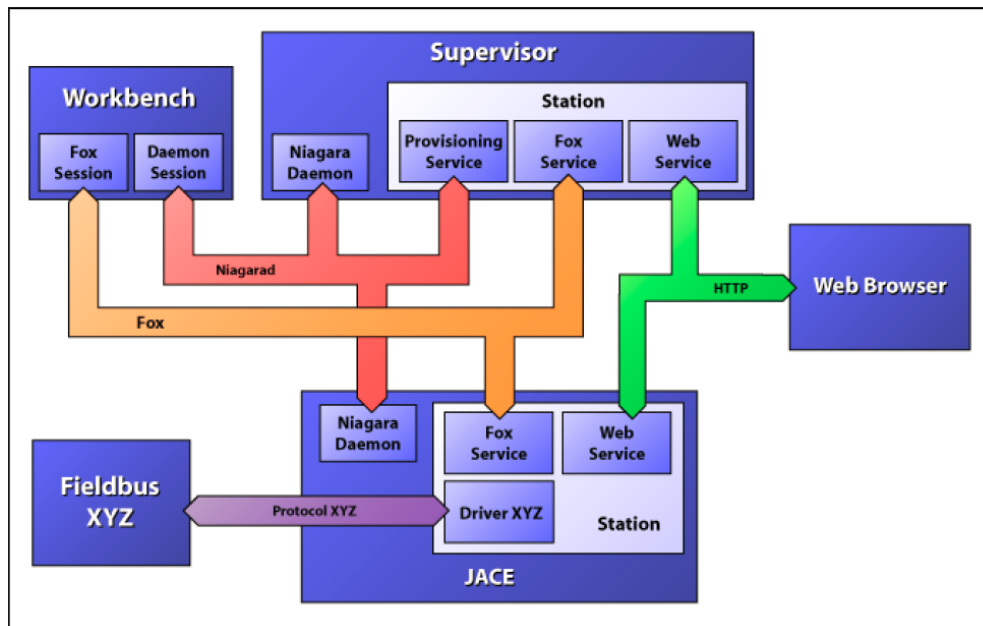


Figure 3.4: Niagara^{AX} communication and deployment [30].

sits on top of a TCP connection” [30]. A *Workbench* is a user interface to perform management tasks whereas the *Web Browser* provides restricted Workbench functionality. The latter provides data retrieved from the Stations Web service modules, which use HTTP and SOAP for communication. The *Niagara Daemon* provides commissioning and bootstrap functionality to Niagara platforms through the also proprietary *Niagarad* protocol [30]. Finally, the communication to devices connected to field buses is established via the *Driver* module. Next, the most important modules of the Niagara software stack depicted in Figure 3.5 will be described.

Software stack

Baja. The *Building Automation Java Architecture (Baja)* is “the core framework built by Tridium⁵ [which] is designed to be published as an open standard” [30]. “Fundamentally Baja is an open specification and the Niagara Framework is an implementation of that specification” [30]. The Niagara type system, as part of Baja, is built on top of the Java type system and identifies types in the *Object model* using the following format: {module name}:{type name} [30]. On top of the *Object model* the *Component model* allows to declaratively model control flows of applications and “integrate a wide range of physical devices, controllers, and primitive control applications including LonMark profiles, BACnet objects, and legacy control points” [30], by assembling the required components. To identify resources within the Niagara environment, *Object Resolution Descriptors (ords)* [30] of the *Naming* module are used, where

⁵www.tridium.com

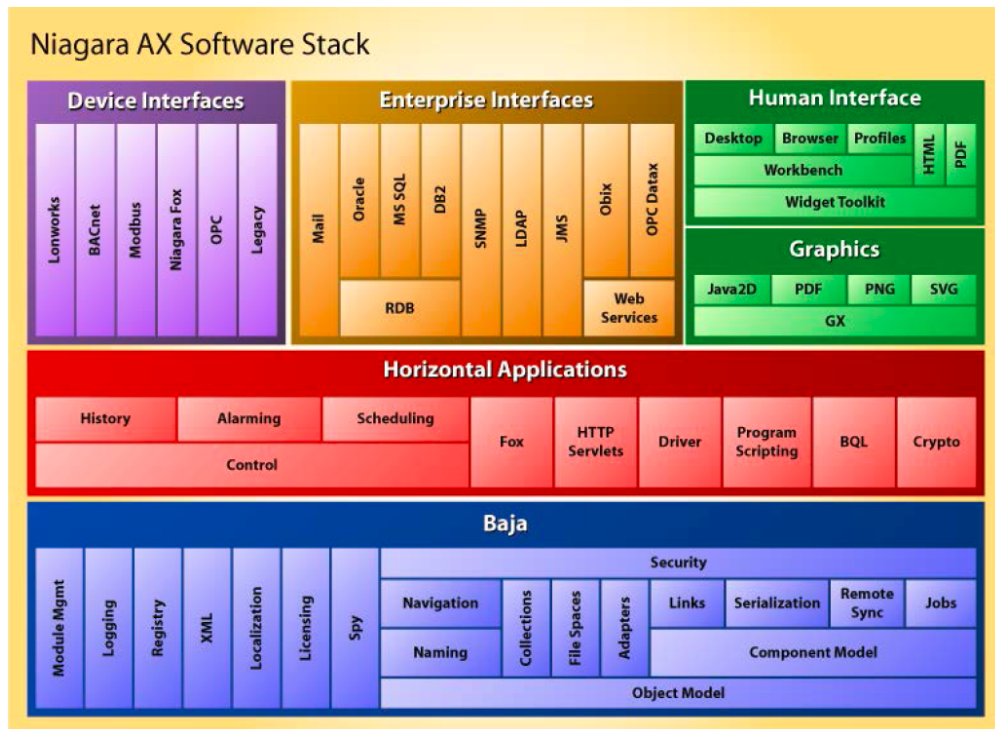


Figure 3.5: Niagara^{AX} Software Stack [30].

e.g. `local:|fox:|station:|slot:/Logic/Add` would identify an Add component in the Logic folder residing in the Station's root.

The *Registry* module provides information about all modules and types available in the current Niagara environment and describes how those types are related or provide functionality to each other. The registry's contents are updated at station startup from information contained within each module. The Niagara registry provides a convenient way to check if a desired type is available on the station and then create a new instance as depicted in Listing 3.2.

Listing 3.2: Look-up and instantiation of a Niagara Add component.

```
1 Sys.getRegistry().getType("control:Add").getInstance();
```

Horizontal Applications. Niagara comes with a broad library of standard components applicable to different M2M domains [30]. The fundamental concept of the *Control* and automation module are “normalized components for representing control points” [30]. “Control points are typically used [...] to read and write points in external devices” [30]. Figure 3.6 shows the graphical representation of a control point that is connected to an external temperature sensor's data point, as displayed in the *Niagara^{AX} Workbench* tool. The driver framework of the *Driver* module helps “to model and synchronize data with external devices or systems” [30], by abstracting from the various underlying communication protocols. There are three modules

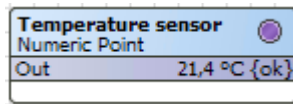


Figure 3.6: Niagara control point.

built on top of the *Control* module [30]. First, the *History* module helps with storing and accessing historical data collected from selected control points whereas the *Alarm* module provides lifecycle management of alarms triggered for example by an output value out of a specified range. Finally, the *Schedule* module is used to fire events or change values of control points in a recurring manner, like switching off the lights of a room every day at 8 pm.

Device & Enterprise Interfaces. A wide range of protocols are supported on the one hand by the *Device Interfaces* module, which integrates various field bus protocols such as Modbus or BACnet [30]. The *Enterprise Interfaces* module [30] on the other hand facilitates the integration of enterprise systems such as relational databases or Web services, where oBIX, as an important example, was discussed in Section 2.2 and 3.2.2.

Deployment process

A system integrator typically configures a gateway by using a Station setup wizard specifying the modules and drivers to be installed. This process yields a station database [30, 32] manifested in a single `config.bog` file, which contains a XML tree structure of the current configuration. On gateway startup, the corresponding Station is then booted from the `file:!stations/{stationName}/config.bog` file into the gateway’s VM. Now, an integrator can build and modify control applications by opening a remote connection using the *Niagara Workbench* tool and “program” the Station’s behaviour by assembling different components graphically. [34] summarizes how Niagara JACE devices can be leveraged to integrate lighting, lawn irrigation, heating and air conditioning (HVAC) and security systems in a house. An example of the graphical representation of a lighting control application, which reacts on daylight and door sensors and makes use of the *Schedule* component is depicted in Figure 3.7. The actual components are created by developers. In the contrary to the Sedona Framework discussed next, the ability of manipulating a Station’s component tree programmatically is restricted to Niagara’s proprietary Fox protocol and thus preventing direct connections from a custom client application outside of the Niagara environment.

3.3.2 Sedona Framework

The Sedona Framework [5] is an open source platform similar to the Niagara framework, but targeting different device platforms. The goal of the Sedona Framework is to support very low cost and low power embedded devices with lack of memory and restricted network access. To meet these requirements, deployed applications require less than 100KB of memory [5] and communication supports 6LoWPAN networks [5].

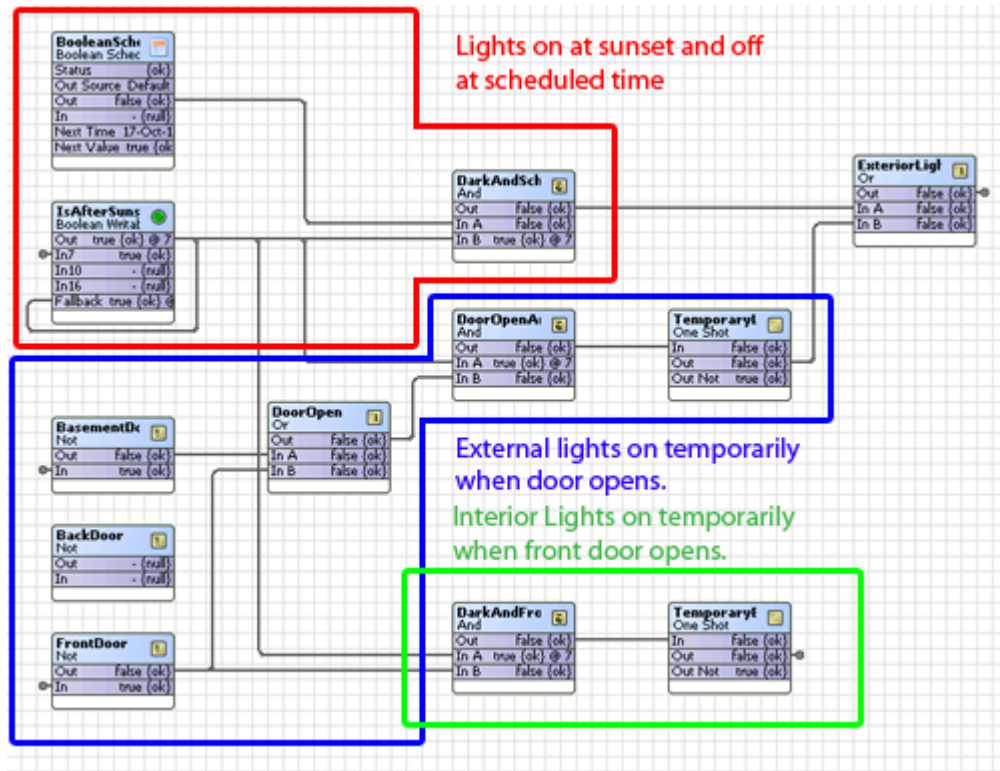


Figure 3.7: Niagara Home Automation example - Lighting control application [34].

Architecture

The UDP based *Sox*⁶ protocol avoids the memory footprint of TCP and relies on important properties like session management, flow control and reliability provided by the underlying *Datagram Authenticated Session Protocol*⁷ (DASP). The Sedona language is a Java-like, component and object oriented programming language designed for embedded platforms with limited resources [35].

Deployment process

Usually, the application deployment process is conducted by two different user groups. On the one hand, developers build components and package them into modules, so called *kits*. Selected kits are installed as a *kits.scode* image on the Sedona Virtual Machine (SVM) of the target Sedona device as illustrated in Figure 3.8. On the other hand, system integrators, which are often domain experts, use graphical tools like the Sedona Framework Workbench⁸ to build applications by assembling the former installed components. The Sedona component model⁹ allows

⁶<http://sedonadev.org/doc/sox.html>

⁷<http://sedonadev.org/doc/dasp.html>

⁸<http://www.sedonadev.org/products.html>

⁹<http://www.sedonadev.org/doc/apps.html>

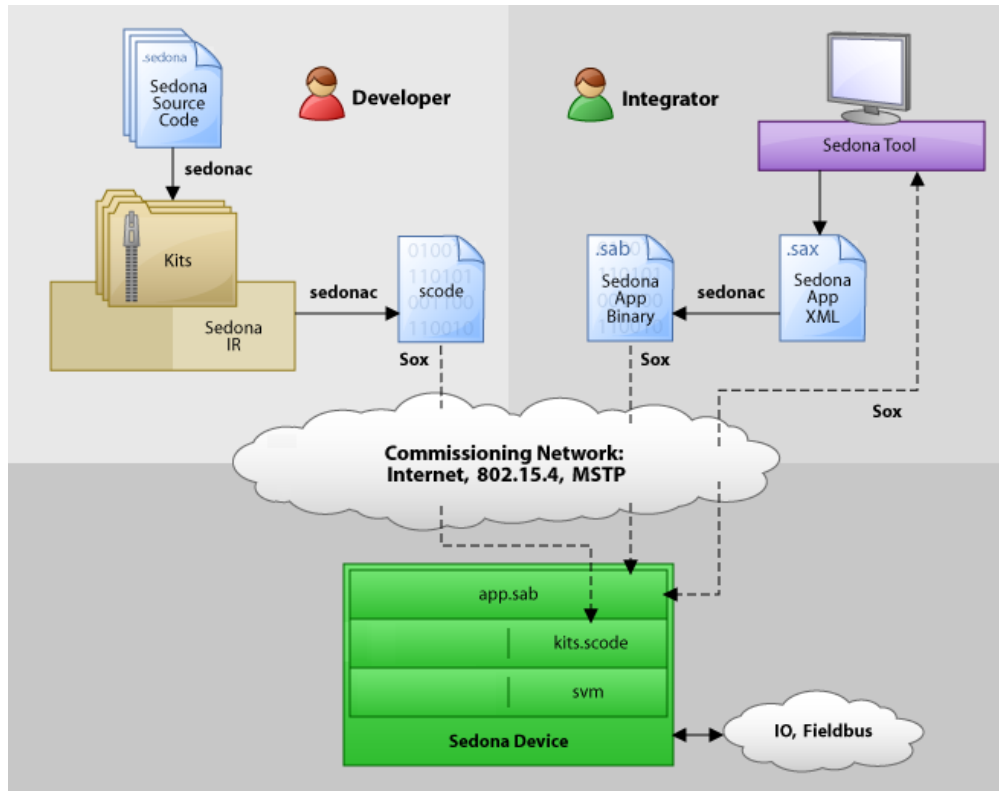


Figure 3.8: Sedona architecture and deployment process [36].

to define the control flow of an application (*app.sab*) in a declarative way by building a tree of components and thus separating it from the code packaged into the referenced kits. The Sox protocol [37] is used to retrieve information from Sedona framework-enabled devices and perform provisioning tasks on them. Furthermore, remote programming performed through the Java client implementation of the Dasp/Sox protocol stack allows full control over the deployed applications and their components. To modify applications at runtime, components can be looked up reflectively using the `kit::component` pattern (cf. Listing 3.3). Thus, a type registry is modelled by the reflection mechanism¹⁰. Consequently, the aforementioned roles of developers and system integrators could be merged, avoiding the use of a graphical assembly tool.

Listing 3.3: Look-up a Sedona type by its qualified name.

```
1 Type t = Sys.findType("control::Ramp")
```

¹⁰<http://www.sedonadev.org/doc/reflection.html>

3.3.3 Comparison

Both Niagara and Sedona use a component model to declaratively model control flows of applications and thus enable graphical programming using standard components. Even though information can be retrieved from and actions can be invoked on Niagara devices via the oBIX protocol, direct access to the object model from outside of the Niagara environment is impossible due to the proprietary Fox protocol. In contrast, the Sedona framework offers full access to its components via the open protocol Sox.

3.4 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [7, 8] “is a new OASIS standard for improving portability of cloud applications in face of growingly heterogeneous cloud application environments” [9]. Binz et al. [38] defines three major challenges TOSCA is dealing with in the area of IT service management:

1. Automated Application Deployment and Management,
2. Portability of Applications and their Management, and
3. Interoperability and Reusability of Components.

Automation is accomplished by workflows which contain expert knowledge about application management procedures, allowing non-experts to manage the applications life-cycle on a higher level of abstraction. *Portability* is assured by the formal definition of the applications structure and their components in a self-contained way. The definition of management tasks is based on widely used and standardized workflow languages. To support *Interoperability and Reusability*, components are defined with their implementations in a reusable way to facilitate application development by composing them. As with Portability, the self-contained packaging is imperative [38].

Throughout this section we describe the major TOSCA service roles as well as important use cases leveraging the advantages provided by TOSCA. Afterwards the TOSCA core concepts are explained and how they are applied by developers and processed by TOSCA execution environments.

3.4.1 TOSCA Service Roles & Benefits

The TOSCA standard defines three roles, namely *Cloud Service Developer*, *Cloud Service Provider*, and *Cloud Service Consumer* [8]. Cloud services are developed by Cloud Service Developers and provided to Cloud Service Consumers by Cloud Service Providers. The actors of this role model benefit from the application of TOSCA in several ways. Consumers gain more flexibility in their selection of an appropriate service provider and experience a cost reduction due to the automation of installation and maintenance tasks. The main advantage of facilitating TOSCA for service providers is to reduce the time to market of cloud services, as the deployment tasks are already specified. Furthermore, the generation of additional service instances is

simplified, leading to lower operational costs. Finally, Developers can build on the operational knowledge of cloud providers and are able to select different cloud providers [8].

3.4.2 Use Cases

In the following, the most important use cases leveraging the advantages of standardizing Service Templates [7] defined by TOSCA are discussed.

Services as Marketable Entities. By defining services in a standardized manner, they can be published in service catalogs, where customers can select from. Providers can adjust the interoperable definitions of the structure of services to map the topology to their hardware environment [7]. Likewise, the plans defined in the Service Templates are adjusted. Several kinds of plans are used, which are usually created by developers and adjusted by providers. Build plans are used to instantiate a service defined by a Topology Template, whereas termination plans are used to destroy them. Management plans provide means for life-cycle management. Thus, a customer can invoke these plans without understanding the inherent domain-knowledge, leading to a reduction of management costs [7].

Portability of Service Templates. Standardizing Service Templates makes service definitions portable. TOSCA defines portability as “the ability of one cloud provider to understand the *structure* and *behavior* of a Service Template created by another party” [7]. Hence, portability of the components of a service is not covered by TOSCA.

Service Composition. Service Templates allow for an abstraction from the concrete hosting environment, which facilitates the composition of a service from several other services hosted on different cloud providers [7].

3.4.3 Service Templates

The TOSCA specification defines a meta-model for defining portable services, facilitating the interoperable deployment and life-cycle management of application layer services [7, 8, 39]. A Service Template (cf. Figure 3.9) formally describes the structure (Topology Template) and behavior (Plans) of a service.

Topology Template

“The structure of a service is defined by the *Topology Template*, which consists of *Node Templates* and *Relationship Templates*. Together they represent a service by a directed graph” [9], which may consist of separated sub-graphs. “In this graph, every component is represented by a *Node Template* that instantiates a *Node Type*, which defines the properties and management operations of a component. To support re-usability, Node Types are defined separately and just referenced in Node Templates” [9]. The *Node Template* defines the properties like IP addresses or credentials required for instantiation as well as the *Capability Definitions* and *Requirement Definitions* to indicate which *Relationship type* can be applied to them and thus which and how many Node Templates can be related to them. “*Relationship Templates* specify the relationship among nodes in the Topology Template, where each Relationship Template refers to a separately

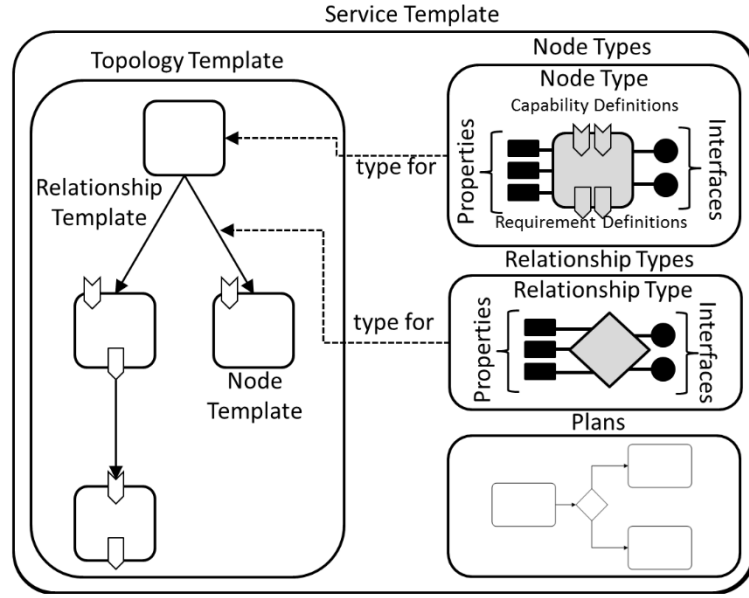


Figure 3.9: Structural Elements of a Service Template and their Relations [7].

defined Relationship Type, which in turn defines the semantics and any properties that can be used to represent a relationship, such as ‘dependOn’ or ‘connectTo’ ” [9].

“The actual scripts, configuration files and application archives required by an application are called *Artifacts*, which are explicitly specified in *Artifact Types* and *Artifact Templates*” [9]. TOSCA defines two different artifact types, namely *Implementation Artifacts* and *Deployment Artifacts*. Implementation Artifacts implement the management operations defined by the interfaces of node types, whereas Deployment Artifacts are installed into target environments realizing the instances of their nodes. Thus, the Implementation Artifacts are deployed into the TOSCA container before any life-cycle operation on nodes is invoked, as they are used to deploy Deployment Artifacts to the target environment [7]. These Artifacts in form of scripts, images, configuration files, or libraries amongst others are then referenced from *NodeTypeImplementations* and *RelationshipTypeImplementations* defining an implementation of a specific Node Type or RelationshipType respectively.

Plans

“The management process of creating, deploying and terminating a service can be defined by *Plans*” [9]. These workflows implement high-level management procedures by orchestrating low-level management operations of interfaces provided by Node Types or Relationship Types. To define these process models as portable and interoperable as possible, existing languages like *Business Process Model and Notation*¹¹ (BPMN) or *Business Process Execution Language*¹² (BPEL) are used [7].

¹¹<http://www.bpmn.org/>

¹²<http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf>

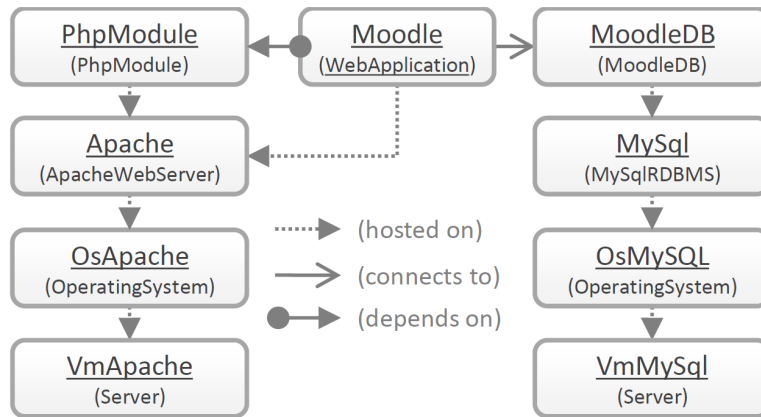


Figure 3.10: Moodle Application Topology [11].

3.4.4 Example Topology Template

Binz et al. [11] defines a Topology Template to describe the structure of the Web based e-learning platform Moodle (cf. Figure 3.10). Basically it tells us that the Moodle Web application is hosted on an Apache Web server which is hosted on an operating system installed on a virtual machine. The Web application depends on a PHP module hosted on the same Apache Web server and connects to a Moodle Database hosted on a different virtual machine. The Relationship Templates interconnect the Node Templates adding semantics, e.g. the “depends on” Relationship Template would establish a remote database connection, whereas a “hosted on” Template would e.g. deploy the Moodle PHP files to the Apache Web server.

Such Topology Templates can be interpreted by a TOSCA-compliant management environment which instantiates the Templates and manages their instances. TOSCA Processing Environments are discussed in the following.

3.4.5 TOSCA Processing Environment

“The topology templates, plans and artifacts of an application are packaged in a Cloud Service Archive (.csar file) and deployed in a TOSCA environment, which is able to interpret the models and perform specified management operations” [9]. Such a TOSCA runtime environment, also called TOSCA container, is hosted by a Cloud Service Provider [8].

Imperative vs. Declarative processing. Plans are only one way of specifying the management procedures of a Service Template. They explicitly define a sequence of steps needed to perform certain management procedures, which is known as *Imperative processing* [8]. In case of *Declarative processing*, the TOSCA environment has to “infer the correct topology and management procedure just by interpreting the topology template” [9], which is restricted to simple applications [11]. Following a strict Declarative style, the complete management logic is implemented by the TOSCA container, making application modeling easier [40]. On the other hand, the Imperative style allows more flexibility in the creation of complex management procedures, as the complete logic resides in the Service Template [40].

3.4.6 Roles Involved in Modeling a Cloud Application

The roles required to develop a cloud service are *Type Architect*, *Artifact Developer* and *Application Architect* [8], which specialize the role *Cloud Service Developer* introduced in Section 3.4.1. A *Type Architect* is specialized on the types of components and the relationships amongst them. For example, vendors can facilitate *Node Type Inheritance* to add product-specific management operations and properties to vendor-neutral node and relationship types previously defined by vendor consortia [8]. Thus, Type Architects create re-usable type definitions that include local management knowledge. An *Artifact Developer* can implement these type definitions by providing *NodeTypeImplementations* (or *RelationshipTypeImplementations*) consisting of *Implementation Artifacts* and *Deployment Artifacts* mentioned above, thus specifying all means to instantiate and manage the respective type. A vendor can now package the type definitions together with their corresponding implementations in a CSAR file. Subsequently, an *Application Architect* creates a service topology by composing these re-usable component definitions. Therefore, node and relationship templates referring to node node relationship types are defined to form the topology template of the service definition. A global application life-cycle management covering all management aspects is developed by orchestrating local management operations defined by type architects. Finally, such a service application itself can be perceived as a composable component by defining the boundary definitions of its topology template. A specific implementation of the TOSCA processing environment is discussed next.

3.5 OpenTOSCA Runtime Environment

OpenTOSCA^{13,14} [11] is an open-source implementation of the TOSCA processing environment supporting imperative processing of TOSCA-based applications. This means that Plans are used to define deployment and life-cycle management procedures (cf. Section 3.4.5).

3.5.1 Architecture & Processing Sequence

Now the process of deploying and instantiating a TOSCA application will be explained in detail (cf. Figure 3.11).

Application Deployment

First of all, the TOSCA application, packaged as Cloud Service Archive (CSAR), is uploaded to the OpenTOSCA container (e.g. using the Admin UI). Now the CSAR is unpacked and the files are saved to the Files store. The *Control* component processes the TOCSA definition files and invokes the *Implementation Artifact Engine* and the *Plan Engine*. As defined by the TOSCA specification, local management operations of Node Types and Relationship Types are implemented by Implementation Artifacts like scripts, Web services or external service invocations. In case of a Web service, which can be packaged in the CSAR as a Web Service Archive (WAR), the Implementation Artifact Engine uses an appropriate *Plugin* to deploy (cf. Figure 3.11, (a))

¹³www.opentosca.org

¹⁴<http://www.iaas.uni-stuttgart.de/OpenTOSCA>

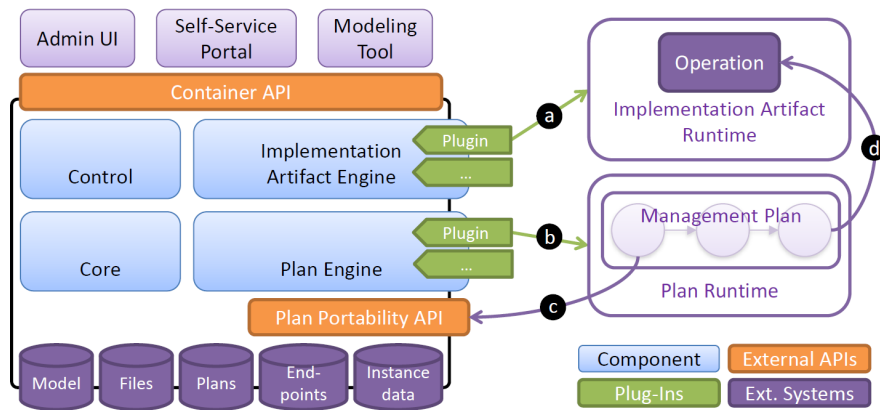


Figure 3.11: OpenTOSCA Architecture Overview and Processing Sequence [11].

the WAR file to the respective *Implementation Artifact Runtime*, e.g. a Tomcat web server. The endpoint of the implemented operation is persisted in the Endpoints database. Now the Plan Engine binds these endpoints to the service invocations defined in each Plan and deploys (cf. Figure 3.11, (b)) it to the related *Plan Runtime*. This deployment-time service binding is facilitated by an *Invoker service* which automatically selects an appropriate service implementation, thus encouraging the portability of Plans [39]. The Plan Engine uses Plugins to support different workflow languages (e.g. BPMN or BPEL) and runtimes (e.g. ODE¹⁵) [11].

Application Instantiation

Build plans are used to instantiate a previously deployed application. They can be invoked by selecting an appropriate plan via the *Self-Service Portal* (the so-called *Vinothek*) or by directly sending a SOAP message containing required input parameters [11]. A Plan uses the *Plan Portability API* to publish the Service Template of the application, which instantiates the Node Templates of its topology model, making it possible to retrieve the Node's properties (cf. Figure 3.11, (c)). With this information at hand, the workflow invokes (cf. Figure 3.11, (d)) the management operations of Node Templates and Relationship Templates necessary for instantiation. Corresponding Deployment Artifacts are retrieved from the Files store and installed on the target environment. The Plan continuously updates the states and properties of Nodes to keep the topology model aware of the actual state of the application instance. Finally, the build plan returns information about the outcome of the Plan's execution as well as the Url of the deployed application instance.

In summary, the extensible OSGi-based plugin architecture supports the introduction of new Artifact and Plan types, whereas the decoupling of concrete service endpoints from the definition of Plans leads to Plans which are portable between runtime environments [11]. Furthermore, the modular architecture allows scalability of each component as needed [11].

¹⁵<http://ode.apache.org/>

3.5.2 OpenTOSCA Ecosystem

In addition to the Runtime Environment discussed in this chapter, the OpenTOSCA Ecosystem provides tools to aid in the process of application development and delivery. The open-source visual modeling tool *Valesca*¹⁶ supports the user in modeling application topologies and related management plans by providing a palette of predefined components to graphically compose an application. As a follow-up, the modeling tool *Winery*¹⁷ has been proposed as an Eclipse project. The modeling process yields a self-contained application archive that can be downloaded from Winery and uploaded into the OpenTOSCA Container using the *Admin UI* to make it available to Customers. They make use of the self-service UI called *Vinothek*, which is used by Customers to choose and instantiate their application of choice.

¹⁶<http://www.cloudcycle.org/en/valesca/>

¹⁷<http://www.eclipse.org/proposals/soa.winery/>

Use Case Definition

The facility management for large commercial buildings consists of systems for Heating, Ventilation, and Air Conditioning (HVAC), lighting control, security, and alarming [20]. These systems are often provided by different vendors, leading to a heterogeneous system setup due to domain- and vendor-specific network protocols and control devices (cf. Section 1.1).

An Air Handling Unit (AHU) is used to condition and circulate air in an HVAC system. AHUs are commonly found in commercial solutions¹, thus making an AHU a good candidate to show how the deployment of its application logic can be automated and how its life-cycle management can be defined in a re-usable way. As depicted in Figure 4.1, sensors and actuators are applied to an AHU in order to remotely monitor and control them. This specific AHU mixes fresh air from the outside with the returning air from the room. A ventilator then blows the air through heating and cooling coils to heat or cool the air to a selected temperature point. The valves can be positioned to economically pre-heat or pre-cool the air by mixing the outside air with the return air.

Until now, building operators had to manually provision and configure the application logic of each gateway, which resulted in a time consuming and cost intensive task. The same applies for management tasks which needed to be ported to other building management setups, especially if gateway environments of different vendors were combined. Furthermore, expert knowledge about vendor-specific management operations was required to perform these tasks.

To show how we address the heterogeneity, this use case consists of two AHU instances which are deployed onto two distinct gateway frameworks. One instance is deployed onto a Niagara [41] gateway whereas the other is deployed onto a Sedona [5] gateway. The automation of deployment and life-cycle management procedures leads to an easier adaption and optimization of the control logic of AHUs. Thus, occupants will experience a better room climate and building operators can reduce the operational costs and energy consumption.

With the help of this use case we will show how IoT application deployment can be automated and how the re-usability of service management procedures can be accomplished.

¹<http://www.pacificcontrols.net/projects/ict-project.html>

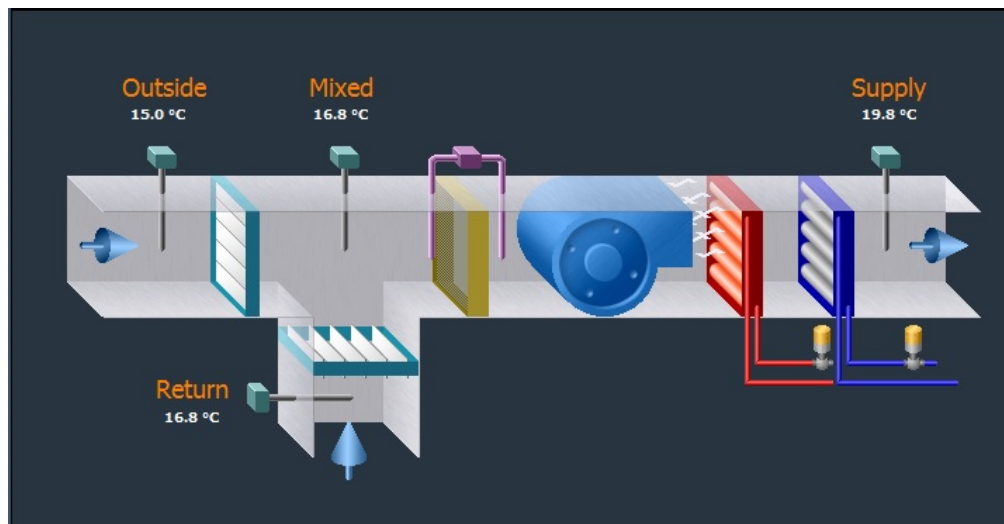


Figure 4.1: Air Handling Unit defined in the Niagara^{AX} [41] demo station.

CHAPTER 5

Design

The goal of this chapter is to describe the design of the prototypical implementation of the Air Handler use case introduced in the previous chapter. The *HVAC Application* depicted in Figure 5.1 consists of one AHU Controller instance deployed on a *Niagara Gateway* and a second AHU Controller instance deployed on a *Sedona Gateway*. This prototype is based on the OpenTOSCA Environment.

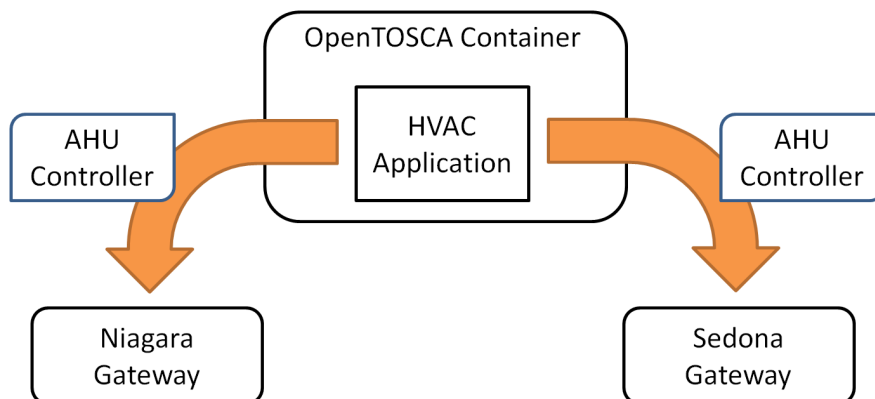


Figure 5.1: Overview of HVAC Application Prototype.

By creating this prototype we will show how IoT application deployment can be automated and how the re-usability of service management procedures can be accomplished. First, the requirements regarding the AHU instance management are defined. This includes the life-cycle states for the prototype's components together with the management operations. Then, the hierarchical node and relationship models defining the reusable components with their life-cycle interfaces are described. Additionally, the relation to the model introduced in our previous work will be discussed. Finally we present the architecture based on the OpenTOSCA Environment and the gateway-specific life-cycle implementations, which are based on interfaces we defined

for the control applications deployed on each gateway environment. This includes the interface definition of the workflows which perform life-cycle management procedures used to deploy, manage and terminate the HVAC application.

5.1 Definition of Life-cycle States & Management Operations

Throughout this Section we define the life-cycle states and management operations in a device-neutral way. The implementation of the HVAC application defined in Chapter 4 then needs to provide a vendor-specific implementation for the Niagara as well as for the Sedona environment. The *Instance Management* of the OpenTOSCA runtime is used to keep track of the state of each node instance. This is essential for executing life-cycle management procedures, as certain management operations only make sense if invoked in the correct starting state. For each class of components we defined suitable states to reflect the behaviour of the physical component it describes. The state changes are performed by life-cycle operations defined by the life-cycle interface of each component. After analyzing the use case we developed the state chart depicted in Figure 5.2 to describe the life-cycle of an Air Handling Unit Controller. It contains all instance states and management operations to deploy, configure, manage and undeploy an AHU control application. Before the AHU Controller is deployed to a gateway environment, it resides in state *Undeployed*. The “deploy” operation leads to a deployed and *Stopped* Controller which then can be configured using the “changeSetpoint” operation. After invoking the “start” operation, state *Running* is reached where the “Setpoint” of the Air Handler can further be adjusted. The “stop” operation stops the control logic from further steering the AHU, thus leading to the *Stopped* state. The “undeploy” operation deletes the application from the gateway environment.

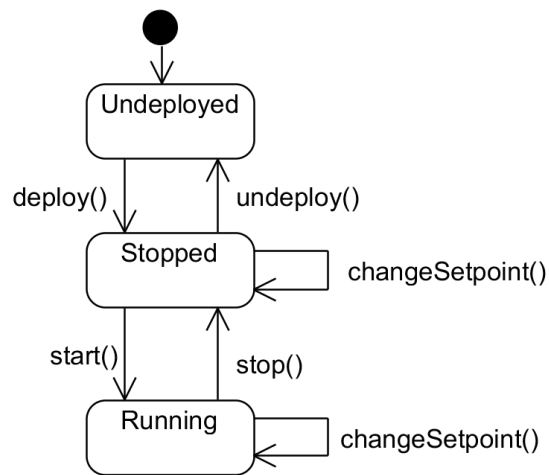


Figure 5.2: Controller’s Instance States and Air Handler’s Management Operations.

The life-cycle states and management operations of a Gateway are illustrated in Figure 5.3. As this work clearly focuses on the automation of application deployment, this state chart was kept simple. Of course, there would be several configuration and management operations like

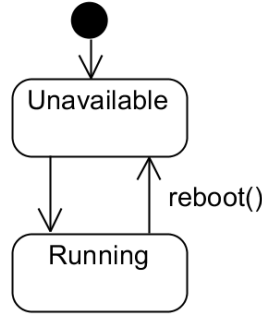


Figure 5.3: Gateway’s Instance States and Management Operations.

installing new control modules, updating existing modules or changing the gateway’s configuration, that would perfectly make sense in a productive environment. Thus, the sole management operation for Gateways implemented by the prototype is the “reboot” operation. After a short *Unavailable* phase, the Gateway changes back to state *Running*. These life-cycle states and operations of Gateways and AHUs will be used throughout the following section to define the IoT node and relationship models based on TOSCA.

5.2 Node and Relationship Models by TOSCA

The TOSCA node model consists of components typically appearing in Cloud applications. By adding *NodeTypes* representing common components occurring in IoT applications, TOSCA facilitates the modeling of building management applications like the AHU use case. The hierarchical node model we proposed in our previous work [9] is depicted in Figure 5.4. The node hierarchy consists of three tiers – *Base Node Types*, *Domain-specific Node Types* and *Concrete Node Types*. “The *Base Node Types* are directly derived from a generic TOSCA root node type. This puts them at the same level as other common cloud application nodes, including server, database and so on. The nodes at this level present the most fundamental concepts in IoT ap-

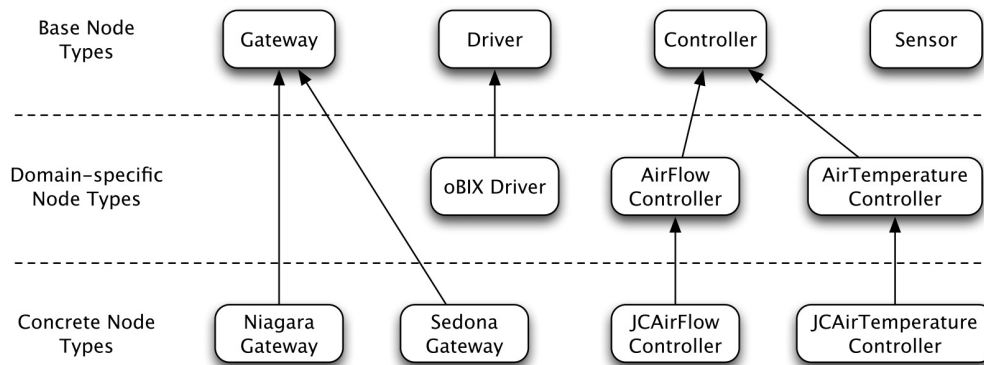


Figure 5.4: Node Types [9].

plications” [9]. We determined *Controller*, *Gateway*, *Driver* and *Sensor* to be the most generic types to classify the components occurring in our use case. “The *Domain-specific Node Types* are related to IoT applications in a certain industrial domain, which is building automation in our case. For example, oBIX is a protocol widely used in building automation projects” [9]. *Air-FlowController* and *AirTemperatureController* add AHU specific properties and interfaces to the definition of the generic Controller Node Type. “The *Concrete Node Types* define the node types to be used in a specific application, with information about specific hardware and software vendors, models and versions” [9]. *NiagaraGateway* and *SedonaGateway* are the concrete Node Types adding vendor-specific properties and interfaces to the generic Gateway node type.

Based on this hierarchical node model we propose a simplified deployment view (cf. Figure 5.5) describing the AHU application prototype throughout our previous work. As one can

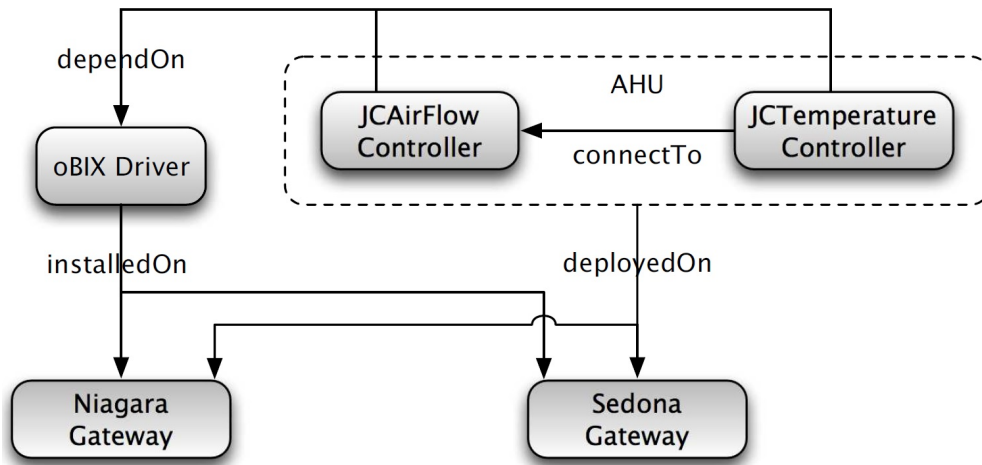


Figure 5.5: Air Handling Unit use case [9].

see, a *JCTemperatureController* together with a *JCAirFlowController* connected to it, build up a logical AHU component. Both Controllers require an *oBIX Driver* to be installed on each Gateway they are deployed on. This approach allows us to model the components of the control logic running on a specific gateway using TOSCA types.

The advantage is a fine-grained topology model of the application, but there is also a downside. Supporting application modeling at this granularity level leads to a significantly increasing development effort for Concrete Node Type implementations induced by gateway framework specifics. After analyzing the architectures of the gateway frameworks occurring in the use case, we decided to deviate from the previously proposed hierarchical node model.

The Topology of the prototype used in this work describes the application logic for each AHU as a whole, by using the individual vendor-specific TOSCA Nodes *SedonaAHUController* and *NiagaraAHUController*. The application logics steering the physical AHUs are provisioned as complete application images to the gateway. Although the adaption of the hierarchical node model changes the level of granularity, this prototype shows the feasibility of applying TOSCA to the IoT domain by facilitating the automation of application deployment and management and making life-cycle management portable even if gateways of different vendors are used.

5.2.1 Hierarchical Node Model

The hierarchical node model used in this work is illustrated in Figure 5.6. We determined *Gateway* and *Controller* as the *Base Node Types* for modeling the building automation application of this prototype. These abstract node types are derived from the *RootNodeType*. The *Gateway* node type describes generic IoT gateways and defines the properties `Host`, `Port`, `Username` and `Password` needed to access them.

SedonaGateway and *NiagaraGateway* are the device-specific Gateway node types, where *NiagaraGateway* defines additional properties related to the Niagara Proxy component, which will be discussed in Section 5.3.4.

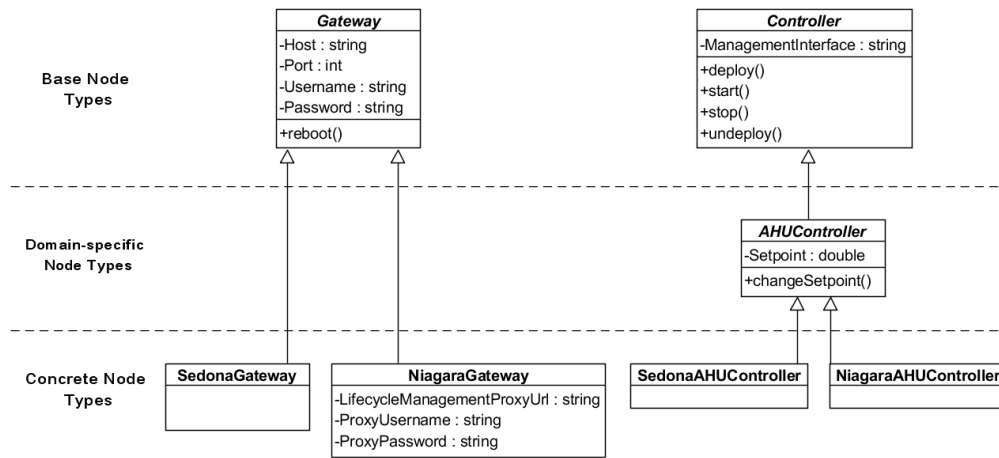


Figure 5.6: Node Types implemented for the use case.

The *Controller* node type defines a `ManagementInterface` referencing a specific component to interface with the Controller’s application logic deployed on a gateway (cf. Section 5.3). The previously defined life-cycle management operations for AHU control applications (cf. Figure 5.2) are incorporated in the node model by requiring any kind of Controller to support at least the “deploy”, “start”, “stop” and “undeploy” operations. The domain-specific abstract *AHUController* node type is derived from *Controller* and adds the `Setpoint` property allowing to configure the desired setpoint temperature of an Air Handler in a device-independent manner. Adding the “changeSetpoint” operation to the four basic management operations derived from *Controller* completes the state transitions of the AHU life-cycle. *SedonAHUController* and *NiagaraAHUController* represent the vendor-specific *AHUController* node types which can be consecutively be instantiated to implement the IoT application (cf. Section 6.3). The implementation of the node types is discussed in detail in Section 6.2.

Gateway and *Controller* are *Base Node Types* using the *Gateway Instance States* and *Controller Instance States* illustrated in Figure 5.7 to keep track of the life-cycle states of the components they represent.

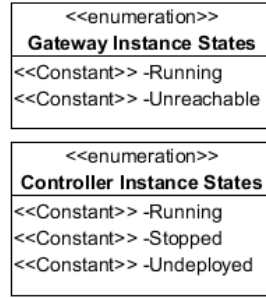


Figure 5.7: Gateway and Controller Instance States.

5.2.2 Life-cycle Interfaces

Each *Concrete Node Type* has at least one life-cycle interface defining management operations with vendor-specific parameters. The abstract node types of the node model (cf. Figure 5.6) define the operations of the interfaces which then get refined with the vendor-specific parameters of the *Concrete Node Types*. Figure 5.8 depicts the life-cycle interfaces to be implemented by the *Concrete Node Types*. The *NiagaraGateway* node type defines the `NiagaraGateway` interface and the *NiagaraAHUController* node type defines the `NiagaraController` interface as well as the `NiagaraAHUController` interface. The *SedonaGateway* node type defines the

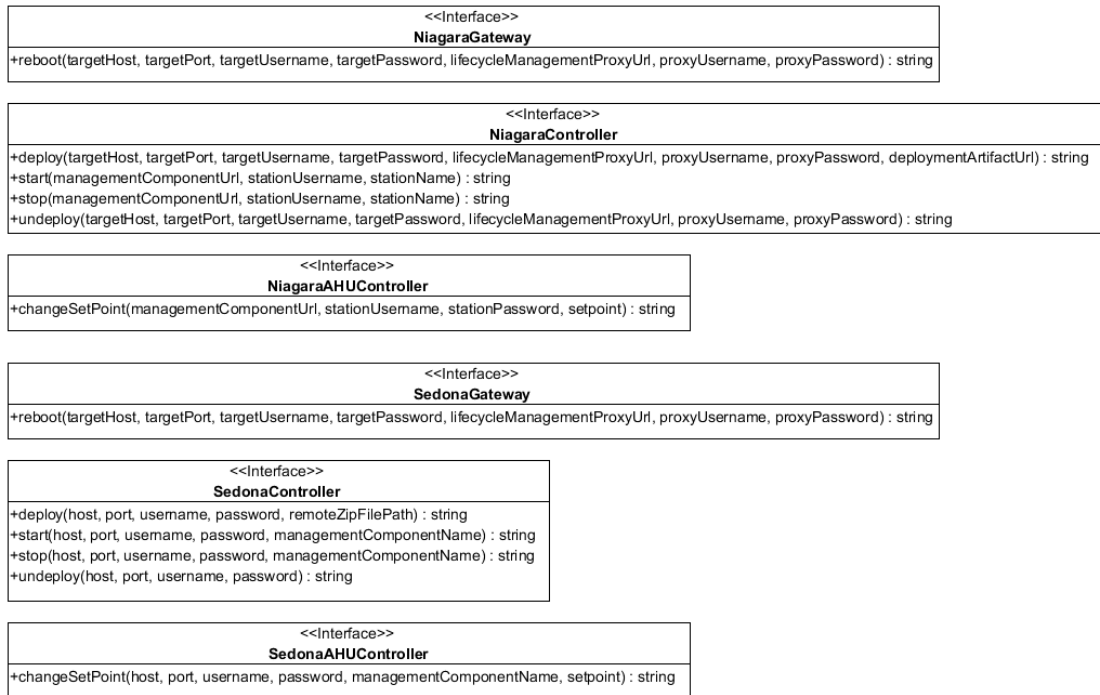


Figure 5.8: Concrete Node Types' Life-cycle Interfaces.

SedonaGateway interface and the *SedonaAHUController* node type defines the *SedonaController* interface as well as the *SedonaAHUController* interface. These life-cycle interfaces are defined by the Node Type definitions of Section 6.2.1. The interfaces of the concrete Node Types are then implemented by the *Node Type Implementations* (cf. Section 6.2.2) to facilitate the invocation of the management operations by high-level management procedures (cf. Section 5.4).

5.2.3 Hierarchical Relationship Model

To model the structure of an application, TOSCA defines *Relationship Types* defining dependencies between application components specified through *Requirement* and *Capability* definitions. We defined the abstract *DeployedOn* relationship type which is derived from *RootRelationshipType* and specifies a semantic relation of a generic *Controller* node type deployed on a generic *Gateway* node type (cf. Figure 5.9). This *Base Relationship Type* defines this relation by specifying a *ControllerRequirement* requirement type as source element and a corresponding *ControllerCapability* capability type as target element. The *Concrete Relationship Types* *NiagaraControllerDeployedOnNiagaraGateway* and *SedonaControllerDeployedOnSedonaGateway* define the vendor-specific *DeployedOn* relationship using the vendor-specific requirement and capability definitions.

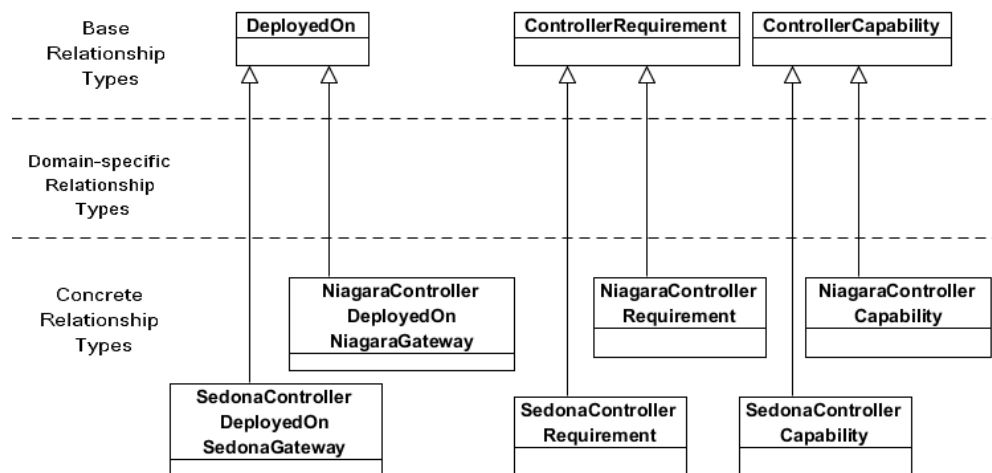


Figure 5.9: Relationship Types implemented for the use case.

To make the Relationship Types applicable to the node types defined in the previous Section, the node types need to define their Requirements and Capabilities. Thus, the Niagara Gateway node type specifies a CapabilityDefinition of type *NiagaraControllerCapability*, meaning that it can host a Niagara Controller application. The *NiagaraControllerRequirement* defined by the Niagara Controller node type can then be satisfied by interconnecting it to the Gateway Node Type through the *NiagaraControllerDeployedOnNiagaraGateway* relationship type. Following the same schema, the Sedona Gateway node type specifies a CapabilityDefinition of type *SedonaControllerCapability*, meaning

that it can host a Sedona Controller application. The `SedonaControllerRequirement` defined by the Sedona Controller node type can then be satisfied by interconnecting it to the Sedona Node Type through the `SedonaControllerDeployedOnSedonaGateway` relationship type. The implementation of the Relationship Types is discussed in Section 6.2.3.

5.3 Architecture

The architecture of the prototype consists of three major components – the OpenTOSCA environment, the Sedona environment and the Niagara environment. The architecture of the OpenTOSCA and Sedona environment is depicted in Figure 5.10. The Niagara environment is illustrated in Figure 5.11. Only the most important components are part of the diagrams to give an overview of the prototypes architecture, including the interfaces between the components.

5.3.1 OpenTOSCA Environment

The architecture of the OpenTOSCA Container Runtime 1.1 was already discussed in detail in Section 3.5. The Container is needed to process the CSAR file containing the artifacts which define the prototypical application.

It deploys the Implementation Artifacts (IAs) (*SedonaIAService.war* and *SedonaIAService.war* WAR files) on the Implementation Artifact Runtime (e.g. Apache Tomcat 7 Web server) and the BPEL workflows defining the *Management Plans* on the Plan Runtime (e.g. WSO2 Business Process Server (BPS)). The *Management Plans* rely on the BPEL4Rest extension to access the REST API of the *Instance Management* to retrieve and modify the properties and states of Node instances. They use the *Service Invoker* to invoke management operations defined by the life-cycle interfaces of node templates. The Service Invoker delegates the invocations to the Implementation Artifact service that implements the corresponding interface. The implementation of the IAs is discussed in Section 6.5 and the interfaces for the Management Plans are defined in Section 5.4. The implementation of these BPEL workflows is explained in Section 6.6. The Web applications of the Admin UI (*admin.war*), Self-Service UI (*vinothek.war*) and the Modeling tool Winery (*winery.war* and *winery-topologymodeller.war*) are part of the OpenTOSCA Ecosystem and are also running on the Web server.

5.3.2 Interfacing with Gateways & Control Applications

The *Implementation Artifacts* implement the life-cycle management operations of the Gateways and AHU Controllers by incorporating deep management knowledge about the gateway environment (cf. Section 3.3) in use. Specifics of the available communication protocols as well as architectural constraints are acknowledged by each implementation. The Sedona gateways for example allow a more direct implementation than the Niagara gateways as they offer low-level access to the Sedona environment via the open Sox protocol. The Niagara environment provides higher-level management operations, but due to the proprietary protocols (Fox, Niagarad) they can't be accessed directly. Sophisticated vendor-specific solutions are worth the development effort, as the Implementation Artifacts can be reused to communicate with any amount of gate-

ways of the same type. Furthermore, the life-cycle management operations are portable between TOSCA runtimes which understand the WAR Implementation Artifact type.

Controllers developed to control Air Handling Units are implemented by gateway environment specific control applications. To change the state and the configuration of an AHU Controller, the properties of the control application have to be changed. To facilitate the automation of the life-cycle management of the control application, each Controller requires a corresponding *Management Interface* definition as part of the control application. Thus, the life-cycle management does not need to understand the data points and control components the application is built of.

5.3.3 Sedona Environment

The architecture of the Sedona Environment is depicted in Figure 5.10. The *SedonaIAService* uses the *Sox Client* to connect to the Sedona Gateway to deploy and configure the Sedona AHU Controller Application. The open source Sox protocol allows live-programming of gateways running the *Sox Service*. The *Sox Service* exposes the *AHUMgmt* interface facilitating the configuration of the actual control logic defined by the *AHUImp* component. The implementation of the AHUMgmt component, as part of the Sedona AHU Controller application, is discussed in Section 6.4.1. The *SedonaIAService* uses the Sox Client API for application deployment, which is explained in Section 6.5.

5.3.4 Niagara Environment

The *NiagaraIAService* uses the *Obix-Toolkit* to deploy and configure Niagara AHU Controllers via the Obix protocol. As indicated by Figure 5.11, deployment and configuration are separated. Deployment and undeployment capabilities are implemented by the *LifecycleManagementProxy* component installed on the *Niagara Supervisor Station* which is hosted on a *Supervisor Platform*¹. The *Obix Network Driver* exposes the management operations of the *LifecycleManagementProxy*. The interface definition (cf. Table 5.1) defines the Obix actions and the required parameters. Its implementation is discussed in Section 6.5.3. The Proxy uses the *Station Man-*

Obix actions	Parameters
reboot/	host, port, username, password
deploy/	host, port, username, password, stationName
undeploy/	host, port, username, password

Table 5.1: LifecycleManagementProxy Obix Interface.

ager to connect to the *Target Platform* running on the *Niagara Gateway* through the *Niagarad* protocol. A Niagara Station (which is represented as a `config.bog` database file) containing the control logic of the AHU Controller is deployed or terminated by the *StationManager*.

¹ A Supervisor is installed on a PC rather than a gateway, providing more computational resources for complex control logics. A supervisor typically consolidates and visualizes data points of several gateways. - http://www.tridium.com/galleries/datasheet_pdf/2011-T-AXS.FINAL.pdf

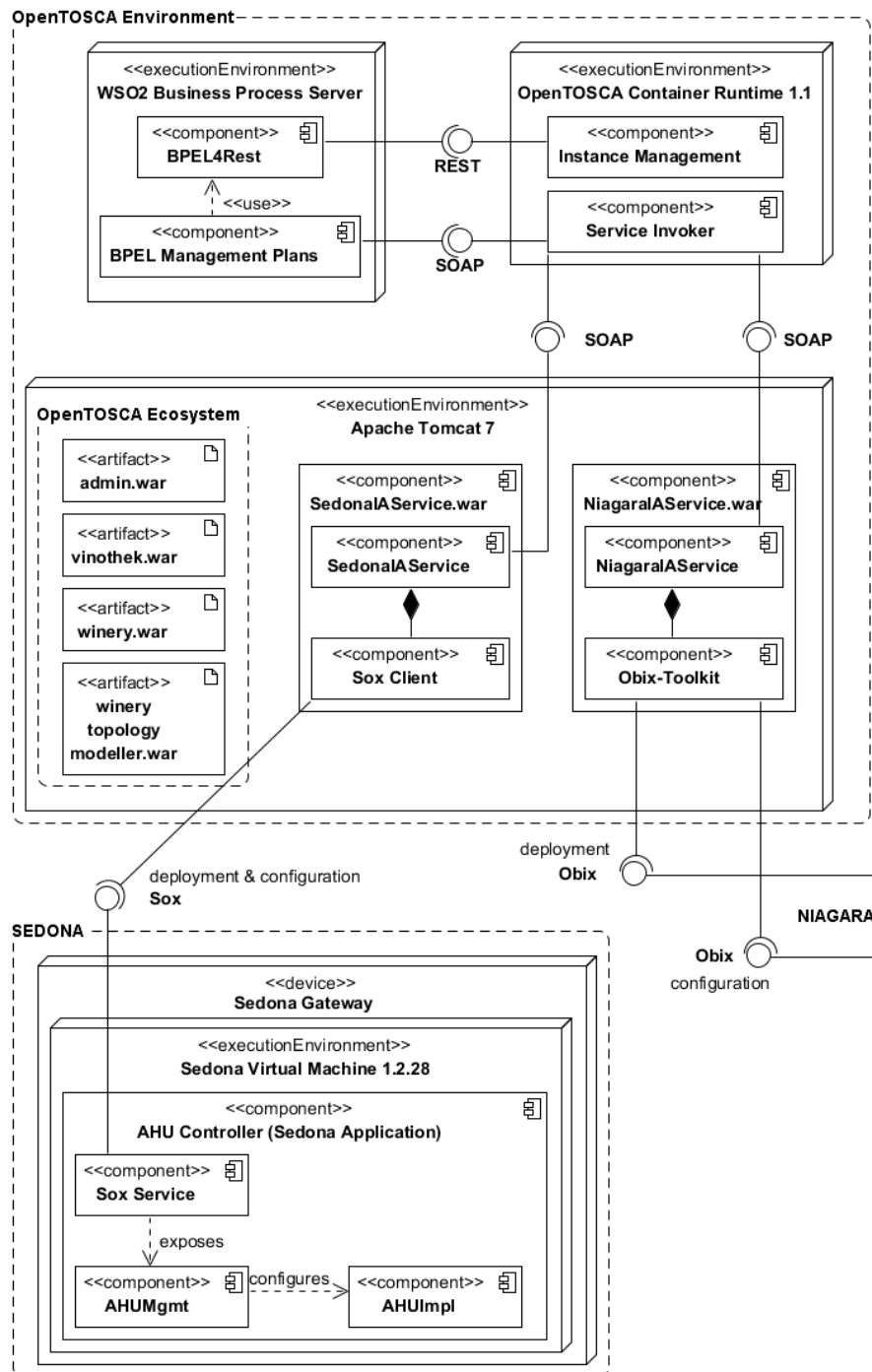


Figure 5.10: OpenTOSCA Environment and Sedona Architecture.

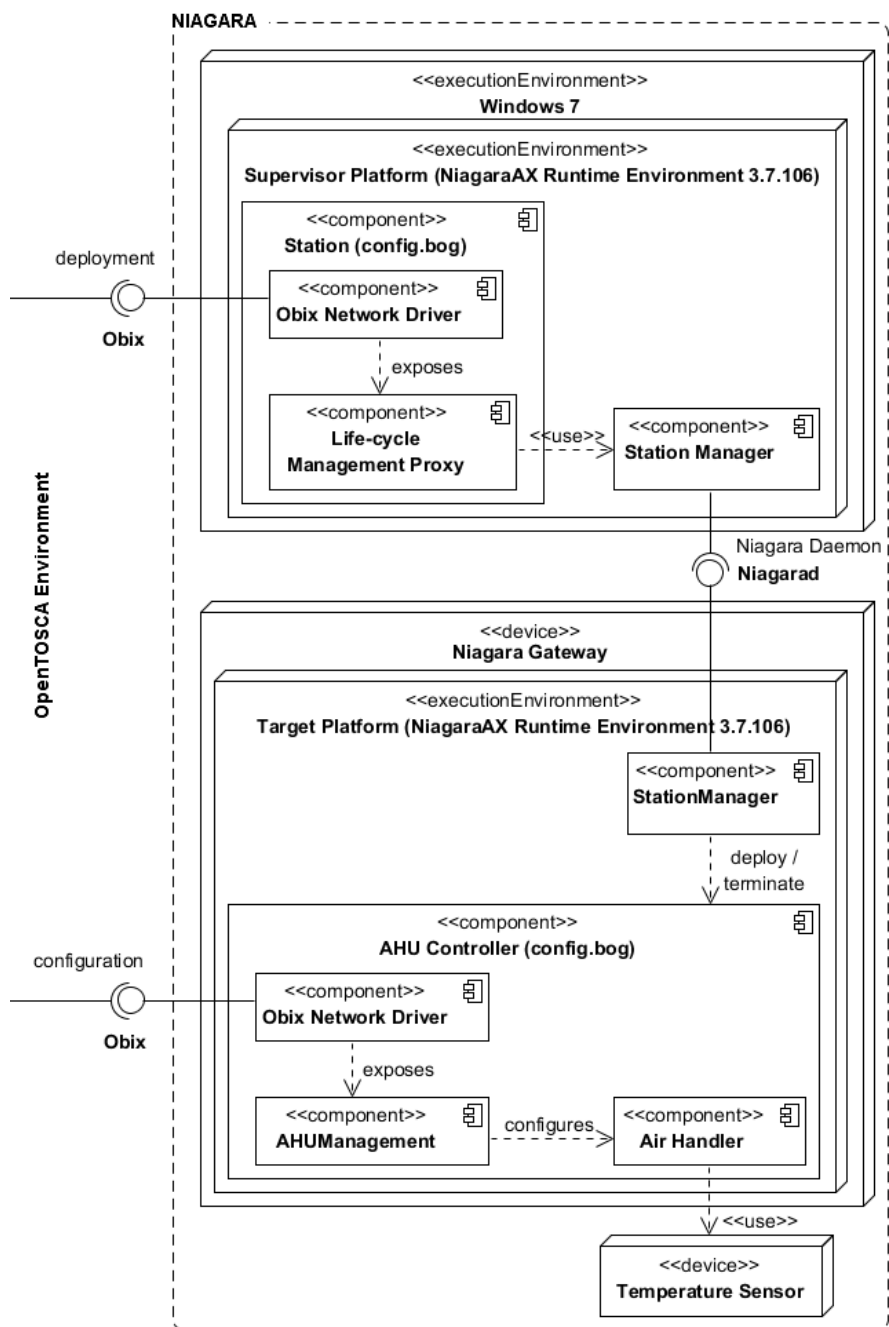


Figure 5.11: Niagara Architecture.

Obix actions	Parameters
startController/	none
stopController/	none
changeSetpoint/	setpoint : float

Table 5.2: AHUManagement Obix Interface.

The *Obix Network Driver* of the *AHU Controller* exposes the *AHUManagement* interface (cf. Table 5.2) to configure the actual control logic defined by the *Air Handler* component. The implementation of the *AHUManagement* component is discussed in Section 6.4.2.

5.4 Life-cycle Management Procedures

Plans are workflows implementing high-level management procedures by orchestrating management operations defined by life-cycle interfaces (cf. Section 5.2.2) of node types. Throughout this section we define the BPEL workflow interfaces for the life-cycle management procedures that will be used in the *ServiceTemplate* definition to implement the prototype application (cf. Section 6.3). A *BuildPlan* is used to instantiate the *ServiceTemplate* of the application by deploying and configuring the AHU Controller control logic on the gateways. A *TerminationPlan* deletes the control logic from the gateways and terminates the service instance. Several plans of type *ManagementPlan* perform life-cycle management operations on specific *Node Instances*. The implementation of the following Plan definitions and their implications on the architectural components will be discussed in detail in Section 6.6.

5.4.1 Application Deployment

The *BuildPlan* instantiates the topology model of the application by deploying and configuring the Niagara and Sedona AHU Controller gateway control logic on the corresponding gateways (cf. Figure 5.12).

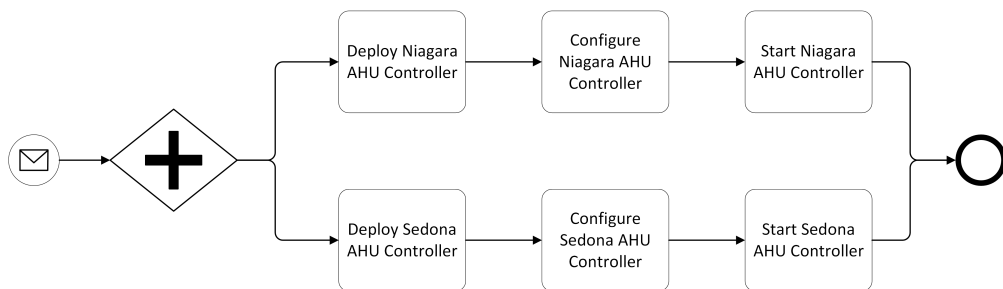


Figure 5.12: IoT Application Deployment Workflow.

The interface of the *BuildPlan* is part of the *Plans* definition of the *ServiceTemplate* and is depicted in Listing 5.1. This *Plan* definition indicates that it is of type *BuildPlan* (*planType*) and uses the WS-BPEL 2.0 workflow language (*planLanguage*). The *BuildPlan* receives the

name of the CSAR file as well as the Url of the Container API. The Plan returns the Uri of the newly created service instance as well as the Uri of each Node Instance. Furthermore, the result of each management operation is returned. The Node Instance Uris are consecutively used by management plans to manipulate the related Node Instances directly. Finally the *PlanModelReference* points to the location of the BPEL workflow implementation.

Listing 5.1: HVAC BuildPlan Definition.

```

1 <Plan id="DeployHVAC"
2   name="Deploy HVAC Instance"
3   planType="http://.../tosca/.../PlanTypes/BuildPlan"
4   planLanguage="http://.../wsbpel/2.0/process/executable">
5
6   <InputParameters>
7     <InputParameter name="csarName" type="string" />
8     <InputParameter name="containerApi" type="string" />
9   </InputParameters>
10
11   <OutputParameters>
12     <OutputParameter name="serviceInstanceUri" type="string" />
13
14     <OutputParameter name="sedonaGatewayNodeInstanceUri" />
15     <OutputParameter name="sedonaAHUControllerNodeInstanceUri" />
16     <OutputParameter name="niagaraGatewayNodeInstanceUri" />
17     <OutputParameter name="niagaraAHUControllerNodeInstanceUri"/>
18
19     <OutputParameter name="sedonaDeployResult" type="string" />
20     <OutputParameter name="sedonaConfigureResult" type="string"/>
21     <OutputParameter name="sedonaStartResult" type="string" />
22     <OutputParameter name="niagaraDeployResult" type="string" />
23     <OutputParameter name="niagaraConfigureResult" />
24     <OutputParameter name="niagaraStartResult" type="string" />
25   </OutputParameters>
26
27   <PlanModelReference
28     reference="Plans/HVACBuildPlan.zip" />
29 </Plan>

```

5.4.2 Application Termination

The *TerminationPlan* decommissions the Niagara and Sedona AHU Controller gateway control logic from the corresponding gateways and removes the Service Instance from the OpenTOSCA Container's Instance Management. The Plan's definition (cf. Listing 5.2) indicates that it is of type *TerminationPlan* (*planType*) and uses the WS-BPEL 2.0 workflow language (*planLanguage*). The plan's interface defines the name of the CSAR file, the Url of the Container

API and the Service Instance Uri as `InputParameters` and the result of the stop and undeployment operation of each controller type as `OutputParameters`. Finally, the *PlanModelReference* points to the location of the BPEL workflow implementation.

Listing 5.2: HVAC TerminationPlan Definition.

```
1 <Plan id="UndeployHVAC"
2   name="HVAC Termination"
3   planType="http://.../tosca/.../PlanTypes/TerminationPlan"
4   planLanguage="http://.../wsbpel/2.0/process/executable">
5
6   <InputParameters>
7     <InputParameter name="containerApi" type="string" />
8     <InputParameter name="csarName" type="string" />
9     <InputParameter name="serviceInstanceUri" type="string" />
10  </InputParameters>
11
12  <OutputParameters>
13    <OutputParameter name="sedonaStopResult" type="string" />
14    <OutputParameter name="sedonaUndeployResult" />
15
16    <OutputParameter name="niagaraStopResult" type="string" />
17    <OutputParameter name="niagaraUndeployResult" />
18  </OutputParameters>
19
20  <PlanModelReference
21    reference="Plans/HVACTerminationPlan.zip" />
22 </Plan>
```

5.4.3 Application Management

The BPEL workflows to configure, start and stop the air handlers controlled by AHU Controllers are defined in a way such that the invoking party doesn't need to know the concrete type (e.g. Niagara or Sedona) of the node instance the operation is invoked on. The plans query the actual concrete node type via the container's instance management to decide which implementation needs to be used.

Air Handler Configuration

A Plan of type `ManagementPlan` to change the setpoint temperature of the AHU Controller is defined in Listing 5.3. The `controllerNodeInstanceUri` points to an AHU Controller node instance whereas the `gatewayNodeInstanceUri` points to the Gateway it is deployed on. The `setpoint` `InputParameter` specifies the desired target temperature of the AHU.

Listing 5.3: HVAC ChangeSetpoint Plan Definition.

```

1 <Plan id="ChangeSetPoint"
2   name="Change SetPoint of AHU Controller"
3   planType="http://.../tosca/.../PlanTypes/ManagementPlan"
4   planLanguage="http://.../wsbpel/2.0/process/executable">
5
6   <InputParameters>
7     <InputParameter name="csarName" type="string" />
8     <InputParameter name="controllerNodeInstanceUri" t="string"/>
9     <InputParameter name="gatewayNodeInstanceUri" t="string" />
10    <InputParameter name="setpoint" type="float" />
11  </InputParameters>
12
13  <OutputParameters>
14    <OutputParameter name="result" type="string" />
15    <OutputParameter name="processedNodeType" />
16  </OutputParameters>
17
18  <PlanModelReference
19    reference="Plans/ChangeSetPointPlan.zip" />
20 </Plan>

```

Start & Stop Controller

The operations to manage the life-cycle of a generic Controller are implemented by the *StartController* and *StopController* management plans. The StartController plan is depicted in Listing 5.4. As the StopController plan differs only by the method names (substitute any “start” by “stop”) it is not shown here. The controllerNodeInstanceUri points to an AHU Controller node instance whereas the gatewayNodeInstanceUri points to the Gateway it is deployed on. As these plans can be invoked on any node instance of a node type that is derived from the generic Controller node type, it can be used to start and stop the AHU Controller instances. The StopController Plan would change the state to “Stopped”.

Listing 5.4: HVAC StartController Plan Definition.

```

1 <Plan id="StartController"
2   name="Start generic Controller"
3   planType="http://.../tosca/.../PlanTypes/ManagementPlan"
4   planLanguage="http://.../wsbpel/2.0/process/executable">
5   <InputParameters>
6     <InputParameter name="csarName" type="string" />
7     <InputParameter name="controllerNodeInstanceUri" />
8     <InputParameter name="gatewayNodeInstanceUri" />
9   </InputParameters>

```

```
10
11   <OutputParameters>
12     <OutputParameter name="result" type="string" />
13     <OutputParameter name="processedNodeType" />
14   </OutputParameters>
15
16   <PlanModelReference
17     reference="Plans/StartControllerPlan.zip" />
18 </Plan>
```

Implementation

Now the implementation of the prototype designed in the previous Chapter will be discussed. First, the structure of the Cloud Service Archive will be explained. Second, we show how the reusable node and relationship types are implemented following the definition of the hierarchical node and relationship models (cf. Section 5.2). Then we describe how the prototype is implemented by composing these components to create the Topology Template of the HVAC application. Afterwards, the implementation of the interfaces for the device-specific AHU Controllers to be deployed on the gateways are discussed. Then, the implementation of the Implementation Artifacts is discussed in detail by explaining their class structure together with important code snippets. This Chapter concludes with a description of the BPEL workflows that implement the Plans defined in Section 5.4. This includes a detailed discussion of the interaction between the Plans and other components of the architecture (cf. Section 5.3) during the execution of the life-cycle management procedures.

6.1 Structure of the Cloud Service Archive (CSAR)

The HVAC application is packaged in a self-contained way as a Cloud Service Archive (CSAR). This `HVAC.csar` file can be processed by the OpenTOSCA container environment. The directory structure of the CSAR is depicted in Figure 6.1.

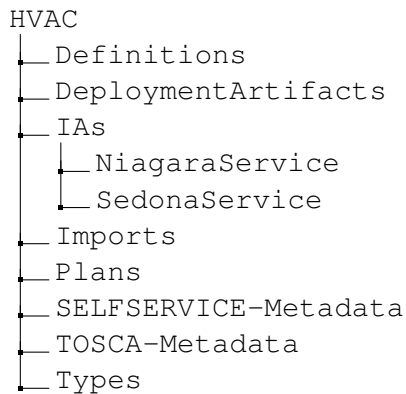


Figure 6.1: CSAR Directory Structure.

Next, each folder’s content will be describe briefly, starting with the most important one.

- The `Definitions` folder, required by the TOSCA specification, contains the definitions of the reusable TOSCA types, categorized in base types, specific types as well as concrete Niagara and Sedona types (cf. Section 5.2 for Design and Section 6.2 for Implementation). Building upon these type definitions, the *Service Template* (cf. Section 6.3) of the HVAC application is defined within the `HVAC-Definitions.xml` file.
- The `TOSCA-Metadata` folder contains the obligatory `TOSCA.meta` file. This file contains metadata about the CSAR as well as an `Entry-Definitions` element advising the TOSCA environment to start with the `HVAC-Definitions.xml` file when processing the CSAR.
- The `DeploymentArtifacts` folder contains the Sedona and Niagara archive artifacts containing all files (e.g. application images) to successfully deploy each gateway application (cf. Section 6.4).
- The `IAs` folder contains the Implementation Artifacts (cf. Section 5.3 for Design and Section 6.5 for Implementation) which implement the vendor-specific Node Types as Web services – `Niagara-Service.war` and `Sedona-Service.war`).
- The `Imports` folder contains all WSDL Web service definitions of Plans (`PlanArtifact.wsdl` files) which are imported by `HVAC-Definitions.xml` as well as the WSDL definitions of the Implementation Artifacts (`SedonaIAService.wsdl` and `Sedona-IAService.wsdl`) which are imported by the Sedona and Niagara node type definitions.
- The `Plans` folder contains the BPEL workflows as “zip” files, which are referenced from the *Service Template* in the `HVAC-Definitions.xml`. The Plans are defined in Section 5.4 and implemented in Section 6.6.
- The `Types` folder contains the XML schema definitions (`xsd` files) to specify a schema for the properties of each node type.

- The SELFSERVICE-Metadata folder contains information used by the “Vinothek” self-service UI. The data.xml is used to describe the HVAC application and to define the “Deploy” and “Terminate” life-cycle management procedures referencing a SOAP message (e.g. plan.input.deploy.xml to automatically invoke the corresponding BPEL workflows.

6.2 Node & Relationship Type Definitions

This Section provides detailed information about the Node and Relationship Types implementing the hierarchical models defined in Section 5.2. Furthermore, the *Node Type Implementations* used to specify the Implementation Artifacts which realize the life-cycle interfaces (cf. Section 5.2.2) of *Node Types* are discussed.

6.2.1 Node Type Definitions

Now we implement the NodeTypes described in Figure 5.6. The abstract Controller Node-Type (cf. (1)¹ of Listing 6.1) is derived from RootNodeType (3) and defines the Controller specific properties (4-5). The RequirementDefinition (7-9) defines the necessity of a “gateway” to deploy the Controller onto. (11-15) defines the possible Controller Instance States introduced in Figure 5.7. The life-cycle interface `http://.../lifecycle/controller/`² (17-22) defines the management operations required by a Controller.

Listing 6.1: Controller NodeType.

```

1 <NodeType name="Controller" abstract="yes">
2   <documentation>Generic IoT Controller</documentation>
3   <DerivedFrom typeRef="tns:RootNodeType" />
4   <PropertiesDefinition
5     element="baseTypes:ControllerProperties" />
6   <RequirementDefinitions>
7     <RequirementDefinition
8       name="gateway" lowerBound="1" upperBound="1"
9       requirementType="tns:ControllerRequirement" />
10  </RequirementDefinitions>
11  <InstanceStates>
12    <InstanceState state="www.example.com/undeployed" />
13    <InstanceState state="www.example.com/running" />
14    <InstanceState state="www.example.com/stopped" />
15  </InstanceStates>
16  <Interfaces>
17    <Interface name="http://.../lifecycle/controller/">

```

¹The numbers refer to the code lines of the listing.

²Full interface name: `http://docs.oasis-open.org/tosca/ns/2011/12/interfaces/lifecycle/controller/`

```

18     <Operation name="deploy" />
19     <Operation name="start" />
20     <Operation name="stop" />
21     <Operation name="undeploy" />
22 </Interface>
23 </Interfaces>
24 </NodeType>

```

The abstract *AHUController* NodeType (cf. (1) of Listing 6.2) is derived from *Controller* (3) and defines the *AHUController* specific properties (4-5). The `http://.../interfaces/lifecycle/ahu/` life-cycle interface (7-9), defining the “changeSetpoint” operation to configure an AHU, is added to the Controller’s management interfaces.

Listing 6.2: AHUController NodeType.

```

1 <NodeType name="AHUController" abstract="yes">
2   <documentation>Generic AHU Controller</documentation>
3   <DerivedFrom typeRef="baseTypes:Controller" />
4   <PropertiesDefinition
5     element="tns:AHUControllerProperties" />
6   <Interfaces>
7     <Interface name="http://.../interfaces/lifecycle/ahu/">
8       <Operation name="changeSetpoint" />
9     </Interface>
10 </NodeType>

```

The gateway environment specific *SedonaAHUController* NodeType (cf. (1) of Listing 6.3) is derived from *AHUController* (4) and defines the *SedonaAHUController* specific properties (5-6). The *RequirementDefinition* (8-10) refines the definition of the Controller node type by requiring a gateway that satisfies the “SedonaControllerRequirement”.

Listing 6.3: SedonaAHUController NodeType.

```

1 <NodeType name="SedonaAHUController">
2   <documentation>Sedona implementation of an
3     Air Handling Unit Controller</documentation>
4   <DerivedFrom typeRef="specificTypes:AHUController" />
5   <PropertiesDefinition
6     element="SedonaAHUControllerProperties" />
7   <RequirementDefinitions>
8     <RequirementDefinition
9       requirementType="SedonaControllerRequirement"
10      name="gateway" upperBound="1" lowerBound="1" />
11   </RequirementDefinitions>
12 </NodeType>

```

The gateway environment specific *NiagaraAHUController* NodeType (cf. (1) of Listing 6.4) is derived from *AHUController* (4) and defines the *NiagaraAHUController* specific properties

(5-6). The RequirementDefinition (8-10) refines the definition of the Controller node type requiring a gateway satisfying the “NiagaraControllerRequirement”.

Listing 6.4: NiagaraAHUController NodeType.

```

1 <NodeType name="NiagaraAHUController">
2   <documentation>Niagara implementation of an
3     Air Handling Unit Controller</documentation>
4   <DerivedFrom typeRef="specificTypes:AHUController" />
5   <PropertiesDefinition
6     element="NiagaraAHUControllerProperties" />
7   <RequirementDefinitions>
8     <RequirementDefinition
9       requirementType="NiagaraControllerRequirement"
10      name="gateway" upperBound="1" lowerBound="1" />
11   </RequirementDefinitions>
12 </NodeType>

```

The abstract Gateway NodeType (cf. (1) of Listing 6.5) is derived from RootNodeType (3) and defines the Gateway specific properties (4-5). The CapabilityDefinition (7-9) defines the capability of deploying a Controller on the Gateway. (11-14) defines the possible Gateway Instance States introduced in Figure 5.7. The life-cycle interface `http://.../interfaces/lifecycle/gateway/` (16-18) defines the management operations required by a Gateway.

Listing 6.5: Gateway NodeType.

```

1 <NodeType name="Gateway" abstract="yes">
2   <documentation>Generic IoT Gateway</documentation>
3   <DerivedFrom typeRef="tns:RootNodeType" />
4   <PropertiesDefinition
5     element="baseTypes:GatewayProperties" />
6   <CapabilityDefinitions>
7     <CapabilityDefinition
8       name="controller" lowerBound="0" upperBound="1"
9       capabilityType="tns:ControllerCapability" />
10  </CapabilityDefinitions>
11  <InstanceStates>
12    <InstanceState state="www.example.com/unreachable" />
13    <InstanceState state="www.example.com/running" />
14  </InstanceStates>
15  <Interfaces>
16    <Interface name="http://.../interfaces/lifecycle/gateway/">
17      <Operation name="reboot" />
18    </Interface>
19  </Interfaces>
20 </NodeType>

```

The gateway environment specific *SedonaGateway* NodeType (cf. (1) of Listing 6.6) is derived from *Gateway* (4) and defines the *SedonaGateway* specific properties (5-6). The *CapabilityDefinition* (8-10) restricts the “controller” capability to *Sedona* Controllers.

Listing 6.6: Sedona Gateway NodeType.

```

1 <NodeType name="SedonaGateway">
2   <documentation>Sedona implementation of an
3     Air Handling Unit Controller</documentation>
4   <DerivedFrom typeRef="base:Gateway" />
5   <PropertiesDefinition
6     element="sedona:SedonaGatewayProperties" />
7   <CapabilityDefinitions>
8     <CapabilityDefinition
9       capabilityType="tns:SedonaControllerCapability"
10      name="controller" upperBound="1" lowerBound="0" />
11   </CapabilityDefinitions>
12 </NodeType>

```

The gateway environment specific *NiagaraGateway* NodeType (cf. (1) of Listing 6.7) is derived from *Gateway* (4) and defines the *NiagaraGateway* specific properties (5-6). The *CapabilityDefinition* (8-10) restricts the “controller” capability to *Niagara* Controllers.

Listing 6.7: Niagara Gateway NodeType.

```

1 <NodeType name="NiagaraGateway">
2   <documentation>Niagara implementation of an
3     Air Handling Unit Controller</documentation>
4   <DerivedFrom typeRef="base:Gateway" />
5   <PropertiesDefinition
6     element="niagara:NiagaraGatewayProperties" />
7   <CapabilityDefinitions>
8     <CapabilityDefinition
9       capabilityType="tns:NiagaraControllerCapability"
10      name="controller" upperBound="1" lowerBound="0" />
11   </CapabilityDefinitions>
12 </NodeType>

```

6.2.2 Node Type Implementations

We decided to capture all *Sedona* related management knowledge in the “*SedonaControl*” Implementation Artifact and all *Niagara* related management knowledge in the “*NiagaraControl*” Implementation Artifact.

The *NodeTypeImplementation* of the *SedonaGateway* node type (cf. (1-2) of Listing 6.8) defines the “*SedonaControl*” interface (5) and references the *ArtifactTemplate* (7) of type “*WAR*” (6) which is discussed next.

Listing 6.8: Sedona Gateway NodeTypeImplementation.

```

1 <NodeTypeImplementation name="SedonaGatewayTypeImpl"
2   nodeType="tns:SedonaGateway">
3   <ImplementationArtifacts>
4     <ImplementationArtifact name="SedonaControl"
5       interfaceName="SedonaControl"
6       artifactType="toscatypes:WAR"
7       artifactRef="demo:SedonaService" />
8   </ImplementationArtifacts>
9 </NodeTypeImplementation>

```

The “SedonaService” ArtifactTemplate (cf. (1) of Listing 6.9) is used to configure the properties of the Web service by defining the ServiceEndpoint (5), the PortType (8) and InvocationType (11). The location of the WAR-file is defined by the ArtifactReference (16-17).

Listing 6.9: Sedona IA Service Artifact Template.

```

1 <ArtifactTemplate id="SedonaService" type="tns:WAR">
2   <Properties>
3     <opentosca:WSProperties>
4       <opentosca:ServiceEndpoint>
5         /services/NiagaraIAService
6       </opentosca:ServiceEndpoint>
7       <opentosca:PortType>
8         {http://sedona.aws.ia.opentosca.org}SedonaIAService
9       </opentosca:PortType>
10      <opentosca:InvocationType>
11        SOAP/HTTP
12      </opentosca:InvocationType>
13    </opentosca:WSProperties>
14  </Properties>
15  <ArtifactReferences>
16    <ArtifactReference
17      reference="IAs/SedonaService/Sedona-Service.war" />
18  </ArtifactReferences>
19 </ArtifactTemplate>

```

The *NodeTypeImplementation* of the NiagaraGateway node type (cf. (1-2) of Listing 6.10) defines the “NiagaraControl” interface (5) and references the ArtifactTemplate (7) of type “WAR” (6) which is discussed next.

Listing 6.10: Niagara Gateway NodeTypeImplementation.

```

1 <NodeTypeImplementation name="NiagaraGatewayTypeImpl"
2   nodeType="tns:NiagaraGateway">
3   <ImplementationArtifacts>

```

```

4     <ImplementationArtifact name="NiagaraControl"
5         interfaceName="NiagaraControl"
6         artifactType="toscatypes:WAR"
7         artifactRef="demo:NiagaraService" />
8 </ImplementationArtifacts>
9 </NodeTypeImplementation>

```

The “NiagaraService” ArtifactTemplate (cf. (1) of Listing 6.11) is used to configure the properties of the Web service by defining the ServiceEndpoint (5), the PortType (8) and InvocationType (11). The location of the WAR-file is defined by the Artifact-Reference (16-17).

Listing 6.11: Niagara IA Service Artifact Template.

```

1 <ArtifactTemplate id="NiagaraService" type="tns:WAR">
2   <Properties>
3     <opentosca:WSProperties>
4       <opentosca:ServiceEndpoint>
5         /services/NiagaraIAService
6       </opentosca:ServiceEndpoint>
7       <opentosca:PortType>
8         {http://niagara.aws.ia.opentosca.org}NiagaraIAService
9       </opentosca:PortType>
10      <opentosca:InvocationType>
11        SOAP/HTTP
12      </opentosca:InvocationType>
13    </opentosca:WSProperties>
14  </Properties>
15  <ArtifactReferences>
16    <ArtifactReference
17      reference="IAs/NiagaraService/Niagara-Service.war" />
18  </ArtifactReferences>
19 </ArtifactTemplate>

```

The implementation of the “Sedona-Service.war” and the “Niagara-Service.war” Implementation Artifacts is discussed in Section 6.5.

6.2.3 Relationship Type Definitions

Following the definition of the relationship model in Section 5.2, we implement the RelationshipTypes described in Figure 5.9. The abstract DeployedOn RelationshipType (cf. (1) of Listing 6.12) is derived from RootRelationshipType (2) and defines the source and target types it can be attached to (3-4). As the relationship types in our prototype are quite simple, we omit the remaining definitions and refer to the following Section where the RelationshipTypes are instantiated and described in detail.

Listing 6.12: Deployed On RelationshipType.

```

1 <RelationshipType name="DeployedOn" abstract="yes">
2   <DerivedFrom typeRef="tns:RootRelationshipType" />
3   <ValidSource typeRef="tns:ControllerRequirement" />
4   <ValidTarget typeRef="tns:ControllerCapability" />
5 </RelationshipType>

```

6.3 Implementing the IoT Application

Following the use case description of Chapter 4 and the overview of the prototype in the Design chapter (cf. Figure 5.1) we implement an IoT application that consists of four components.



Figure 6.2: Prototype's Application Topology as depicted by the Winery Modeling Tool.

An *Air Handling Unit* representing a Sedona AHU Controller control logic deployed on a *Sedona VM 1.2.28*³ and a *Main Air Handler* representing a Niagara AHU Controller control logic deployed on a *PC M2M JACE*⁴ running the Niagara^{AX} 3.7.106 framework [30, 32] (cf. Section 3.3.1). The latter represents the Niagara gateway. A graphical representation of the prototypical application's topology is depicted in Figure 6.2. The TOSCA *ServiceTemplate* (cf. Listing 6.13) describes the structure as well as the behaviour of the HVAC application. The structure becomes manifest in the *TopologyTemplate* whereas the behaviour is defined by *Plans*. The *Plans* have already been defined in the Design chapter (cf. Section 5.4), thus they are omitted here.

³<http://sedonadev.org/download/build/>

⁴https://www.tridium europe.com/storage/downloads/TridiumEuropeDatasheet_m2mjace_1355839010.pdf

Listing 6.13: HVAC Service Template.

```

1 <ServiceTemplate id="HVAC" name="HVAC Template">
2   <TopologyTemplate>
3     <NodeTemplate id="SedonaAHUController"
4       name="Air Handling Unit"
5       type="sedona:SedonaAHUController">...</NodeTemplate>
6
7     <RelationshipTemplate
8       id="SedonaControllerDeployedOnSedonaGateway"
9       name="SedonaController DeployedOn SedonaGateway"
10      type="sedona:SedonaControllerDeployedOnSedonaGateway" />
11
12    <NodeTemplate id="SedonaGateway" name="Sedona VM"
13      type="sedona:SedonaGateway">...</NodeTemplate>
14
15    <NodeTemplate id="NiagaraAHUController"
16      name="Main Air Handler"
17      type="niagara:NiagaraAHUController">...</NodeTemplate>
18
19    <RelationshipTemplate
20      id="NiagaraControllerDeployedOnNiagaraGateway"
21      name="NiagaraController DeployedOn NiagaraGateway"
22      type="niagara:NiagaraControllerDeployedOnNiagaraGateway"/>
23
24    <NodeTemplate id="NiagaraGateway" name="PC M2M JACE"
25      type="niagara:NiagaraGateway">...</NodeTemplate>
26  </TopologyTemplate>
27
28  <Plans targetNamespace="http://.../ServiceTemplates/HVAC">
29    ...
30  </Plans>
31 </ServiceTemplate>

```

6.3.1 Topology Template

The `TopologyTemplate` (cf. (2-26) of Listing 6.13) consists of four `NodeTemplate` definitions, each instantiating a specific `NodeType` with concrete properties. The concrete node types are imported either from the `sedona` or `niagara` namespace. The composition of these components is completed by two `RelationshipTemplate` definitions which model the relations between the components. Next, each `NodeTemplate` and `RelationshipTemplate` of the `TopologyTemplate` will be discussed in detail.

The *SedonaAHUController* `NodeTemplate` (cf. (1-2) of Listing 6.14) defines `Properties`, `Requirements` and `DeploymentArtifacts`. The `Properties` define the

actual values needed to instantiate a SedonaAHUController Node Type. An initial Setpoint temperature (7) as well as the ManagementInterface (8) for the Air Handler is specified here. The DeploymentArtifact (17-20) defines the SedonaApplication-archive (20) containing the control logic to be deployed on the Sedona gateway. The artifactRef (18) points to the ArtifactTemplate which specifies the exact location of the archive. Finally, the “SedonaController_Environment” Requirement (12-14) describes the necessity of an environment being able to host this Controller.

Listing 6.14: Sedona AHU Controller NodeTemplate.

```

1 <NodeTemplate id="SedonaAHUController" name="Air Handling Unit"
2   type="sedona:SedonaAHUController">
3   <Properties>
4     <sedona:SedonaAHUControllerProperties
5       xmlns:sedona="http://www.example.com/tosca/Types/Sedona"
6       xmlns="http://www.example.com/tosca/Types/Sedona">
7       <Setpoint>21.0</Setpoint>
8       <ManagementInterface>AHUMgmt</ManagementInterface>
9     </sedona:SedonaAHUControllerProperties>
10  </Properties>
11  <Requirements>
12    <Requirement id="SedonaController_Environment"
13      name="container"
14      type="sedona:SedonaControllerRequirement" />
15  </Requirements>
16  <DeploymentArtifacts>
17    <DeploymentArtifact
18      artifactRef="at-c5a973c5-3823-4f24-ae28-f944fe896666"
19      artifactType="baseTypes:ArchiveArtifact"
20      name="SedonaApplication-archive" />
21  </DeploymentArtifacts>
22 </NodeTemplate>

```

The *SedonaController_DeployedOn_SedonaGatewayRelationshipTemplate* (cf. Listing 6.15) defines the “deployedOn” relationship between the SedonaAHUController node template and the SedonaGateway node template. The relationship connects the Requirement for a hosting environment (SourceElement) to the Capability of hosting an application (TargetElement).

Listing 6.15: Sedona Controller RelationshipTemplate.

```

1 <RelationshipTemplate name="deployed on"
2   id="SedonaController_DeployedOn_SedonaGateway"
3   type="sedona:SedonaControllerDeployedOnSedonaGateway">
4   <SourceElement ref="SedonaController_Environment" />
5   <TargetElement ref="SedonaController_Application" />
6 </RelationshipTemplate>

```

The *SedonaGateway* NodeTemplate defines Properties and Capabilities (cf. Listing 6.16). The Properties definition configures the NodeTemplate with the parameters needed to connect to the Sedona VM gateway. The “SedonaControllerApplication” Capability referenced by the *SedonaController_DeployedOn_SedonaGateway* relationship template defines the ability of handling the deployment of a Sedona controller application on the gateway’s application environment.

Listing 6.16: Sedona Gateway NodeTemplate.

```

1 <NodeTemplate id="SedonaGateway" name="Sedona VM"
2   type="sedona:SedonaGateway">
3   <Properties>
4     <sedona:SedonaGatewayProperties>
5       <Host>localhost</Host>
6       <Username>admin</Username>
7       <Password>*****</Password>
8       <SoxPort>1876</SoxPort>
9     </sedona:SedonaGatewayProperties>
10  </Properties>
11  <Capabilities>
12    <Capability id="SedonaController_Application"
13      name="Environment for Sedona controller deployment."
14      type="sedona:SedonaControllerCapability" />
15  </Capabilities>
16 </NodeTemplate>

```

The *NiagaraAHUController* NodeTemplate (cf. (1-2) of Listing 6.17) defines Properties, Requirements and DeploymentArtifacts. The Properties define the actual values needed to instantiate a NiagaraAHUController Node Type. An initial Setpoint temperature (10) is specified as well as the ManagementInterface (12) for the Main Air Handler. As a Niagara Station running on a Niagara Platform requires additional log-in information, the credentials are provided here (8-9). The DeploymentArtifact (22-25) defines the NiagaraApplication-archive containing the control logic (the Niagara Station) to be deployed on the Niagara gateway. The artifactRef (23) points to the ArtifactTemplate which specifies the exact location of the archive. Finally, the “NiagaraController_Environment” Requirement (17-19) describes the necessity of an environment being able to host this Controller.

Listing 6.17: Niagara AHU Controller NodeTemplate.

```

1 <NodeTemplate id="NiagaraAHUController" name="Main Air Handler"
2   type="niagara:NiagaraAHUController">
3   <Properties>
4     <niagara:NiagaraAHUControllerProperties
5       xmlns:niagara="http://www.example.com/tosca/Types/Niagara"
6       xmlns="http://www.example.com/tosca/Types/Niagara">

```

```

7      <!-- Niagara Station on M2M JACE -->
8      <Username>admin</Username>
9      <Password>*****</Password>
10     <Setpoint>21.0</Setpoint>
11     <ManagementInterface>
12       http://128.131.172.101/obix/config/AHUManagement/
13     </ManagementInterface>
14   </niagara:NiagaraAHUControllerProperties>
15 </Properties>
16 <Requirements>
17   <Requirement id="NiagaraController_Environment"
18     name="container"
19     type="niagara:NiagaraControllerRequirement" />
20 </Requirements>
21 <DeploymentArtifacts>
22   <DeploymentArtifact
23     artifactRef="at-c5a973c5-3823-4f24-ae28-f944fe897777"
24     artifactType="baseTypes:ArchiveArtifact"
25     name="NiagaraApplication-archive" />
26 </DeploymentArtifacts>
27 </NodeTemplate>

```

The *NiagaraController_DeployedOn_NiagaraGateway* RelationshipTemplate (cf. Listing 6.18) defines the “deployedOn” relationship between the NiagaraAHUController node template and the NiagaraGateway node template. The relationship connects the Requirement for a hosting environment (SourceElement) to the Capability of hosting an application (TargetElement).

Listing 6.18: Niagara Controller RelationshipTemplate.

```

1 <RelationshipTemplate name="deployed on"
2   id="NiagaraController_DeployedOn_NiagaraGateway"
3   type="niagara:NiagaraControllerDeployedOnNiagaraGateway">
4   <SourceElement ref="NiagaraController_Environment" />
5   <TargetElement ref="NiagaraController_Application" />
6 </RelationshipTemplate>

```

The *NiagaraGateway* NodeTemplate defines Properties and Capabilities (cf. Listing 6.19). The Properties definition configures the NodeTemplate with the parameters needed to connect to the Niagara platform running on the gateway. The LifecycleManagementProxyUrl defines the Obix Url to access the *LifecycleManagementProxy* component running on the Niagara Supervisor Station. Thus, the credentials for the Supervisor Station are defined too. The NiagaraStationDB specifies the path to the local stations database of the Niagara installation. The “NiagaraController_Application” Capability referenced by the *NiagaraController_DeployedOn_SedonaGateway* relationship template defines the ability

of the gateway’s application environment to handle the deployment of a Niagara controller application.

Listing 6.19: Niagara Gateway NodeTemplate.

```

1 <NodeTemplate id="NiagaraGateway" name="PC M2M JACE"
2   type="niagara:NiagaraGateway">
3   <Properties>
4     <niagara:NiagaraGatewayProperties>
5       <!-- Platform on M2M JACE -->
6       <Host>128.131.172.101</Host>
7       <Port>3011</Port>
8       <Username>tridium</Username>
9       <Password>*****</Password>
10      <!-- Niagara Supervisor Station on server,
11        acting as Proxy -->
12      <LifecycleManagementProxyUrl>
13        http://localhost/obix/config/LifecycleManagementProxy/
14      </LifecycleManagementProxyUrl>
15      <ProxyUsername>admin</ProxyUsername>
16      <ProxyPassword>*****</ProxyPassword>
17      <NiagaraStationDB>
18        C:\Niagara\Niagara-3.7.106\stations
19      </NiagaraStationDB>
20    </niagara:NiagaraGatewayProperties>
21  </Properties>
22  <Capabilities>
23    <Capability id="NiagaraController_Application" name="app"
24      type="niagara:NiagaraControllerCapability" />
25  </Capabilities>
26 </NodeTemplate>

```

The “SedonaApplication-archive” ArtifactTemplate depicted in Listing 6.20 is of type=“base:ArchiveArtifact“. This generic Base Artifact Type was deliberately chosen so any TOSCA environment can basically handle it, without knowing what it actually contains. The logic on how to process the Sedona application image files is contained in the implementation of the Sedona Gateway Node Type. The ArchiveArtifactProperties contain information about the type (“zip”) of the archive. The ArtifactReference points to the zip-file containing the Sedona application image files.

Listing 6.20: Sedona Air Handler DeploymentArtifact ArtifactTemplate.

```

1 <ArtifactTemplate id="at-c5a973c5-3823-4f24-ae28-f944fe896666"
2   type="base:ArchiveArtifact" name="SedonaApplication-archive">
3   <Properties>
4     <base:ArchiveArtifactProperties

```



```

5      xmlns:base="http://.../tosca/ns/2011/12/ToscaBaseTypes"
6      xmlns="http://.../ns/2011/12/ToscaBaseTypes">
7      <ArchiveInformation archiveType="zip"
8          archiveReference="files/SedonaAirHandler.zip" />
9      </base:ArchiveArtifactProperties>
10 </Properties>
11 <ArtifactReferences>
12     <ArtifactReference reference="files/SedonaAirHandler.zip" />
13 </ArtifactReferences>
14 </ArtifactTemplate>

```

The “NiagaraApplication-archive” ArtifactTemplate depicted in Listing 6.21 follows the same concept, only that it’s ArtifactReference points to the zip-file containing the Niagara Station image files.

Listing 6.21: Niagara Air Handler DeploymentArtifact ArtifactTemplate.

```

1 <ArtifactTemplate id="at-c5a973c5-3823-4f24-ae28-f944fe897777"
2   type="base:ArchiveArtifact" name="NiagaraApplication-archive">
3   <Properties>
4       <base:ArchiveArtifactProperties
5           xmlns:base="http://.../tosca/ns/2011/12/ToscaBaseTypes"
6           xmlns="http://.../tosca/ns/2011/12/ToscaBaseTypes">
7           <ArchiveInformation archiveType="zip"
8               archiveReference="files/NiagaraAirHandler.zip" />
9           </base:ArchiveArtifactProperties>
10  </Properties>
11  <ArtifactReferences>
12      <ArtifactReference reference="files/NiagaraAirHandler.zip"/>
13  </ArtifactReferences>
14 </ArtifactTemplate>

```

6.4 AHU Controller Gateway Applications

The *SedonaAHUController* and the *NiagaraAHUController* defined in the previous Section reference a *Deployment Artifact* which describes the device-specific control logic to be deployed on the gateways. For demonstration purpose we use the AHU control logic (cf. Figure 6.3) of the Niagara^{AX} [41] demo station. In this demo station the control logic is visualized by a graphical representation of an Air Handler (cf. Figure 4.1). The Sedona implementation uses a similar AHU control logic.

6.4.1 Sedona Interface

To implement the management interface for the Sedona AHU Controller, a custom Sedona component was developed. The `tosca::AHUManagement` component depicted in Figure 6.4 is

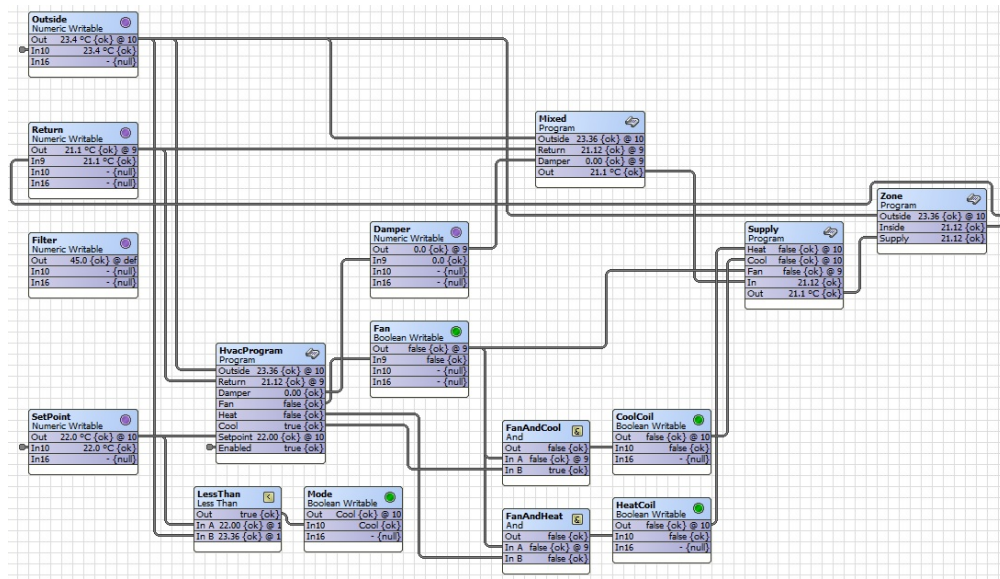


Figure 6.3: Niagara Workbench wire-sheet view of Air Handler control logic. Adapted from the Niagara Framework [41] Demo Station.

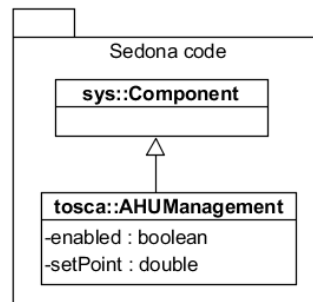


Figure 6.4: Sedona Air Handling Unit Management component.

derived from the more general `sys::Component` class and exposes the properties `enabled` and `setpoint` of the AHU Controller it is connected to. The Sedona code implementing the `AHUMangement` component is depicted in Listing 6.22.

Listing 6.22: Sedona AHUMangement Component.

```

1 public class AHUMangement extends Component
2 {
3     @config property bool enabled = false
4
5     @unit=Units.celsius
6     @config property float setpoint = 21.0
7 }

```

This interface component representing an AHU Controller implementation must be placed below the App node of the Sedona application (cf. Figure 6.5 (a)) and the properties must be connected to the corresponding AHU Controller (“AHUImpl”) to propagate property changes immediately. The name of the interface (e.g. “AHUMgmt”) is configured in the `ManagementInterface` property of the controller’s node template (cf. Section 6.3), which is the *SedonaAHUController* node template in our prototype. If the value of the `enabled` property is set to `true`, the Controller is in state “Running”, whereas `false` means it is in state “Stopped”. These properties are changed via the Sox Service running on the target Sedona gateway environment.

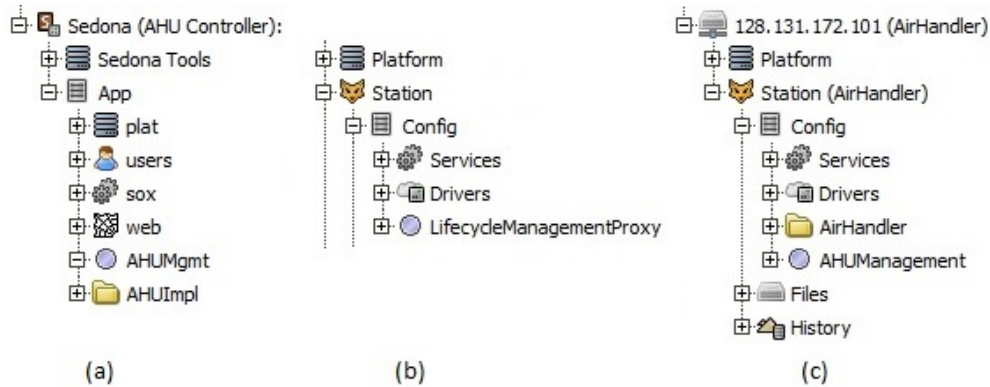


Figure 6.5: AHU Management Interface and Implementation as displayed in the Niagara Workbench. (a) Sedona Application. (b) Niagara Supervisor Station. (c) Niagara Station.

6.4.2 Niagara Interface

The `BAHUManagement` component depicted in Figure 6.6 implements the `AHUManagement` Obix Interface (cf. Table 5.2). It is derived from the basic Niagara `BComponent` class. It defines the `enabled` and `setpoint` properties which are changed by invoking the methods

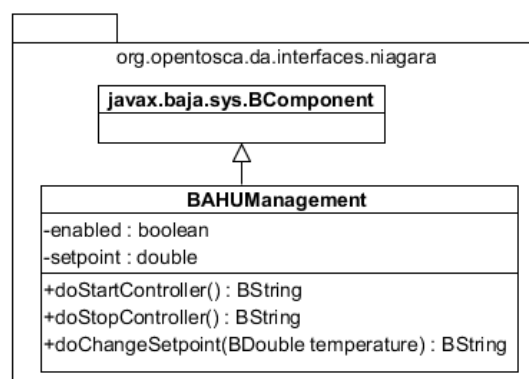


Figure 6.6: Niagara Air Handling Unit Management component.

`doStartController()`, `doStopController()` and `doChangeSetpoint(BDouble temperature)`. Listing 6.23 shows the declarative definition of the properties and actions. The Slot-o-matic tool of the Eclipse Niagara Plugin 7.1.7⁵ was used to generate boiler-plate code from the declarative component definition.

Listing 6.23: Declarative definition of Niagara BAHUManagement component.

```

1  /*-
2   class BAHUManagement
3   {
4       properties
5       {
6           enabled: boolean
7           default {[ false ]}
8
9           setpoint: double
10          default {[ 21.0 ]}
11      }
12      actions
13      {
14          startController()
15          stopController()
16          changeSetpoint(temperature: BDouble)
17              default{[ 22.0 ]}
18      }
19  }
20  -*/

```

This interface component representing an AHU Controller implementation that can be placed at any location of the Niagara target Station, as the Obix Service allows direct access (e.g. via `http://<host>/obix/config/AHUManagement/`) to the *AHUManagement* component. This Url indicates that the component was placed below the `config` node of the Niagara Station. It is configured via the `ManagementInterface` property of the controller's node template, which is the *NiagaraAHUController* node template in our prototype. The properties (Niagara Slots) must be connected to the corresponding AHU Controller implementation to propagate property changes immediately. The `enabled` property of the *AHUManagement* component was connected to the “enabled” slot of the *HvacProgram* component and the `setpoint` property to the “In10” slot of the *SetPoint* component of the Niagara control logic (cf. Figure 6.3). The Niagara navigation tree depicted in Figure 6.5 (c) shows the *AHUManagement* component and the *AirHandler* folder containing the control logic as part of the “AirHandler” Station's *Config* node.

⁵<https://community.niagara-central.com/ord?portal:/blog/BlogEntry/273>

The Implementation Artifacts realizing the management operations for Niagara and Sedona node types are implemented as Web services. The class structure of both Services is depicted in Figure 6.7. The SedonaIService implements all life-cycle interfaces (cf. Section 5.2.2) related



⁶<http://files.opentosca.de/v1.1/>

Deployment Artifacts from the Container’s File store. *IUnpacker* is implemented by *Unpacker* which uses the *ZipManager* to unzip the downloaded zip files. The *DirectoryVisitor* allows to process the files of a Deployment Artifact after it was uncompressed.

6.5.1 Sedona

The SedonaIAService uses the *SedonaGatewayConnection* to connect to a gateway via the *Sox-Client*. Listing 6.24 depicts the process of deploying and configuring a Sedona AHU Controller application. First, a SoxClient connection based on a DaspSocket is established to connect to the target gateway. The basic provisioning steps are depicted in Lines 7-13. It includes copying the application files (7-8) with exactly these names followed by a renaming (9-10) of the very same files. The provisioning process is completed by restarting the gateway application (13). The AHU Controller is then configured by looking up the management interface (15-16) and changing the setpoint temperature (17-18). Finally, the changes are saved (21).

Listing 6.24: Deploy Sedona Control Application.

```
1 DaspSocket dasp =
2   DaspSocket.open(-1, null, DaspSocket.SESSION_QUEUEING);
3 SoxClient sox = new SoxClient(dasp,
4   InetAddress.getByAddress(host), port, username, password);
5 sox.connect();
6 // provision
7 sox.putFile("app.sab.writing", SoxFile.make(appSab));
8 sox.putFile("kits.scode.writing", SoxFile.make(scodeBin));
9 sox.renameFile("app.sab.writing", "app.sab.stage");
10 sox.renameFile("kits.scode.writing", "kits.scode.stage");
11 // reboot
12 SoxComponent app = sox.loadApp();
13 sox.invoke(app, app.slot("restart"), null);
14 // change setpoint
15 SoxComponent managementComp = sox.loadApp().
16   getChild(managementComponentName);
17 sox.write(managementComp, managementComp.slot("setpoint"),
18   sedona.Float.make((float) temperature));
19 // save changes
20 app = sox.loadApp();
21 sox.invoke(app, app.slot("save"), null);
```

6.5.2 Niagara

As discussed in the Design chapter (cf. Section 5.3), the NiagaraIAService uses two distinct interfaces for deployment and configuration. The NiagaraIAService uses the *NiagaraProxy-Connection* to invoke the “reboot”, “deploy” and “undeploy” operations of the LifecycleManagementProxy (cf. Section 6.5.3). Listing 6.25 outlines how a new *ObixSession* object is created

based on the Url of the LifecycleManagementProxy component together with the credentials (1-2).

Listing 6.25: Invoke Deployment Action Method on LifecycleManagementProxy.

```
1 ObixSession os = new ObixSession(new Uri(
2   lifecycleManagementProxyUrl), proxyUsername, proxyPassword);
3 os.invoke(lifecycleManagementProxyUrl+"deploy/", new Str(
4   paramsToString(host, port, username, password, stationName));
```

In a second step, the “deploy” action is invoked (3-4) with the parameters defined in Table 6.1. The LifecycleManagementProxy is discussed in the next Section.

To configure the AHU Controller, the NiagaraIAService invokes the management operations defined by Table 5.2. Lines (3-4) of Listing 6.26 show the invocation of the “changeSetpoint” operation of the AHUManagement component.

Listing 6.26: Change Setpoint Temperature via Niagara AHUManagement Interface.

```
1 ObixSession os = new ObixSession(
2   new Uri(ahuManagementUrl), stationUsername, stationPassword);
3 os.invoke(ahuManagementUrl+"changeSetpoint/",
4   new obix.Real(temperature));
```

6.5.3 Niagara Proxy

As discussed earlier, the architecture of the Niagara Environment requires an extra component, namely the *LifecycleManagementProxy*, that is part of a Niagara Supervisor Station. The NiagaraIAService is configured with the location of the local Station Database defined by the NiagaraStationDB property of the NiagaraGateway node template. During application deployment, the NiagaraIAService copies the new Niagara Station image to the Station Database before invoking the deployment operation of the LifecycleManagementProxy running on the Supervisor Station. The BLifecycleManagementProxy component depicted in Figure 6.8 is derived from the basic Niagara BComponent class. It defines the life-cycle operations doDeploy(BStringparams), doUndeploy(BStringparams) and doReboot(BString params) as Obix action handlers which can be invoked via the Obix Service running on the Niagara Supervisor Station. Due to the restriction of Obix action invocations to one parameter only, the sole parameter is passed to the operations as a comma separated list of the parameters (cf. Table 6.1). The Action Handlers implement the Obix actions of the interface definition of the LifecycleManagementProxy Obix Interface (cf. Table 5.1).

Action Handlers	Comma Separated Parameter Strings
doDeploy	“host,port,username,password,stationName”
doUndeploy	“host,port,username,password”
doReboot	“host,port,username,password”

Table 6.1: LifecycleManagementProxy methods with parameters.

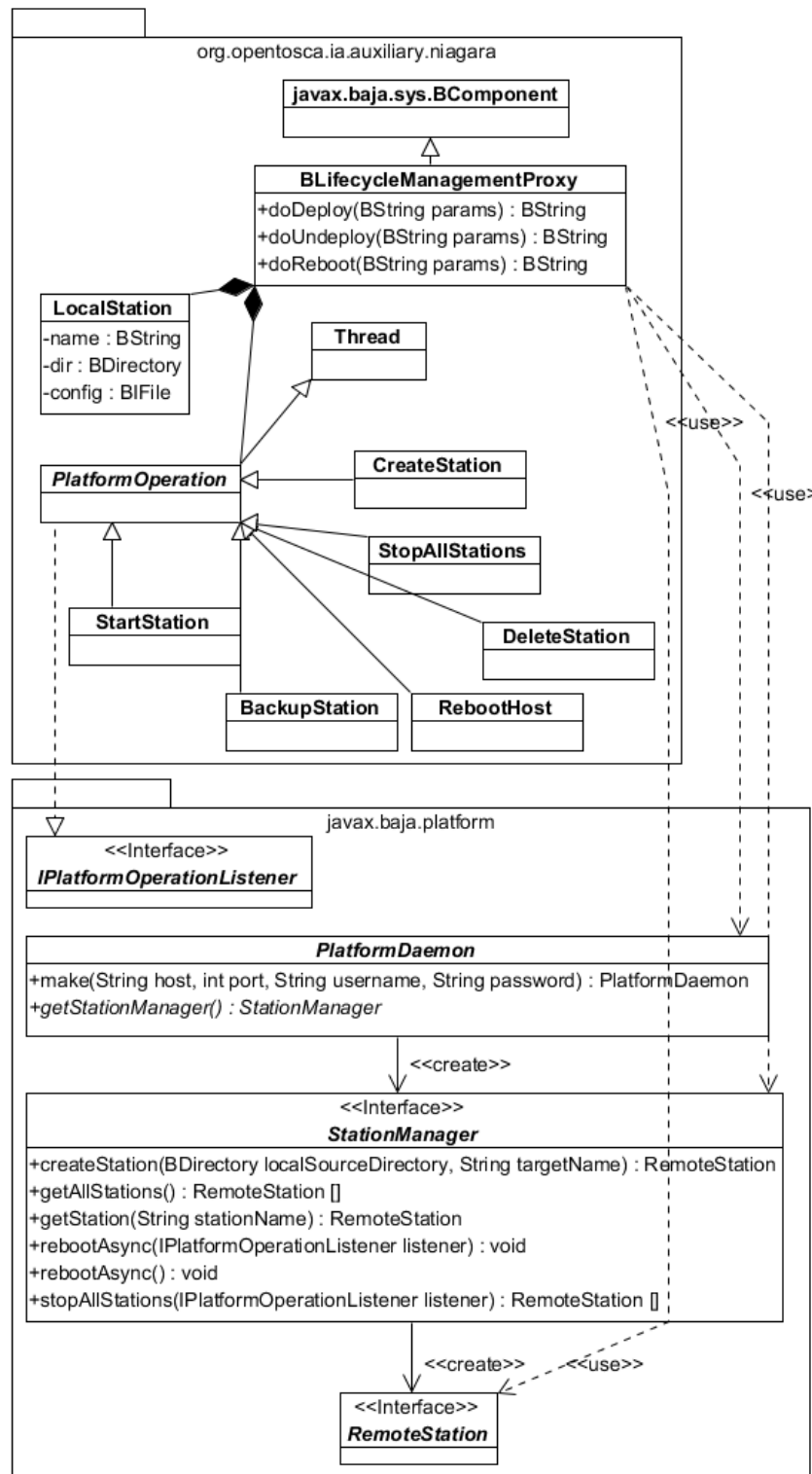


Figure 6.8: BLifecycleManagementProxy component for the Niagara Supervisor Station.

As depicted by the Niagara architecture (cf. Figure 5.11), the deployment and undeployment capabilities are implemented by the *LifecycleManagementProxy* component installed on a *Niagara Supervisor Station*. In the prototype, the *LifecycleManagementProxy* was placed as a child node of the *Config* node of the Supervisor Station depicted in Figure 6.5 (b).

The *BLifecycleManagementProxy* (cf. Figure 6.8) is an adaption of the *PlatformUtil*⁷ that uses the `javax.baja.platform` API. The *BLifecycleManagementProxy* uses platform and station specific life-cycle management functionality defined by the `javax.baja.platform` package to compose the higher-level management procedures. The *StationManager* interface (cf. Figure 6.8) offers station management functions, for example to create a new station on a remote gateway, retrieve access to an existing station or to reboot a remote gateway. The *RemoteStation* interface allows to modify, start, stop and delete a station running on a remote gateway. The life-cycle methods defined by the *BLifecycleManagementProxy* use *PlatformOperations* implementing the *IPlatformOperationListener* interface to start long-running platform tasks by spawning a new *Thread* for each operation. Platform operations to create, start, stop, backup and delete stations as well as rebooting a gateway are defined. The *LocalStation* is used to represent stations retrieved from the Niagara Station Database which then can be copied to a remote gateway environment during deployment. The deployment process is depicted in Listing 6.27 where the abstract *PlatformDaemon* factory (1-2) is used to connect to the platform daemon of the target gateway environment. The *CreateStation* platform operation copies the station from the Niagara Station DB to the target gateway (3-5). The Station creation is performed by the *StationManager* (7).

Listing 6.27: Niagara Station Deployment by *BLifecycleManagementProxy*.

```
1 PlatformDaemon daemon =
2   PlatformDaemon.make(host, port, username, password);
3 CreateStation platformOp =
4   new CreateStation(station.dir, station.name, resultLog);
5 platformOp.execute();
6 // performed by CreateStation:
7 daemon.getStationManager().createStation(dir, name, this);
```

6.6 Implementing the Life-cycle Management Procedures

The Plan definitions described in Section 5.4 are implemented using the WS-BPEL 2.0 workflow language together with XPath 2.0. To access the REST API of the OpenTOSCA container, the BPEL4RestLight Extension (cf. Listing 6.28) is used.

In the following, the most important concepts used by the management procedures are explained. Sequence diagrams are used to depict the management operations that get invoked by the BPEL workflows incorporating all subsequent major invocations on architectural components involved in the deployment and management of the application – from high-level management plans to gateway interfaces.

⁷Niagara Ord: `module://docSource/demoAppliance/appliance.ui/PlatformUtil.java`

Listing 6.28: Defining the BPEL4RestLight Extension.

```

1 <bpel:extensions>
2   <bpel:extension
3     namespace="http://iaas.uni-stuttgart.de/
4     bpel/extensions/bpel4restlight" mustUnderstand="yes" />
5 </bpel:extensions>

```

6.6.1 Application Deployment

The application deployment plan (cf. Section 5.4.1) consists of a common part (cf. Figure 6.9) and management operations specific to the Sedona and Niagara environment. The HVAC De-

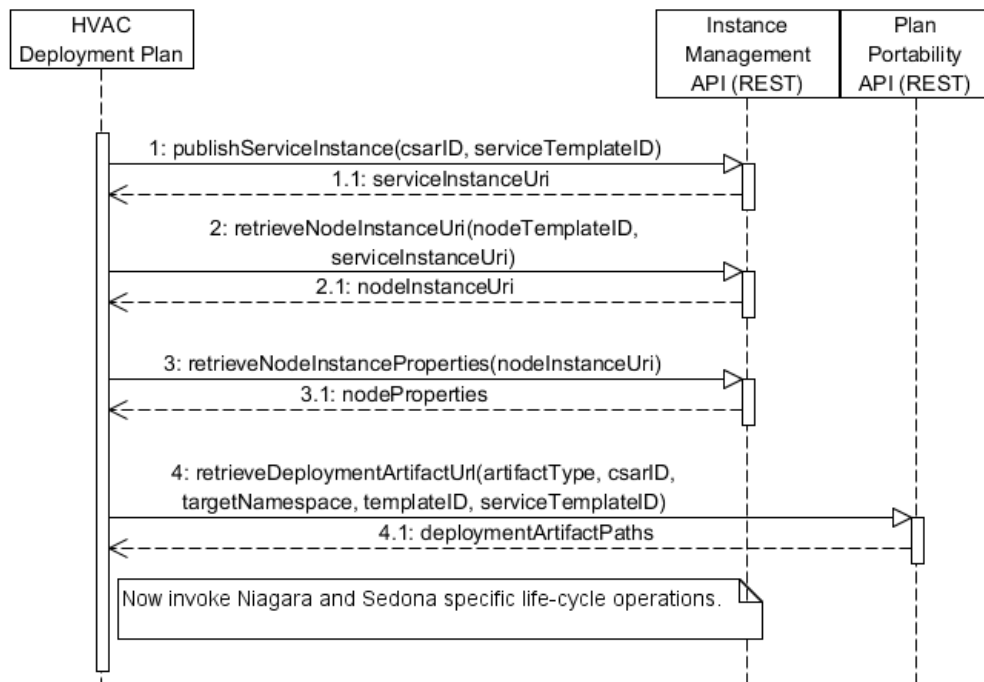


Figure 6.9: Common part of the HVAC Build Plan.

ployment Plan first publishes the Service Instance (cf. Figure 6.9, 1) to the Instance Management, which instantiates the selected Service Template defined in the CSAR file. Listing 6.29 shows how the Service Template is selected by the *csarID* (e.g. HVAC.csar) and *serviceTemplateID* (e.g. http://localhost:1337/containerapi) parameters. The workflow receives the *ServiceInstanceUri* in response.

Step (2) of Figure 6.9 is a repetitive task to retrieve the Uri of the Node Instance related to the Node Template defined in the Topology Template. An example for retrieving the *nodeInstanceUri* of a *NodeTemplate* is depicted in Listing 6.30. The node instances are identified by the Uri of the Service Instance together with the *NodeTemplate* identifiers.

Listing 6.29: Publish the Service Instance.

```

1 <bpel:extensionActivity>
2   <bpel4RestLight:POST
3     uri="$bpelvar[ContainerURL]/instancedata/serviceInstances?
4       csarID=$bpelvar[CSARName];
5       serviceTemplateID={http://.../ServiceTemplates/HVAC}HVAC"
6       accept="application/xml"
7     response="instanceAPIResponse">
8   </bpel4RestLight:POST>
9 </bpel:extensionActivity>

```

Listing 6.30: Node Instance Uri retrieval.

```

1 <bpel:extensionActivity>
2   <bpel4RestLight:GET
3     uri="$bpelvar[ContainerURL]/instancedata/nodeInstances?
4       nodeTemplateID={.../ServiceTemplates/HVAC}SedonaGateway;
5       serviceInstanceID=$bpelvar[theServiceInstance_URI]"
6       accept="application/xml"
7     response="instanceAPIResponse">
8   </bpel4RestLight:GET>
9 </bpel:extensionActivity>

```

The properties of each Node Instance are retrieved in Step (3) of Figure 6.9. The Plan Portability API is used to query for the Urls where the Deployment Artifacts can be downloaded from (Figure 6.9, 4). Listing 6.31 shows the process of querying for the Deployment Artifact of the SedonaAHUController. First, the information about the Deployment Artifacts is retrieved from the Plan Portability API (Listing 6.31, Lines 1-12). In a second step, a XPath 2.0 query is used (Listing 6.31, Lines 16-24) to acquire the Url where the “zip” file is located.

Listing 6.31: Query for the Sedona Deployment Artifact download URL.

```

1 <bpel:extensionActivity>
2   <bpel4RestLight:GET
3     uri="$bpelvar[ContainerURL]/portability/artifacts?
4       artifactType=DA;
5       csarID=HVAC.csar;
6       targetNamespace=http://.../tosca/ServiceTemplates/HVAC;
7       templateID=SedonaAHUController;
8       serviceTemplateID=HVAC"
9     accept="application/xml"
10    response="sedona_DeploymentArtifacts">
11   </bpel4RestLight:GET>
12 </bpel:extensionActivity>

```

```

13 <bpel:assign name="sedonaDeploymentArtifactPathAssign">
14   <bpel:copy>
15     <bpel:from variable="sedona_DeploymentArtifacts">
16       <bpel:query
17         queryLanguage="...:wsbpel:2.0:sublang:xpath2.0">
18         <![CDATA[//*[local-name()='Artifacts']
19           /*[local-name()='deploymentArtifacts']
20           /*[local-name()='deploymentArtifact'
21             and @name='SedonaApplication-archive']
22           /*[local-name()='references']
23           /*[local-name()='ref']/text()]]>
24       </bpel:query>
25     </bpel:from>
26     <bpel:to variable="sedona_DA_path" />
27   </bpel:copy>
28 </bpel:assign>

```

The deployment procedures for Sedona and Niagara are then performed concurrently. The Niagara specific deployment sequence is depicted in Figure 6.10. The “deploy” management operation (Figure 6.10, 1) provided by the *NiagaraIAService* is invoked through the Container’s *Invoker Service* (cf. Section 5.3). The *invokeOperationSync* data structure (cf. Listing 6.32) is configured to invoke the “deploy” operation (cf. Listing 6.32, 12) of the “NiagaraControl” interface (cf. Section 6.2.2) of the “NiagaraGateway” node template. The parameters (13-34) adhere to the interface definition of Section 5.2.2 and are provided as key value pairs. The properties retrieved from the instance management are then copied to the corresponding elements of the invocation message (not shown here).

Listing 6.32: Define parameters for Niagara Controller deployment.

```

1 <impl:invokeOperationSync
2   xmlns:impl="http://siserver.org/schema"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4   <impl:CsarID>value</impl:CsarID>
5   <impl:ServiceTemplateIDNamespaceURI>
6     http://www.example.com/tosca/ServiceTemplates/HVAC
7   </impl:ServiceTemplateIDNamespaceURI>
8   <impl:ServiceTemplateIDLocalPart>HVAC
9   </impl:ServiceTemplateIDLocalPart>
10  <impl:NodeTemplateID>NiagaraGateway</impl:NodeTemplateID>
11  <impl:InterfaceName>NiagaraControl</impl:InterfaceName>
12  <impl:OperationName>deploy</impl:OperationName>
13  <impl:Params>
14    <impl:Param>
15      <impl:key>host</impl:key>
16      <impl:value>value</impl:value>

```

```

17     </impl:Param>
18     <impl:Param>
19         <impl:key>port</impl:key>
20         <impl:value>value</impl:value>
21     </impl:Param>
22     <impl:Param>
23         <impl:key>username</impl:key>
24         <impl:value>value</impl:value>
25     </impl:Param>
26     <impl:Param>
27         <impl:key>password</impl:key>
28         <impl:value>value</impl:value>
29     </impl:Param>
30     <impl:Param>
31         <impl:key>deploymentArtifactUrl</impl:key>
32         <impl:value>value</impl:value>
33     </impl:Param>
34 </impl:Params>
35 </impl:invokeOperationSync>

```

The Service Invoker service is then invoked with the `invokeOperationSync` message (cf. Listing 6.33) leading to a selection and invocation of the “deploy” management procedure of the NiagaraIAService.

Listing 6.33: Invoke Niagara Deployment through Service Invoker.

```

1 <bpel:invoke name="deployNiagaraControllerSI"
2   partnerLink="SIInvokerPL"
3   operation="invokeOperationSync"
4   inputVariable="niagara_invokerSyncRequest"
5   outputVariable="niagara_invokerResponse" />

```

The NiagaraIAService then downloads the Deployment Artifact from the Container’s file store and copies it to the Station Database of the Niagara Supervisor installation. Now the *Lifecycle Management Proxy* (cf. Section 6.5.3) running on the Supervisor Station is invoked to create a new Niagara Station by copying it to the *Target Platform* and rebooting the target Gateway. Next, the AHU Controller implemented by the Station is configured by changing the Setpoint temperature (Figure 6.10, 2). From now on, the NiagaraIAService invokes the *AHU Management* component running on the target Gateway directly. After the Air Handler is configured, the Controller is started (Figure 6.10, 3) again via the *AHU Management*. Finally, all Node Instance states are set to state “Running” (Figure 6.10, 4).

The Sedona specific deployment sequence is depicted in Figure 6.11. The “deploy” management operation (Figure 6.11, 1) provided by the *SedonaIAService* is invoked through the Container’s *Invoker Service* (like the invocation of the NiagaraIAService discussed earlier). The SedonaIAService then downloads the Deployment Artifact from the Container’s file store and uses the *Sedona Gateway Connection* to provision it via the *Gateway Sox Service* to the Gate-

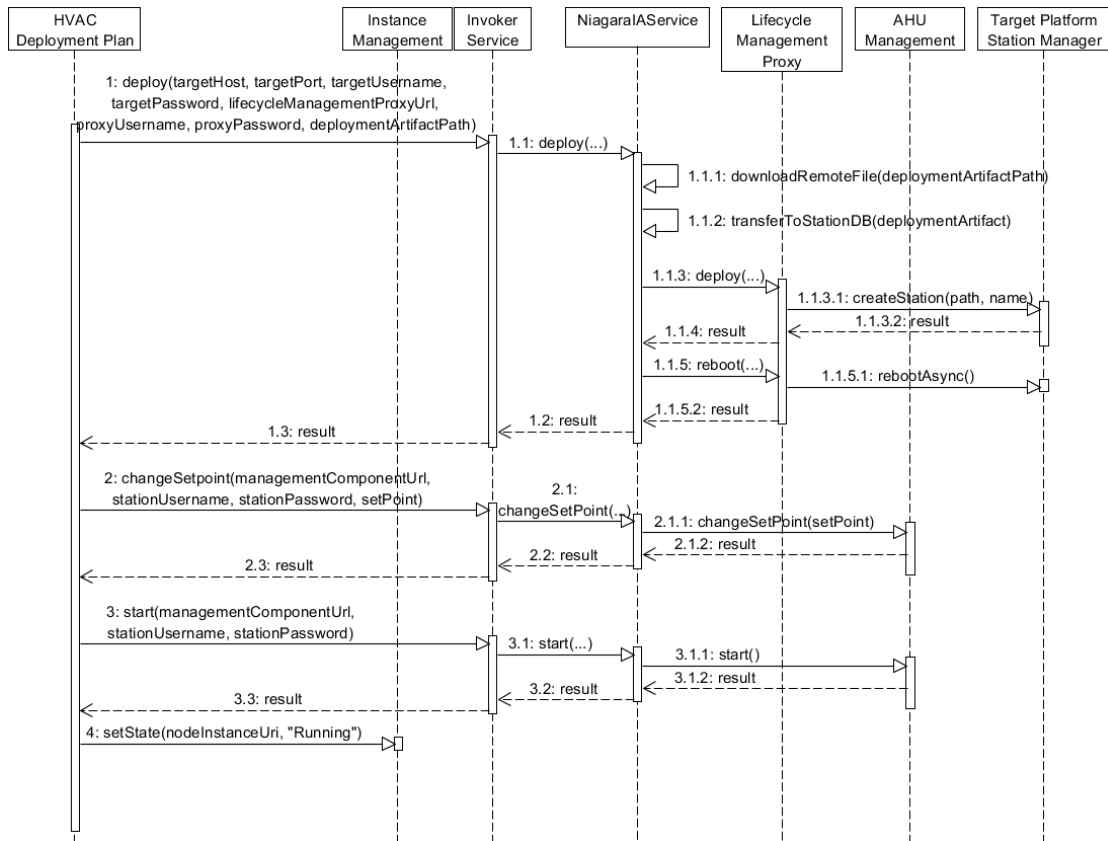


Figure 6.10: Niagara Deployment Procedure of HVAC Build Plan.

way. The Gateway is then restarted to load the new application. Next, the AHU Controller implemented by the Sedona application is configured by changing the Setpoint temperature (Figure 6.11, 2). After the Air Handler is configured, the Controller is started (Figure 6.11, 3) and the Node Instance states are set to state “Running” (Figure 6.11, 4).

6.6.2 Termination Plan

The interaction of the HVAC Termination Plan (cf. Section 5.4.2) with the Instance Management API is illustrated in Figure 6.12. First, the node instance Uri’s (e.g. `http://localhost:1337/containerapi/instancedata/serviceInstances/1`) of the application instance are retrieved (1) to query for the properties (2) needed to undeploy the gateway applications. After invoking the Niagara and Sedona specific life-cycle operations, the Service Instance is unpublished from the Instance Management (3).

The Niagara specific undeployment sequence is depicted in Figure 6.13. The properties retrieved in the general part of the plan are used to invoke the “stop” operation of the Niagara-IAService via the Invoker Service (1). The `managementComponentUrl` is used to connect to the AHUManagement component on the Niagara gateway where the Controller is stopped.

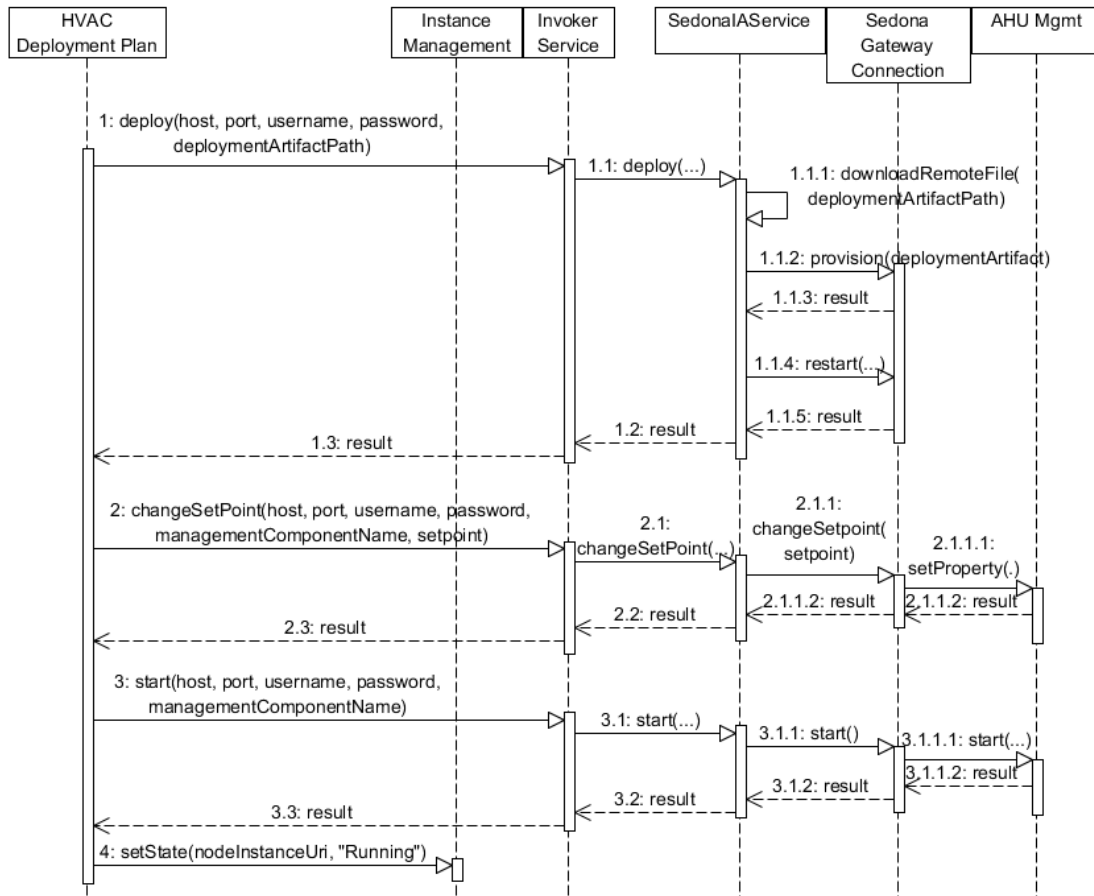


Figure 6.11: Sedona Deployment Procedure of HVAC Build Plan.

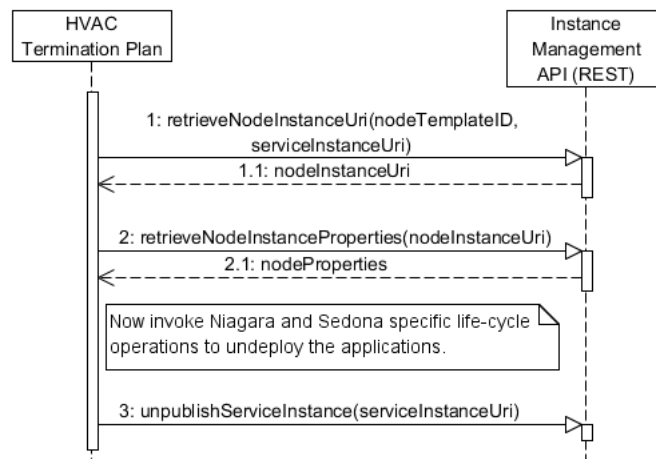


Figure 6.12: HVAC Termination Plan.

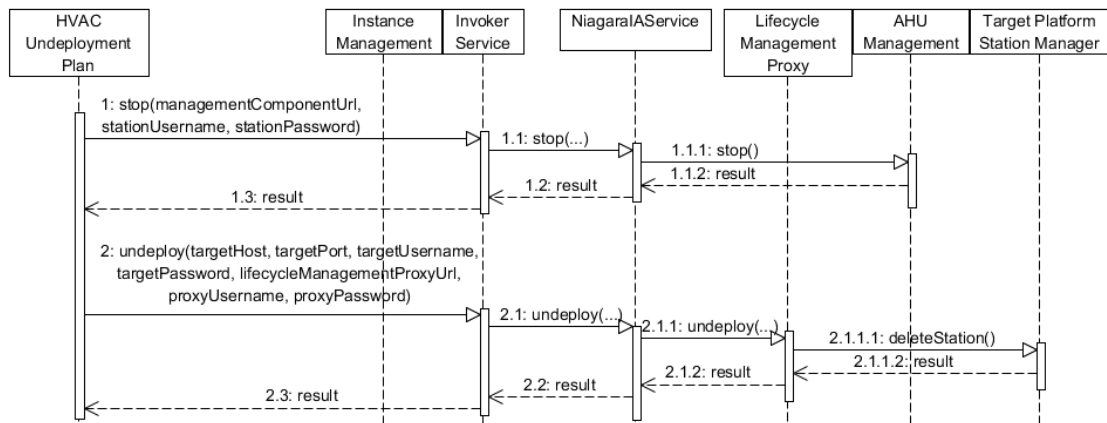


Figure 6.13: Niagara Undeployment Procedure of Application Termination Plan.

The “undeploy” operation of the Lifecycle Management Proxy component on the Supervisor station is invoked to perform a “deleteStation” platform operation to delete the controller application from the target gateway platform (2). A reboot is not necessary.

The Sedona specific undeployment sequence is depicted in Figure 6.14. The properties retrieved in the general part of the plan are used to invoke the “stop” operation of the SedonaIAService via the Invoker Service (1). The `managementComponentName` is used to connect to the AHUMgmt component on the Sedona gateway to stop the AHU Controller. The “undeploy” operation (2) of the SedonaIAService decommissions the AHU Controller by deploying a minimal Sedona application which runs the required Sox service. A gateway reboot completes the undeployment procedure.

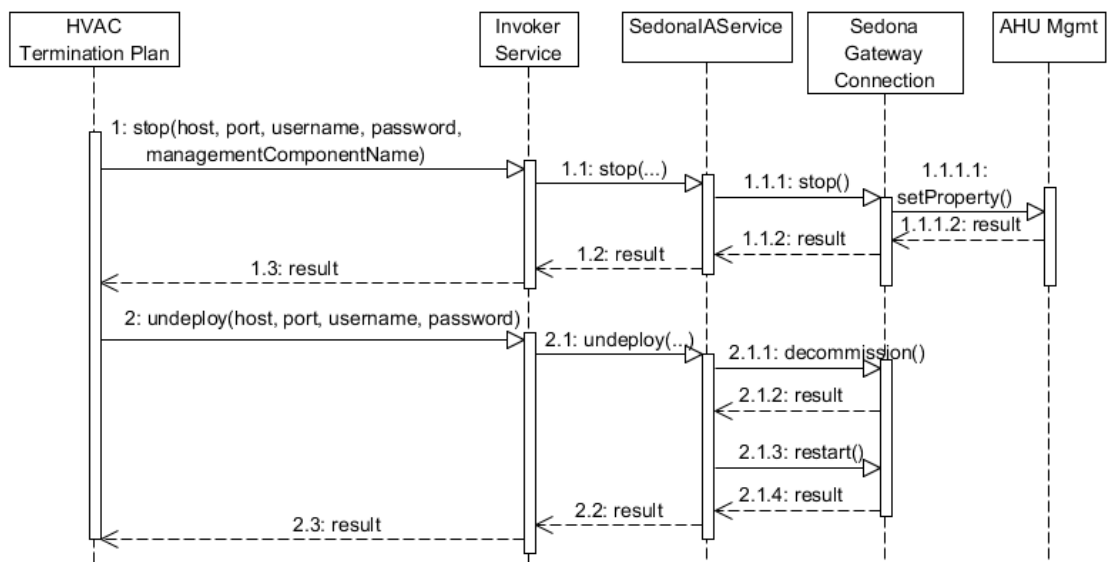


Figure 6.14: Sedona Undeployment Procedure of Application Termination Plan.

6.6.3 Management Plans

The management plans (cf. Section 5.4.3) to start, stop and configure AHU Controllers are described in this section.

Air Handler Configuration

The management procedure to change the setpoint temperature of an AHU Controller is illustrated in Figure 6.15. First, the ChangeSetpoint Plan queries (1) for the actual `NodeType` of the node instance that is referenced by the `controllerNodeInstanceUri`. Then, either the properties for the Sedona (2) or the Niagara (4) node instance are retrieved from the Instance Management. The properties are used to invoke the “changeSetpoint” operation via the Invoker Service. In case of the SedonaAHUController (3), the SedonaIAService sets the setpoint property of the Sedona AHU Mgmt interface component to the new value. The new value is immediately propagated to the AHU Controller’s control logic attached to it. In case of a Niagara node type, the NiagaraIAService invokes the start operation (5) of the Niagara AHU Management interface component to change the configuration of the Niagara AHU Controller’s control logic. Finally, the state of the AHUController node instance is set to “Running” (6).

Start & Stop Controller

The management procedure to start a Controller is illustrated in Figure 6.16. This figure and the explanations apply to the stop Controller management procedure by substituting the “start” by “stop” operations. First, the Start Controller Plan queries (1) for the actual `NodeType` of the node instance that is referenced by the `controllerNodeInstanceUri`. Then, either the properties for the Sedona (2) or the Niagara (4) node instance are retrieved from the Instance Management. The properties are used to invoke the “start” operation via the Invoker Service. In case of the SedonaAHUController (3), the SedonaIAService starts the Controller via the Sedona AHU Mgmt component whereas the NiagaraIAService invokes the start operation (5) of the Niagara AHU Management component in case of a NiagaraAHUController. Finally, the state of the AHUController node instance is set to “Running” (6). The StopController Plan would change the state to “Stopped”.

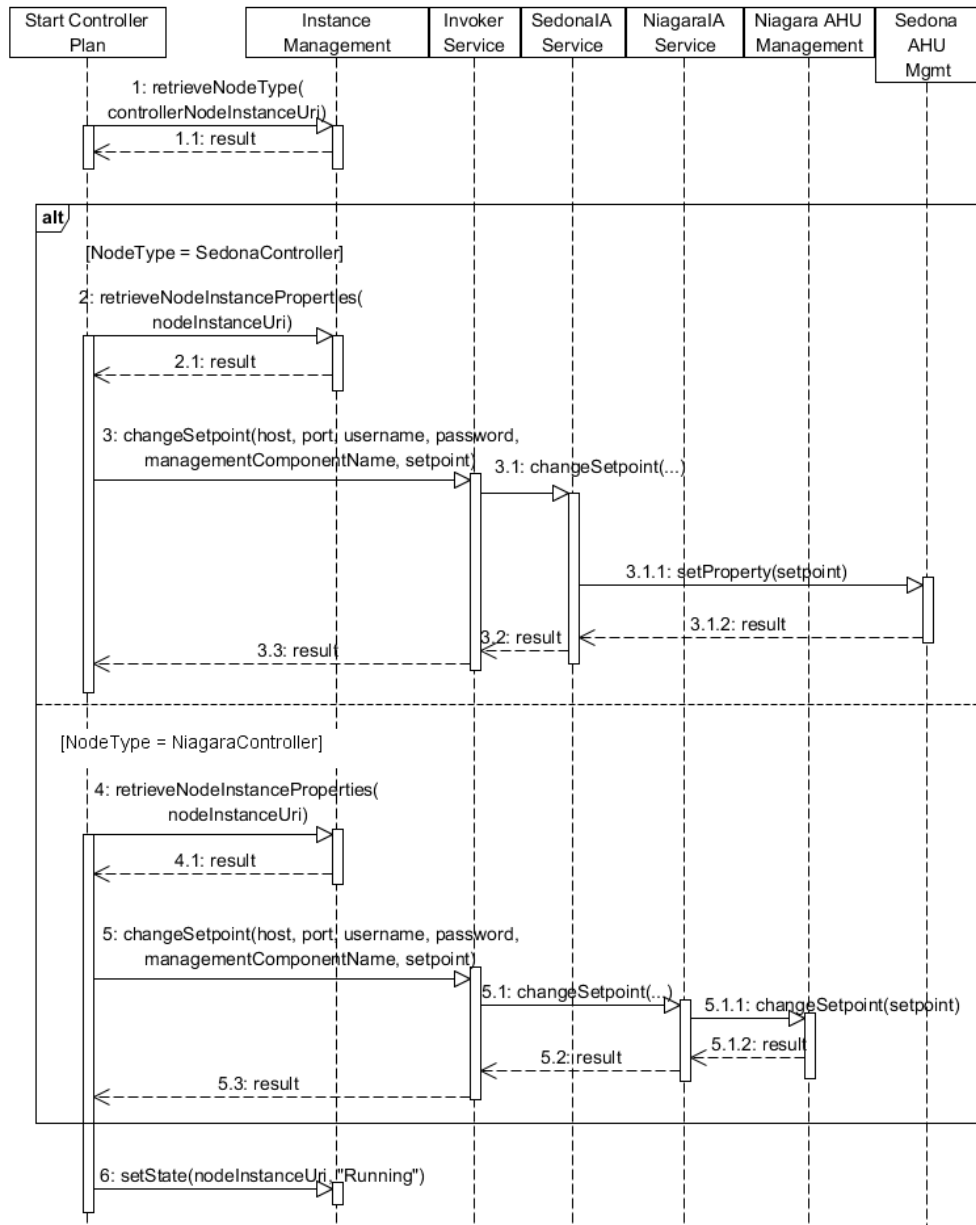


Figure 6.15: Management Plan to Change Setpoint of AHU Controller.

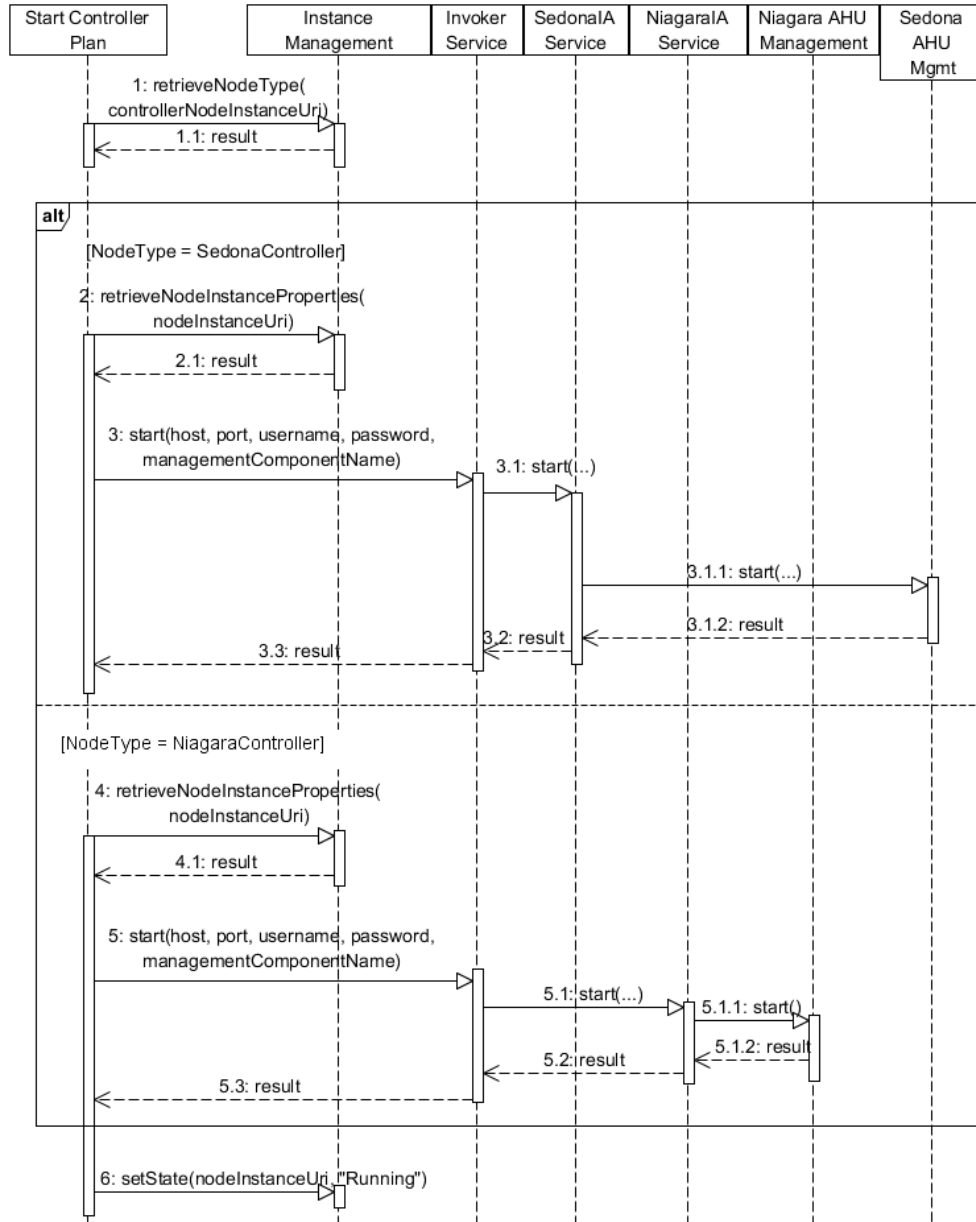


Figure 6.16: Start Controller Management Plan.

Deployment & Demonstration

7.1 Use Case Setup

The concrete setup for the use case described in Chapter 4 consists of a Niagara and a Sedona gateway. A M2M JACE¹ (NPM-2) depicted in Figure 7.1 running the Niagara^{AX} 3.7.106 framework [30,32] (cf. Section 3.3.1) represents the Niagara gateway. It is connected to the local area network via Ethernet using a static IP address (128.131.172.101).



Figure 7.1: Niagara M2M JACE (NPM-2)²

The *Platform Daemon* runs on default port 3011. The Niagara Station (cf. Section 3.3.1) that will be deployed on the JACE gateway is running the Obix Network driver to provide access to the Obix API via `http://128.131.172.101/obix/`. A temperature sensor, to sense

¹https://www.tridium europe.com/storage/downloads/TridiumEuropeDatasheet_m2mjace_1355839010.pdf

²<http://one-sightsolutions.com/wp-content/uploads/2013/04/M2M-JACE-300x200.jpg>

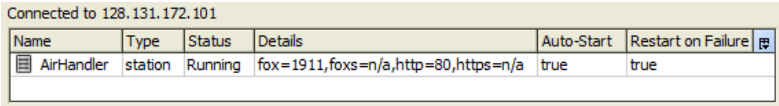
the room temperature, is connected to the universal input of the gateway and accessed via the Niagara Remote Input Output (NRIO³) driver network.

A Sedona VM 1.2.28⁴ is running on the local workstation in platform mode – started with the `svm -plat` command – representing the Sedona gateway (cf. Section 3.3.2). A basic user interface for runtime diagnostics can be accessed via `http://localhost:8099/`. The Sox service is running on default port 1876.

The *Niagara Workbench* is used to connect to the gateways and examine their status. The proprietary Sedona Framework TXS 1.2.100.1⁵ plugin is installed on the Niagara Workbench to connect to the Sedona gateway.

7.2 Application Deployment & Management

The prototype (HVAC.csar file discussed in Section 6.1) was deployed on the OpenTOSCA Container by uploading it to the OpenTOSCA Admin UI (`http://localhost:8080/admin/`). In a second step, the HVAC application was selected via the OpenTOSCA Self-Service UI (`http://localhost:8080/vinothek/`). There, the life-cycle management procedure to deploy the HVAC application on the gateway environments was initiated by selecting and invoking the corresponding build plan interface (cf. Listing 5.1). The deployment plan then instantiated the topology model of the application and configured and started the AHU Controllers as discussed in detail in Section 6.6.1. The successful deployment of the two AHU Controller instances (implemented by a Niagara and a Sedona control application) was verified by monitoring the gateways through the Niagara Workbench tool. The process copied the new Niagara Station to the Niagara gateway. After a reboot of the gateway, the Niagara Station was started and became accessible via the Niagara Workbench (cf. Figure 7.2). The structure of the



Connected to 128.131.172.101

Name	Type	Status	Details	Auto-Start	Restart on Failure	
AirHandler	station	Running	fox=1911,foxs=n/a,http=80,https=n/a	true	true	

Figure 7.2: Application Director view of M2M JACE Platform showing the “AirHandler” Station with Status “Running”.

Niagara Station deployed on the Niagara platform is shown in Figure 7.3. The “Config” node of the “AirHandler” Station defines platform services and drivers as well as the *AHUMangement* component (cf. Section 6.4.2), which implements the management interface of the “AirHandler” control application. The wire-sheet view of the Niagara AHU control application is depicted in Figure 7.4. It shows the Niagara AHU control application that is connected to the temperature sensor (“Outside” component) and the AHUMangement interface (“SetPoint” and “HvacProgram” components).

³<http://www.victordistcontrols.com/wp-content/uploads/2012/02/Niagara-AX-NRIO-Guide.pdf>

⁴<http://sedonadev.org/download/build/>

⁵http://www.tridium.com/galleries/datasheet_pdf/Sed-TXS-FINAL.pdf

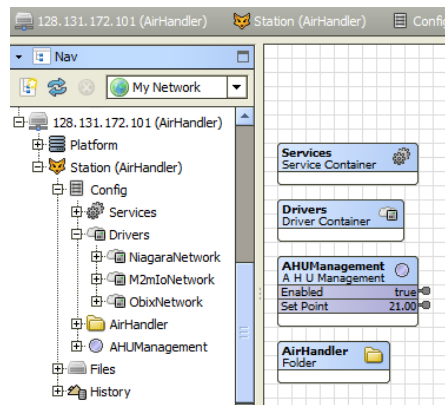


Figure 7.3: Structure of the Niagara “AirHandler” Station including the “AHUManagement” interface and the “AirHandler” control application.

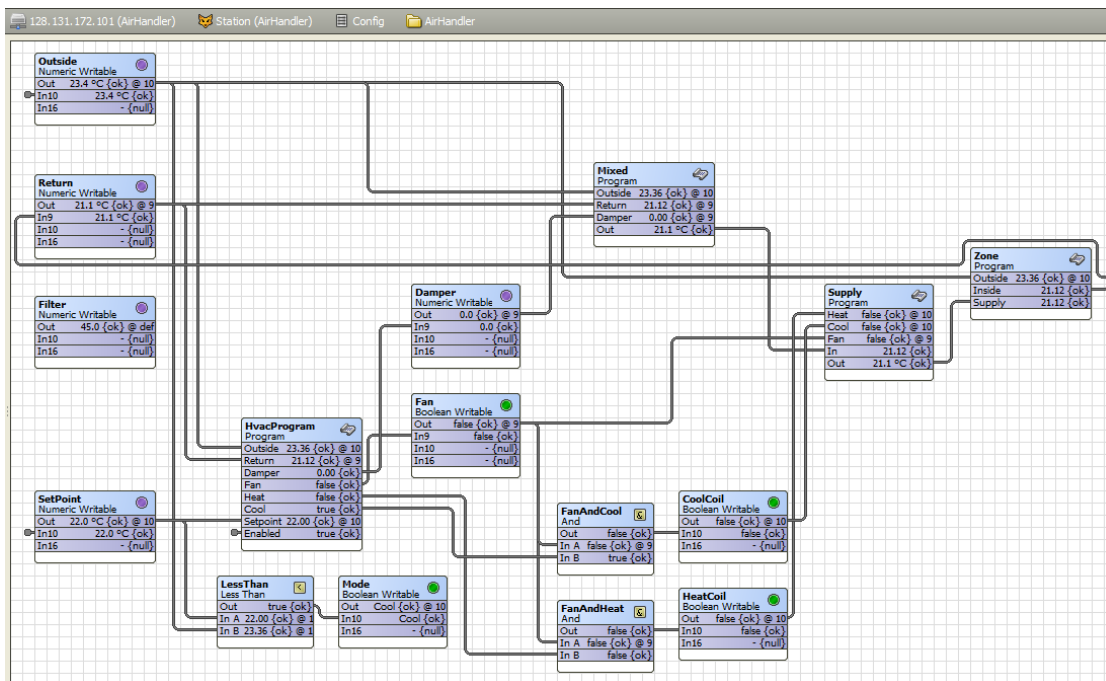


Figure 7.4: Wire-sheet view of the Niagara AHU Control Application.

At the same time, the Sedona application files were copied to the Sedona gateway before rebooting it. The Sedona application then was accessible via the Sedona plugin of the Niagara Workbench tool. Figure 7.5 shows the structure of the Sedona application deployed on the Sedona VM. The “App” node defines the Sedona platform services as well as the *AHUMgmt* component (cf. Section 6.4.1) which implements the management interface of the AHU control application. The wire-sheet view of the “AHUImpl” control logic, which is connected to the “AHUMgmt” management interface (“AHU” component), is depicted in Figure 7.6.

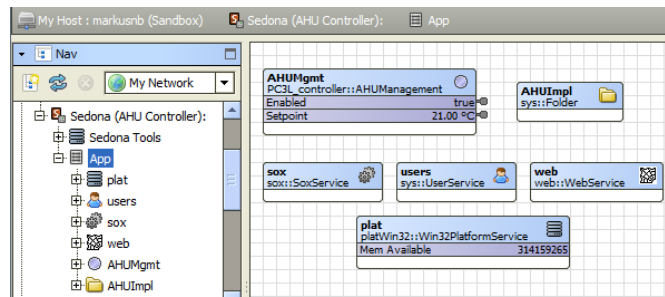


Figure 7.5: Structure of the Sedona “AHU Controller” application including the “AHUMgmt” interface and the “AHUImpl” control application.

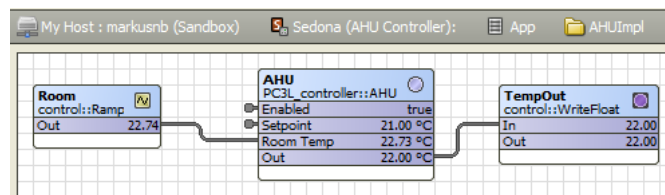


Figure 7.6: Wire-sheet view of the Sedona AHU Control Application connected to the AHUMgmt interface (AHU).

The output parameters of the deployment plan contain the Uri of the service instance (<http://localhost:1337/containerapi/instancedata/serviceInstances/1>), as well as the Uris of the node instances, which point to the instance management REST API of the OpenTOSCA container. The Uris of the instances of the *NiagaraAHUController* node template,

<http://localhost:1337/containerapi/instancedata/nodeInstances/2>

the *NiagaraGateway* node template,

<http://localhost:1337/containerapi/instancedata/nodeInstances/4>

the *SedonaAHUController* node template,

<http://localhost:1337/containerapi/instancedata/nodeInstances/3>

and of the *SedonaGateway* node template are returned.

<http://localhost:1337/containerapi/instancedata/nodeInstances/1>

These Uris consecutively facilitate the invocation of life-cycle management procedures on the node instances. SoapUI⁶ was used to demonstrate the invocation of the management procedures. For example, the *ChangeSetpoint Management Plan* (<http://localhost:9763/services/ChangeSetpointPlanService/>) was invoked with the SOAP message depicted in Listing 7.1.

⁶<http://www.soapui.org/>

Listing 7.1: SOAP message to invoke ChangeSetpoint management procedure.

```
1 <soapenv:Envelope
2   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3   xmlns:buil="http://www.opentosca.org/examples/HVAC/BuildPlan">
4   <soapenv:Header/>
5   <soapenv:Body>
6     <buil:ChangeSetPointPlanRequest>
7       <buil:csarName>HVAC.csar</buil:csarName>
8       <buil:controllerNodeInstanceUri>
9         http://.../containerapi/instancedata/nodeInstances/2
10      </buil:controllerNodeInstanceUri>
11      <buil:gatewayNodeInstanceUri>
12        http://.../containerapi/instancedata/nodeInstances/4
13      </buil:gatewayNodeInstanceUri>
14      <buil:setpoint>21</buil:setpoint>
15    </buil:ChangeSetPointPlanRequest>
16  </soapenv:Body>
17 </soapenv:Envelope>
```

The `ChangeSetPointPlanRequest` (Lines 6-15) contains the name of the CSAR file (7) as well as the Uris of the *NiagaraAHUController* node instance (9) and the *NiagaraGateway* node instance (12) where the Controller is deployed on. The ChangeSetpoint Management Plan then changes the `setpoint` temperature (14) of the Niagara specific AHU Controller as explained in Section 6.6.3. The management procedures to start and stop the Controllers were tested the same way. Again, the Niagara Workbench tool was used to monitor the effects the management procedures had on the control components. Finally, the service instance was terminated by invoking the Termination Plan interface (cf. Listing 5.2) resulting in an undeployment of the control applications as explained in Section 6.6.2.

7.3 Result

With the prototype implemented throughout this thesis we showed that by applying TOSCA to the IoT domain we can automate the deployment and life-cycle management of applications in heterogeneous IoT environments.

As intended by TOSCA, we defined the structure of the HVAC application in a topology template (cf. Section 6.3.1) composed of reusable components. The reusable components defined (cf. Section 6.2.1) and implemented (cf. Section 6.2.2) as node type definitions pose the basic building blocks for the HVAC application. They capture Niagara and Sedona framework specific management knowledge, which can be reused by any other application that builds on these frameworks. The high-level life-cycle management procedures (cf. Section 5.4) were implemented using BPEL workflows (cf. Section 6.6) which orchestrate the management operations provided by node types. The HVAC application is portable to any TOSCA environment which supports the BPEL workflow language for Plans and Implementation Artifacts of type “WAR”.

7.4 Possible Integration with IoT PaaS

The knowledge gained throughout the implementation of the prototype can be applied to the IoT PaaS architecture (cf. Section 3.1) to enhance the management of IoT applications.

The concept of providing control applications on IoT PaaS, explained in Figure 3.2 is revised in the following way. IoT resources like control applications can be registered to the *IoT resource management* as TOSCA node template definitions (Step 1 of Figure 3.2). The node templates define their management operations via the referenced node types (cf. Section 6.2). Solution providers subscribe to the control applications they want to use in their virtual vertical solution (Step 2.1+2.2). They define the application context (Step 3) by adjusting the parameters of the node templates (cf. Section 6.3). During solution deployment the IoT resource management monitors the availability of IoT resources (Step 4.2) through the gateway interfaces defined by TOSCA node type definitions. The application context management then instantiates (Step 4.1) the node templates, which represent the selected control applications.

Conclusion & Future Work

8.1 Conclusion

In the face of a rapidly growing number of IoT devices installed in today's facility management solutions, the automation of application deployment and life-cycle management is of vital importance. So far, the heterogeneous nature of the IoT domain together with the lack of standardized management procedures have led to physically isolated IoT solutions requiring high development and maintenance efforts.

In the course of this thesis we investigated how TOSCA can be applied to the IoT domain to automate application deployment and life-cycle management in a portable and reusable way. We extended the TOSCA node and relationship models with IoT specific node and relationship types to model the components typically occurring in IoT solutions. By defining and implementing the life-cycle interfaces of Niagara and Sedona framework specific node types, we showed that the specific knowledge required to manage IoT resources can be defined in a portable and reusable way. The topology of the prototypical IoT application we implemented throughout this thesis is composed of these reusable component definitions. The automation of application deployment and management was achieved by implementing high-level management procedures as workflows, which orchestrate the management operations provided by the life-cycle interfaces of node types. Finally we demonstrated that by deploying a self-contained application package to a TOSCA runtime environment, the life-cycle of IoT applications can be managed without understanding the internal details of the targeted gateway environments. Furthermore we discussed how the concepts implemented by this prototype can be applied to the IoT PaaS architecture we proposed in previous work. In summary, we showed that by applying TOSCA to the IoT domain, the IoT application deployment and life-cycle management can be automated.

8.2 Future Work

The prototype we have implemented proved that the deployment of IoT applications can be automated by OpenTOSCA. Further research will focus on improving and extending the concepts proposed in this thesis.

- In this thesis we used a TOSCA node model where a control application is represented by one single TOSCA node (cf. Section 5.2). In a next step, the hierarchical node model will be adjusted to describe application components at a finer grained level (cf. Section 5.3). This will allow the definition of application topologies (cf. Section 5.5) which describe the components and dependencies of control application. This modification leads to the ability of managing several controller instances on the same gateway.
- As a next step, we will integrate the prototype into the IoT PaaS architecture (cf. Section 3.1) to improve the automated management of IoT resources to enhance IoT solution delivery.
- In a dynamic environment like the IoT domain, changes to the topology of an application instance are common. Since the modification of an instantiated topology model is out of the scope of the TOSCA specification, possible solutions to deal with this requirement need to be investigated.

Bibliography

- [1] Tridium Inc., “The Web of Things.” <http://sedonadev.org/whitepapers/web-of-things.pdf>, 2009.
- [2] W. Kastner, G. Neugschwandtner, S. Soucek, and H. M. Newman, “Communication Systems for Building Automation and Control,” in *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1178–1203, 2005.
- [3] W. Granzer and W. Kastner, “Information modeling in heterogeneous Building Automation Systems,” *2012 9th IEEE International Workshop on Factory Communication Systems*, pp. 291–300, May 2012.
- [4] T. Samad and B. Frank, “Leveraging the Web: A Universal Framework for Building Automation,” *Proceedings of the 2007 American Control Conference*, pp. 4382–4387, 2007.
- [5] Tridium Inc., “Sedona Framework.” <http://sedonadev.org/>, Accessed: 15.10.2013.
- [6] M. Jung, J. Weidinger, C. Reinisch, W. Kastner, C. Crettaz, A. Olivieri, and Y. Bocchi, “A Transparent IPv6 Multi-protocol Gateway to Integrate Building Automation Systems in the Internet of Things,” *2012 IEEE International Conference on Green Computing and Communications (GreenCom)*, pp. 225–233, Nov. 2012.
- [7] OASIS Standard, “Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0.” <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>, 2013.
- [8] OASIS Committee Note Draft 01, “Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0.” <http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html>, 2013.
- [9] F. Li, M. Vögler, M. Claeßens, and S. Dustdar, “Towards Automated IoT Application Deployment by a Cloud-based Approach,” in *Proceedings of 2013 6th IEEE International Conference on Service Oriented Computing and Applications, SOCA 2013. Kauai, Hawaii*, 2013.

- [10] F. Li, M. Vögler, M. Claeßens, and S. Dustdar, “Efficient and scalable IoT service delivery on Cloud,” in *6th IEEE International Conference on Cloud Computing, (Cloud 2013), Industrial Track*, (Santa Clara, CA, USA), 2013.
- [11] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications,” in *11th International Conference on Service-Oriented Computing*, LNCS, Springer, 2013.
- [12] A. Chazalet and P. Lalanda, “A Meta-Model Approach for the Deployment of Services-oriented Applications,” *Services Computing, 2007. SCC 2007.*, no. Scc, 2007.
- [13] A. Chazalet and P. Lalanda, “Deployment of Services-Oriented Applications Integrating Physical and IT Systems,” *21st International Conference on Advanced Networking and Applications (AINA '07)*, pp. 38–45, May 2007.
- [14] T. Frenken, P. Spiess, and J. Anke, “A Flexible and Extensible Architecture for Device-Level Service Deployment,” *Towards a Service-Based Internet*, pp. 230–241, 2008.
- [15] D. R. A. De Groot, F. M. T. Brazier, and B. J. Overeinder, “Cross-Platform Generative Agent Migration,” 2004.
- [16] F. M. T. Brazier, B. J. Overeinder, M. Van Steen, and N. J. E. Wijngaards, “Agent factory: Generative migration of mobile agents in heterogeneous environments,” in *Proceedings of the AIMS Workshop at SAC 2002*, pp. 101–106, 2002.
- [17] OASIS, “Open Building Information Exchange (oBIX) Version 1.1.” <https://www.oasis-open.org/committees/download.php/38212/oBIX-1-1-spec-wd06.pdf>, 2010.
- [18] M. Jung, J. Weidinger, W. Kastner, and A. Olivieri, “Building Automation and Smart Cities: An Integration Approach Based on a Service-Oriented Architecture,” in *2013 27th International Conference on Advanced Information Networking and Applications Workshops*, pp. 1361–1367, IEEE, Mar. 2013.
- [19] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, “sMAP: A Simple Measurement and Actuation Profile for Physical Information,” in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems - SenSys '10*, SenSys '10, (New York, NY, USA), pp. 197–210, ACM, 2010.
- [20] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler, “BOSS: Building Operating System Services,” in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, nsdi'13*, (Berkeley, CA, USA), pp. 443–458, USENIX Association, 2013.
- [21] A. Krioukov, G. Fierro, N. Kitaev, and D. Culler, “Building Application Stack (BAS),” in *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, (New York, NY, USA), pp. 72–79, ACM, 2012.

- [22] J. Wettinger, M. Behrendt, T. Binz, U. Breitenbücher, G. Breiter, F. Leymann, S. Moser, I. Schwertle, and T. Spatzier, “Integrating configuration management with model-driven cloud management based on toasca,” in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, pp. 437–446, SciTePress, 2013.
- [23] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, and A. Weiß, “Improve Resource-Sharing through Functionality-Preserving Merge of Cloud Application Topologies,” in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, pp. 0–8, SciTePress, May 2013.
- [24] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann, “Pattern-based runtime management of composite cloud applications,” in *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, 8-10 May 2013, Aachen, Germany*, SciTePress, 2013.
- [25] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio, “Interacting with the SOA-Based Internet of Things: Discovery, Query, Selection, and On-Demand Provisioning of Web Services,” *IEEE Transactions on Services Computing*, vol. 3, pp. 223–235, July 2010.
- [26] American Society of Heating Refrigerating and Air-Conditioning Engineers Inc., “BACnet/WS - Addendum c to ANSI/ASHRAE Standard 135-2004.” <http://www.bacnet.org/Addenda/Add-2004-135c.pdf>, 2006.
- [27] T. Considine, “Work Plan for oBIX 2.0.” <http://www.automatedbuildings.com/news/apr13/columns/130330111101considine.html>, Accessed: 15.10.2013.
- [28] IETF Internet Draft, “Constrained Application Protocol (CoAP) Draft-ietf-core-coap-17.” <https://datatracker.ietf.org/doc/draft-ietf-core-coap/>, Accessed: 15.10.2013.
- [29] IETF Internet Draft, “Observing Resources in CoAP draft-ietf-core-observe-08.” <http://tools.ietf.org/html/draft-ietf-core-observe-08>, Accessed: 15.10.2013.
- [30] Tridium Engineering, “NiagaraAX-3.7 Developer Guide,” 2012.
- [31] A. Fernbach, W. Granzer, and W. Kastner, “Interoperability at the Management Level of Building Automation Systems A Case Study for BACnet and OPC UA,” *IEEE 16th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp. 1–8, Sept. 2011.
- [32] Tridium Engineering, “NiagaraAX-3.7 User Guide,” 2012.
- [33] Tridium Engineering, “Java Application Control Engine (JACE).” http://www.tridium.com/cs/products/_/_services/jace, Accessed: 15.10.2013.

- [34] L. Adcock, “My Adventures in NiagaraAX Home Automation - Lighting.” <http://www.niagara-central.com/ord?portal:/blog/BlogEntry/261>, Accessed: 15.10.2013.
- [35] Tridium Inc., “Sedona Framework Documentation.” <http://sedonadev.org/doc/index.html>, Accessed: 15.10.2013.
- [36] Tridium Inc., “Sedona Framework Architecture.” <http://sedonadev.org/doc/architecture.html>, Accessed: 15.10.2013.
- [37] Tridium Inc., “Sox Protocol.” <http://sedonadev.org/doc/sox.html>, Accessed: 15.10.2013.
- [38] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, *Advanced Web Services*, ch. TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. New York: Springer, January 2014.
- [39] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, “Portable Cloud Services Using TOSCA,” *IEEE Internet Computing*, 2012.
- [40] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, “TOSCA and OpenTOSCA: TOSCA Introduction and OpenTOSCA Ecosystem Overview.” <http://de.slideshare.net/OpenTOSCA/tosca-and-opentosca-tosca-introduction-and-opentosca-ecosystem-overview>, 2013.
- [41] Tridium Inc., “Niagara Framework.” http://www.niagaraax.com/cs/products/niagara_framework, Accessed: 15.10.2013.