FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic Interoperability Layer for oBIX

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Igor Pelesić
Matrikelnummer 0006828

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Projektass. Dipl.-Ing. Bakk.techn. Andreas Fernbach

Wien, 10. Jänner 2017

Igor Pelesić	Wolfgang Kastner

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Semantic Interoperability Layer for oBIX

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Igor Pelesić
Registration Number 0006828

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Projektass. Dipl.-Ing. Bakk.techn. Andreas Fernbach

Vienna, 10th January, 2017

Igor Pelesić	Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

Igor Pelesić
Braungasse 27/3, 1170 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.

Wien, 10. Jänner 2017

Igor Pelesić

Danksagung

Zuallererst möchte ich mich ganz herzlich bei meiner Ehefrau Karin für ihre großartige Unterstützung und den grenzenlosen Optimismus bedanken. Ich möchte mich auch beim Wolfgang Kastner bedanken, der mir die Gelegenheit geboten hat, diese Arbeit in der Automation Systems Group zu schreiben und der immer beratend zur Seite stand. Andreas Fernbach gebührt ganz großer Dank von meiner Seite, da die Diskussionen mit ihm und seine Ratschläge einen ganz großen Einfluss auf diese Arbeit hatten und ganz wesentlich zur Verbesserung derselben beigetragen haben.
Ich möchte auch meinen Eltern, Mira und Aladin, für die Geduld und Unterstützung danken, die ich Zeit meines Lebens von ihrer Seite erfahren habe. Ich möchte mich auch bei meiner Schwester Ida bedanken, die immer für mich da war und auch für das Korrekturlesen dieser Arbeit.

Acknowledgements

Foremost, I would like to thank my beloved wife Karin for her great support and the inspiring motivation that accompanied me during the evolution of this thesis. I would also like to express my gratitude to Wolfgang Kastner, who gave me the opportunity to write this thesis at the Automation Systems Group and supported me with his advice throughout this work. Likewise, I want to give a big thanks to Andreas Fernbach for the great help he offered me. These advices and discussions have had a great impact on this thesis and majorly contributed to the improvement of it.
I want also to thank my parents, Mira and Aladin, for encountering me with patience and support throughout my whole life. Furthermore, I want to thank my sister Ida for her steady encouragement and for reviewing this document.

Kurzfassung

Die Integration verschiedener Building Automation Systems (BAS) Technologien ermöglicht es, die Gebäudeenergieeffizienz zu steigern und gleichzeitig den Bewohnerkomfort zu erhöhen. Web Services (WS) erlauben einen technologieunabhänigen Datenaustausch und gelten als vielversprechendes Vehikel um die Integration heterogener Systeme in das Internet of Things (IoT) voranzutreiben. Das Zusammenspiel von BASs und Smart Grids eröffnet großes Potential, um die Energieeffizienz zu steigern. Um dieses Potential zu realisieren, sollte der Großteil der Energie dann konsumiert werden, wenn genügend erneuerbare Energie produziert wird. Dafür bedarf es aber einer bidirektionalen Verbindung zwischen den Abnehmern und den Stromerzeugern, die aber durch eine Vielzahl heterogener Geräte, Protokolle und Technologien erschwert wird. open Building Information eXchange (oBIX) ist ein offener Standard, der einen technologieunabhängigen Zugang zu BASs bietet. Die Semantik der ausgetauschten Daten ist dennoch nicht explizit formalisiert und ist Gegenstand weiterer Interpretationen. Eine vollumfassende Integration verschiedener BASs kann aber nur unter der Voraussetzung gelingen, dass die ausgetauschten Daten semantisch konsistent sind. Semantische Interoperabilität erlaubt den Datenaustausch eindeutig bestimmter Daten innerhalb des IoT und erleichtert die Kommunikation zwischen den Dingen(*things*). Das Ziel dieser Arbeit ist es einen semantischen Interoperabilitätslayer für oBIX zu definieren. Ausgehend von typischen Anwendungsfällen aus der Gebäude- und Heimautomatisierung soll eine Web Ontology Languagy (OWL) Ontologie erstellt werden, die den Austausch semantisch konsistenter Daten unterstützt. In einem weiteren Schritt sollen die ausgetauschten Daten semantisch, mittels des Vokabulars der Ontologie, angereichert werden, damit eine Transformation vom oBIX Modell zu OWL möglich wird. Hierfür ist der Einsatz von XSLT angedacht. Nicht nur statische Daten, sondern auch Laufzeitdaten, wie aktuelle Sensorenwerte, sollen während der Transformation berücksichtigt werden. Um die Evaluierung zu erleichtern, soll ein Prototyp im Zuge dieser Arbeit erstellt werden.

Abstract

The integration of different Building Automation System (BAS) technologies in cooperation with smart grids shows great potential for enhancing the energy efficiency of buildings and inhabitant comfort. In order to realize this potential, smart buildings should consume most of their energy when there is enough of renewable energy available and postpone or reduce their consumption otherwise. The communication between devices on the demand side the whole way up to power distributors is aggravated by a great diversification of BASs, which is caused by a variety of devices, protocols and technologies. Web Services (WS) provide a technology independent way of accessing data. Therefore, they represent a viable path to ease integration of various systems and technologies into an Internet of Things (IoT). The open Building Information eXchange (oBIX) is an open standard which offers a technology independent access to BASs and allows integration on a syntactical level by abstracting from different control protocols and network technologies. Still, the data exchanged does not necessarily bear the same semantics and thus requires further interpretation. In order to reach the goal of a comprehensive integration of different BAS technologies, this semantic incompatibility has to be resolved by enabling the distribution of semantically consistent data by various diversified devices. Semantic interoperability allows different interconnected agents to interpret the exchanged data without ambiguity and further empowers automated communication between *things*. The aim of this work is to provide a Semantic Interoperability Layer for the oBIX standard. Starting from a set of use cases describing typical home and building automation scenarios, an ontology shall be designed which will allow to exchange semantically consistent data across various BASs. The intention is to use the Web Ontology Languagy (OWL) for this task. An extension or integration of existing ontologies from the building automation sector should be considered. In a second step, the oBIX standard shall be enriched with meta-data annotations, whereby the semantic of these annotations should be taken from the created ontology in order to ensure consistent data across various devices. Further, a transformation from oBIX to OWL needs to be performed. Not only static data like e.g. the location of a sensor should be considered for the transformation but also runtime data, such as current sensor values. In order to ease the evaluation of the project, a working prototype shall be developed as part of this work.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The integration of different Building Automation System (BAS) technologies allows increasing both the energy efficiency of buildings and inhabitant comfort. Web Services (WS) provide a technology independent way of accessing and representing data and represent a promising approach to ease integration of heterogeneous systems and technologies [1] into an Internet of Things (IoT) [2]. The term IoT basically refers to a network of connected *things* such as sensors, embedded devices or phones. The fundamental objective of interconnecting *things* is to empower them to create a more precise model of their context thus allowing them to provide services which react intelligently to dynamic changes within their environment [3].

Buildings are responsible for about one third of the world wide overall energy consumption and thus represent the sector with the highest energy demand [4]. Building Automation System (BAS) in cooperation with smart grids, which are considered as one of the most important applications of the IoT [5], show great potential for enhancing the energy efficiency of buildings. In order to realize this potential, smart buildings should consume most of their energy when there is enough of renewable energy available and postpone or reduce consumption otherwise.

The intelligent bidirectional interaction from devices on the demand side the whole way up to power distributors is hindered by a great heterogeneity of BAS, which is caused by the various devices, protocols and technologies produced by different vendors [6].

The open Building Information eXchange (oBIX) [7] is an open standard which offers a technology independent access to BAS and eases integration on a syntactical level by abstracting from different control protocols.

Still, the data exchanged does not necessarily bear the same semantics and thus requires further interpretation. In order to reach the goal of a comprehensive integration of

different BAS technologies, this semantic incompatibility has to be resolved by enabling the distribution of semantically consistent data by various diversified devices.

Overcoming this limitation would make it possible to share data across a distributed BAS that is compatible in terms of semantic aspects and to implement a common knowledge base which in turn will enable the realization of more comprehensive automation scenarios that support the objective of improving the energy efficiency of buildings [8].

1.2 Problem statement

The interconnection of *things* based on WS can only be seen as an intermediate step towards the fulfillment of the IoT vision. On this level, heterogeneous devices are enabled to interact via a formally specified application protocol. However, reasoning about the meaning of the exchanged messages requires human interaction. In order to empower the collaboration of autonomous devices and agents, a common understanding between them has to be established. This requires the utilization of Semantic Web technologies.

Semantic interoperability allows different agents connected to the IoT to interpret the exchanged data unambiguously and furthermore empowers automated communication between *things*. Unambiguous data descriptions that are interpretable by machines and software agents are a major driver for automated information exchange within the IoT. SPARQL queries and semantic reasoning could be used to solve problems related to knowledge extraction, data abstraction and discovery of resources [3]. Thus, by achieving semantic interoperability, autonomous distributed devices and agents would be enabled to collectively answer questions respectively perform actions like:

- Is every switching actuator of a distinct floor in "off" state?
- How many rooms of a building are occupied?
- Which lamps in a building have exceeded a distinct operating time?
- Turn on the light in a specific room!
- Turn on all electrical boilers of a site having a distinct offset between current temperature and setpoint!

In order to provide a Semantic Interoperability Layer for oBIX (SILo), as a first step, a transformation of the oBIX object model to an OWL ontology is required. The generic transformation process has to consider the flexible nature of the oBIX object model and a possible gap in the level of abstraction between the oBIX model and the target OWL ontology. As the oBIX standard offers little implicit semantic meaning, a mechanism has to be designed which allows to enrich the oBIX model with further meta-data. It has to be investigated whether the transformation process can be fully automated.

Generally, it has to be verified whether a semantic interoperability layer allowing to monitor and control a BAS, based on oBIX is achievable. Therefore, a SPARQL interface is proposed which should allow to retrieve data from the knowledge base describing the actual status of the underlying BAS. Further, it should provide the means to change the state of the BAS by executing SPARQL update statements, thus providing the means to actively control a BAS. In order to be of use SILo has to provide a scalable solution which also takes the timing requirements of BASs into account.

1.3 Aim of the work

The aim of this work is to provide a Semantic Interoperability Layer for the oBIX standard. Starting from a set of use cases describing typical home and building automation scenarios, an ontology shall be designed which will allow to exchange semantically consistent data across various BASs. The intention is to use the Web Ontology Languagy (OWL) [9] for this task. An extension or integration of existing ontologies from the building automation sector should be considered.

In a second step, the oBIX standard shall be enriched with meta-data annotations, whereby the semantic of these annotations should be taken from the created ontology in order to allow consistent data across various devices.

Further, a transformation from oBIX to OWL needs to be performed. Not only static data like e.g. the location of a sensor should be considered for the transformation but also live data like current sensor values. It is envisaged to use an XSL Transformation (XSLT) for this task.

In order to ease the evaluation of the project, a working prototype shall be developed as part of this work. For the prototype, we will concentrate on a KNX [10] based BAS. The prototype shall support the engineer throughout the SILo transformation and will maintain a common knowledge base as a Resource Description Framework (RDF) [11] triple store. Therefore, a Semantic Web crawler will be developed which recurrently searches for devices and updates the RDF triple store with fresh data transformed from the oBIX interfaces of the devices.

The prototype should also provide means to query the data from the RDF triple store with SPARQL [12].

1.4 Methodological approach

The methodical approach will include the following steps:

Literature review
: A literature review will be performed in order to collect all the required information necessary for the theoretical part of the work.

OWL ontology
Existing ontologies from the home and building automation area will be investigated and searched for reusable components. Based on a set of typical automation scenario use cases, an OWL ontology shall be created.

Semantically enriched oBIX
An already existing mapping of KNX to oBIX [13] will be enhanced by semantic meta-data annotations.

oBIX to OWL transformation
An XSL Transformation will be developed which will allow the mapping of semantically enhanced oBIX data to RDF triples in XML format.

Semantic Web crawler
A Web crawler will be developed that autonomously gathers all the RDF triples from a BAS and stores the data in an RDF triple store.

SPARQL interface
An interface will be developed that allows querying the RDF triple store via SPARQL.

Evaluation
The implemented prototype will be evaluated against the basic use case set. During the assessment it should be verified that SPARQL queries are returning the correct and desired information.

1.5 Structure of the work

This section provides a short overview of the structure of this work. In Chapter 2, required information on the building blocks of SILo is presented. A short excerpt of related works in these fields is provided as a description of the IoT, which is currently an extensively discussed topic in the research community. Additionally, the semantic evolution of the IoT is presented, together with a short introduction to smart grids. In the oBIX section, an overview of the oBIX specification is provided, offering details about its object model and network bindings. The KNX section describes the control network technology that is used for the prototype implementation in this work, providing a short introduction on the new KNX Web Service specification draft. Besides, a summary of the Semantic Web and its technologies is delivered as well. Furthermore, this chapter provides a short introduction to the ThinkHome ontology and is concluded with a presentation of IoTSys, the oBIX server implementation used for the proof-of-concept implementation.

In Chapter 3, the Semantic Interoperability Layer for oBIX (SILo) is presented. It contains a detailed description of the transformation process required to map an oBIX model to an arbitrary OWL ontology. Additionally, it describes the binding of SILo to oBIX used for the data synchronization. In Chapter 4, the prototype implementation based on IoTSys

and the ThinkHome energy-resource OWL ontology is described. Chapter 5 presents the evaluation results from the test lab, where the usability of the proposed solution is discussed. The thesis is finalized with a conclusion and a description of possible further research tasks in Chapter 6.

CHAPTER 2

State of the art

2.1 Related work

The IoTSys [13] developed at Vienna University of Technology realizes the IoT concept within the sphere of BAS. It provides basic means to interconnect various different building automation technologies by applying various protocol bindings between field level protocols and management level WS protocols such as oBIX and OPC UA [14]. The data provided by WS protocols may not necessarily be semantically consistent as, e.g., the units used may differ.

The integration of BAS into the Semantic Web has been in focus of research as indicated by a number of ontologies from this area, such as ThinkHome [15], SSN-XG [16] or SensorML [17]. The goal of these ontologies is simply put to provide a generic vocabulary which eases the interpretation of data provided by BASs. This allows more complex scenarios to be implemented.

The SPITFIRE [18] project also defines an own OWL ontology. It already contains the idea of a Semantic Web crawler collecting the data from all the sensors currently present and feeding the sensor values to an RDF triple store. Additionally, an approach is presented which allows to recognize the type of newly deployed sensors, by statistically comparing their data with already present sensors.

2.2 Internet of Things

The term Internet of Things (IoT) basically refers to a network of connected *things* such as sensors, consumer electronics or smart phones. A concise definition as "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" is provided in [19]. Thus, the Web paradigm is extended to include the *things* of everyday life and features of Internet applications are usable for Machine-to-Machine

(M2M) and Human-to-Machine (H2M) interactions [20]. The fundamental objective of interconnecting *things* is to empower them to create a more precise model of their context allowing them to provide services which react intelligently to dynamic changes within their environment [3].

The huge amount of interconnected devices and the data they expose uncloses a set of new application scenarios currently not available and simultaneously imposes a considerable amount of challenges to be solved. New applications related to smart environments (home, office, plant), transportation and logistics, healthcare, the energy sector (smart grids) will emerge provided that the participating *things* are connected to the IoT. In order to unlock the full potential of the IoT, several challenges related to the huge amount of data involved, the heterogeneity of the connected *things*, privacy and security issues still need to be solved [2]. Searching for specific data and discovery of services in the IoT represents one of the major challenges [3]. Semantic technologies could provide a solution to problems arising due to the expected high amount of data that are related to storage, searching, interconnection and organization of data [2]. Semantic technologies in the sphere of the IoT have generally received a lot of interest in the research community [21] [17] [22] [23].

Semantic interoperability would allow different agents connected to the IoT to interpret the exchanged data unambiguously and further would empower automated communication between *things*. Unambiguous data descriptions which are interpretable by machines and software agents are a major driver for automated information exchange within the IoT. SPARQL queries and semantic reasoning could be used for solving problems related to knowledge extraction, data abstraction and discovery of resources [3].

A proposal utilizing semantic technologies is the evolution of the IoT to the Semantic Web of Things (SWoT) as shown in Figure 2.1 [24]. According to that, the IoT is a first step where all the *things* should be connected to a global network and the interoperability between heterogeneous devices should be enabled. In a second step, all *things* should be connected via the same application protocol, preferably HTTP, to the global Web of Things. In the third step, all *things* are integrated into the SWoT where a common understanding between all objects is established by utilizing Semantic Web standards. It is recommended to use ontologies to create a common semantic reference in order to enable M2M interactions, thus allowing to provide services with higher context awareness, knowledge and reusability [24].

Still, there are some challenges when it comes to semantic technologies in the domain of the IoT. There is a need for standardized ontologies that are widely adopted within the IoT. Semantic annotations allow to create machine-readable and interpretable data. Nevertheless, as this data is not necessarily machine-understandable there is a need for advanced methods to create useful abstractions. Additionally, lightweight semantic technologies that can be executed on devices with limited resources are of importance [3].

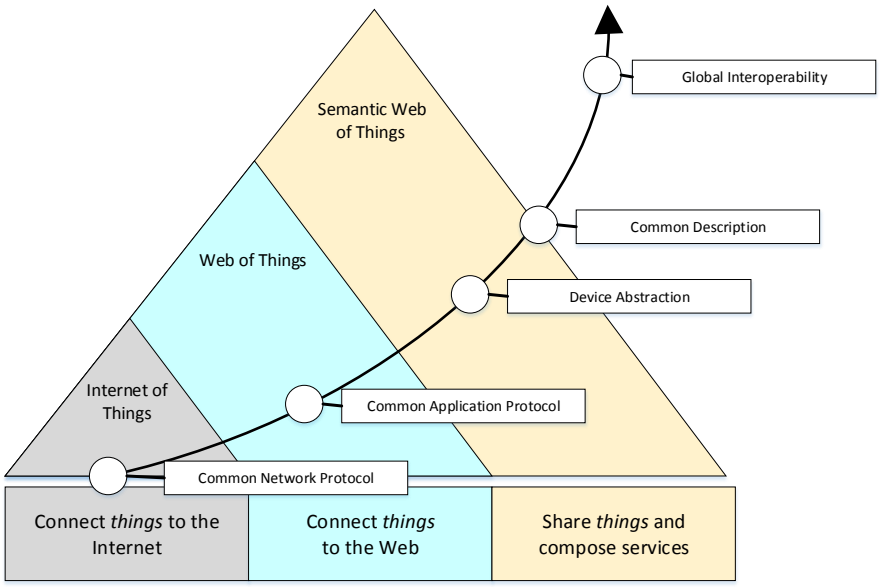

Figure 2.1: Semantic Web of Things (adopted from [24])

2.2.1 Smart Grid

The smart grid is considered as one of the most important applications of the IoT [5]. The core element of the smart grid is a bidirectional communication between energy producers and energy consumers. It is expected that smart grids will change the paradigm of energy distribution from a centrally controlled system to a system where the control is distributed to various objects. This should allow a more precise monitoring and controlling of the system which should result in less energy loss and higher efficiency [25].

The current approach of energy producers to adapt the production to a given energy demand is about to be reversed to adjustment of the demand according to the current production possibilities. Provided that a precise metering of the energy consumption over time is possible, energy producers could introduce time-dependent tariffs in order to animate the consumers to flatten their peak demand. It is not enough to publish the tariffs as due to the increased amount of renewable energy that is injected into the power grid the tariffs might change in timely manner and therefore a real-time communication between consumers and operators is required [5].

The combination of IoT technologies with the energy domain is denoted as Internet of Energy (IoE) [26]. Achieving the goal of higher energy efficiency is only possible

by combining users, operators, and intermediaries into such an IoE while complying with the climate protection goals. For this to be successful, some obstacles still have to be overcome, such as defining missing standards related to the semantics of the data exchanged [27].

2.3 Open Building Information Exchange

The open Building Information eXchange (oBIX) specification is released and maintained by the Organization for the Advancement of Structured Information Standards (OASIS). The purpose of oBIX [7] is to empower communication between heterogeneous devices by abstracting from their actual hardware and low-level protocols they use. It provides a common and standardized interface that models a device to a set of datapoints allowing to access them via Web Services (WS). A similar WS based approach is also possible via OPC UA [14] and BACnet/WS [28].

Every accessible oBIX object is identified by its URI [29]. The according information is exchanged in XML or JSON format over HTTP, which makes the information available to every Web browser. Essentially oBIX provides the means to enable and improve the Machine-to-Machine (M2M) communication. oBIX complies very well to the Representational State Transfer (REST) paradigm which is a core concept in the IoT. A RESTful service is characterized through a set of principles such as resource-orientation, identification of resources, a uniform interface and stateless requests which are all essential to oBIX as well.

oBIX provides a central access object termed *lobby*, which provides a central access point to the oBIX server. It offers information about the supported encodings and bindings of the server, as well as additional information about the specific oBIX server implementation. In order to start a discovery of provided oBIX objects, a client basically just needs to connect to the well known *lobby* (http://server/obix) address.

Additionally, oBIX supports an *alarm* management mechanism to handle situations which require the propagation of an alarm state to a responsible agent. It also provides a logging mechanism to store the *history* of a datapoint which is essentially a timestamped list of datapoint values. A *batch* functionality allows to trigger the execution of multiple actions with a single request in order to reduce the network overhead. Handling of partial failures during a *batch* operation is left to the server implementations [7].

2.3.1 Object Model

oBIX provides a flexible and extendible object model which is depicted in Figure 2.2 [7]. The common base primitive of this model is the object abstraction *obix:obj*. An exemplary *obix:obj* implementation is shown in Listing 2.1.

Every further object type like *real, int, str* is an extension of the base object. Any object type can contain further objects, i.e. the concept of composition (*has-a* relationship) is

```
<obj href="/devices/light_switch/" is="obix:Point">
  <bool name="value" href="value" val="false" writable="true"/>
</obj>
```

Listing 2.1: *obix:obj* Example

property	description
name	name of the object
href	URI Reference to the object
is	Contracts that the object implement
null	assertion whether the object contains a value
val	value of an object
ts	tag of an object

Table 2.1: *obix:obj* properties

Type name	Payload
bool	true or false
int	integer value
real	floating point value
str	UNICODE string
enum	enumerated value within a fixed range
abstime	timestamp
reltime	duration or timespan
date	date as day, month, and year
time	time of day as hour, minutes, and seconds
uri	URI

Table 2.2: oBIX Value Types [7]

supported. The extendibility of the model is based on the concept of *contracts*, which act as templates for inheritance (*is-a* relationship). Contracts additionally define properties for each object type such as default values and attributes attached to it. Properties supported by all object types are *name, href, is, null, val, ts*. The meaning of these properties is explained in Table 2.1.

Further properties e.g. *min, max, displayName, unit, range* that provide meta-data about objects are named *facets*. The *unit* facet supports most relevant SI-units and the *range, min and max* facets limit the range of values which an object may store. The oBIX standard contains a set of core value objects, each of them storing a different value type. These are explained in Table 2.2 [7].

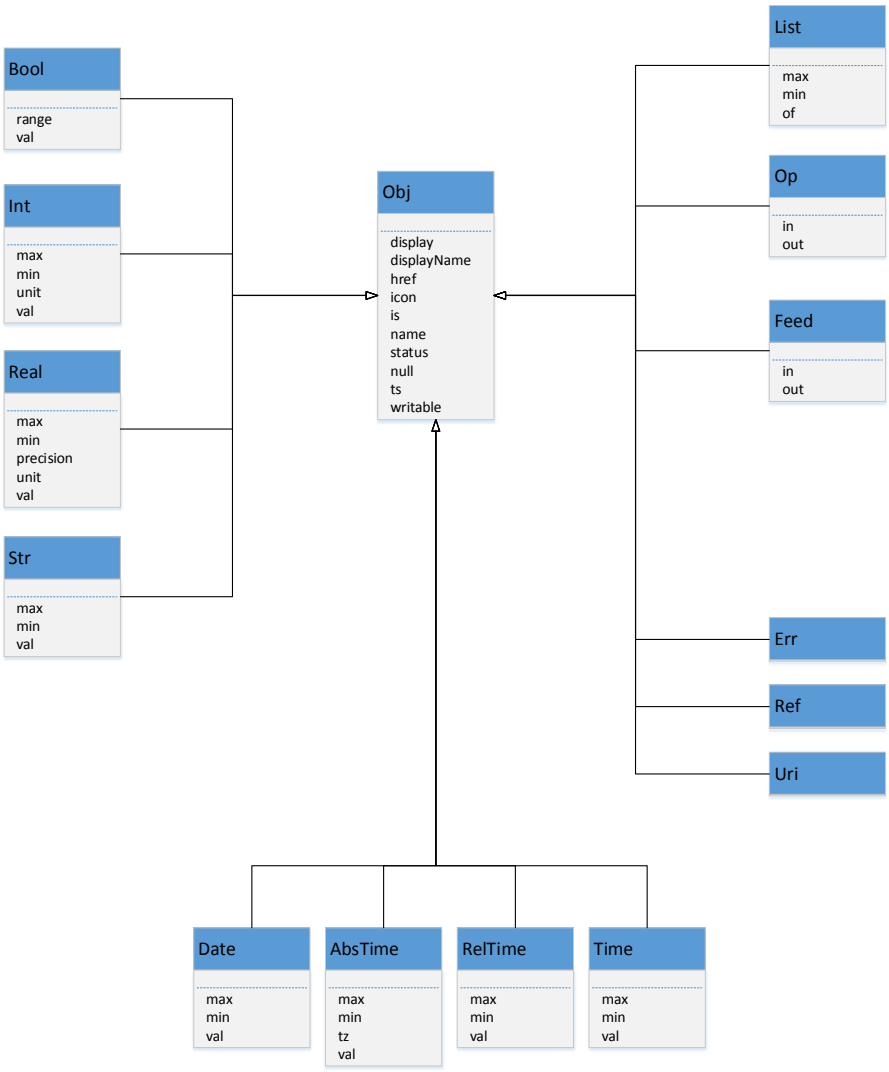

Figure 2.2: oBIX object model (adapted from [7])

2.3.2 Contracts

In order to group different object instances sharing common properties, oBIX introduces the concept of *contracts*, which are comparable to classes in object oriented languages. Contract definitions are templates which are expressed as simple oBIX objects and can be referenced by their URI using the *is* attribute. The contract definition for the base

```
1 <obj href="obix:obj" null="false" writable="false" status="ok" />
```

Listing 2.2: *obix:obj* Contract [7]

oBIX Request	HTTP Method
read	GET
write	PUT
invoke	POST
delete	DELETE

Table 2.3: oBIX HTTP mapping [30]

object *obix:obj* is shown in Listing 2.2 [7].

Contracts are proper for modelling inheritance relationships in oBIX. Even multiple-inheritance is supported. By defining a type, it is possible to assign default values to its instances and agree on the semantics of an object across different vendor systems e.g. an *obix:Alarm* instance has the same object structure on different vendor systems and implicitly provides information about an alarming condition. The object structure is explicitly defined by a contract, whereas the reasoning about its semantics usually demands human interaction.

The concept of contracts is simple and flexible and allows to introduce new abstractions without inserting new syntax elements to the standard.

2.3.3 Networking

The communication in oBIX conforms to a client/server paradigm, where a client sends service requests to a server that is handling these requests. The supported service requests are *Read, Write, Invoke and Delete*. There exist different protocol bindings for these atomic operations as REST [30], SOAP [31] and Websockets [32].

The mapping of oBIX service requests to HTTP request methods is shown in Table 2.3 according to [30].

Different encodings such as XML, JSON and EXI are provided for the oBIX standard.

In order to allow a client to keep track of real time information e.g. a temperature sensor value, the oBIX *watch* mechanism is presented. The client creates an *obix:Watch* object where it registers all the datapoints it is interested in and gets informed if any of these get updated. Using bindings where it is not possible to push events from server to client e.g. HTTP, the client has to continuously poll the server for updates, by sending a *pollChanges* request, whereas using the Websocket binding allows the server to directly push events to the client in case of changes on registered datapoints [7].

2.4 KNX

The KNX Association, founded in 1999, was the result of a merge between three different European associations active in the sphere of building automation, each of them providing an own BAS: Batibus, European Installation Bus (EIB) and the European Home System (EHS). The association provides an open standard [8] and offers certification services related to it [5].

The KNX architecture is decentralized which means that all nodes can directly communicate with each other without requiring a central control unit to coordinate the communication [5]. This structure allows for building reliable systems as there is no single point of failure.

KNX supports a set of different physical layers as Twisted Pair (KNX TP1), Powerline (KNX PL), Radio Frequency (KNX RF) and Ethernet (KNX IP). The access to KNX TP1 is regulated with a CSMA/CA access control [5]. KNX IP allows to access a KNX network via Ethernet and supports two different mechanisms KNXnet/IP routing and KNXnet/IP tunneling which allows for integrating KNX into IP based networks. The first one allows to send telegrams simultaneously and connectionless to multiple receivers via a KNXnet/IP router. KNXnet/IP tunneling allows to address a single receiver in a connection oriented manner [33].

The functionality of KNX devices is expressed by functional blocks (FB). FBs consist of inputs, outputs and parameters that are required to perform a given functionality e.g. sunblind actuator basic [5]. An example from [10] is shown in Figure 2.3. FBs are defined by a set of datapoints (DPs) specified by the KNX standard that provide access to the functionality of a block. DPs e.g. switch on/off are determined by the KNX specification in format and encoding of the according datapoint type (DPT) as range and unit. They are exposed either as group objects (GO) or as interface objects properties (IOP). GOs may be read or written over the network using a multicast transmission in order to exchange sensor or actuator data. IOPs are mostly used for management tasks like configuration and programming of devices and rely on unicast communication with individual addresses [34].

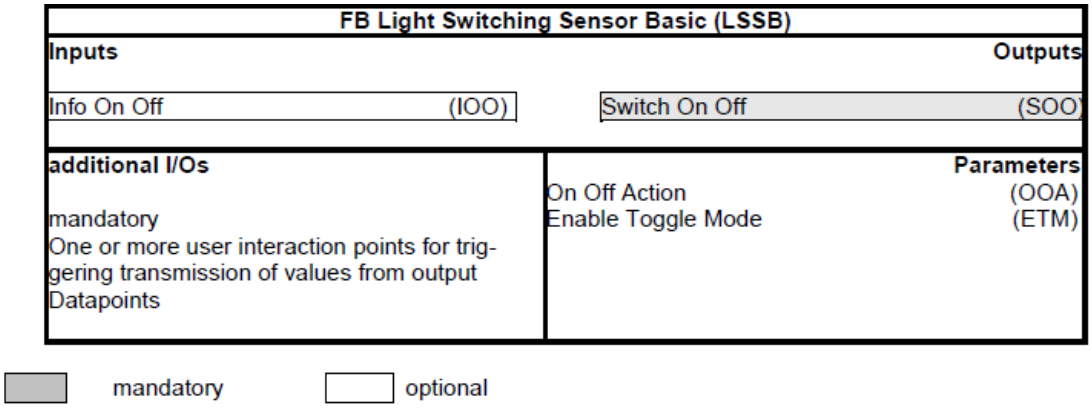

Figure 2.3: KNX function block [10]

2.4.1 KNX Web Services

The KNX specification is about to be extended with a KNX Web Service specification [35] which defines a standardized interface that allows to integrate KNX networks with other IT systems like e.g. the IoT. The interconnection between KNX devices and other IT systems is enabled by the introduction of a *KNX Gateway* as shown in Figure 2.4. The *KNX Web interface* of the *KNX Gateway* has to support either oBIX, OPC UA and BACnet/WS and allows Web clients to read and modify data within the KNX network. The *KNX Network access* interface is responsible for the communication to KNX devices. The *KNX information model* specifies the structure of the input model that is used to represent the *KNX Network* within the *KNX Gateway* [35].

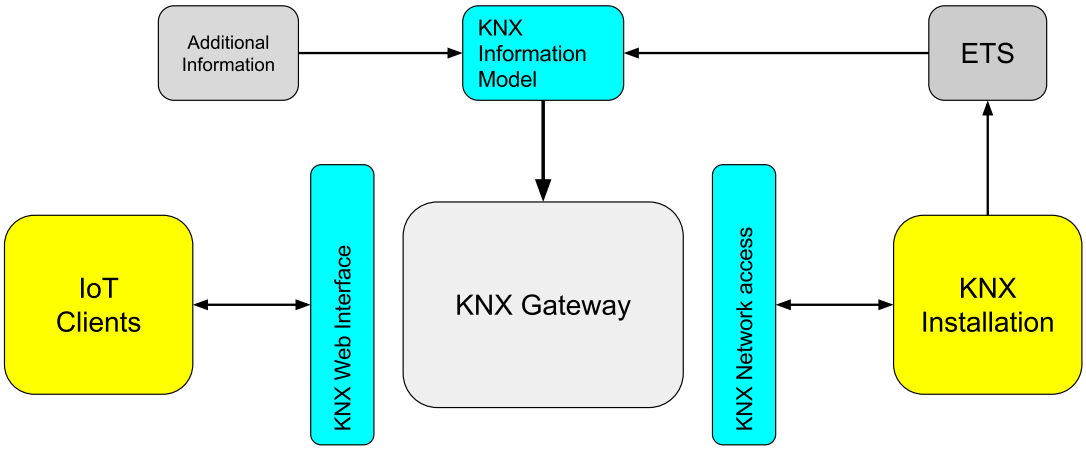

Figure 2.4: KNX gateway (adapted from [35])

The *KNX information model* is based on the *KNX Tag vocabulary* which specifies a set of tags and their relations to each other. It is an extension to the already existing ETS object model and determines the static structure of a KNX Network whereas the runtime values are accessed via the KNX Network interface. The presented model foresees additional sources of information as input to the *KNX information model* which could provide required information missing in the ETS e.g. semantic annotations. The *KNX information model* allows to specify entities by assigning them tag-value pairs as shown in Figure 2.5. Tags with a *null* are named *marker* tags and define a is-a relationship whereas the tags ending with '*Ref*' are reference tags pointing to other entities. The entities used to model the general KNX Network structure are shown in Figure 2.6 [35].

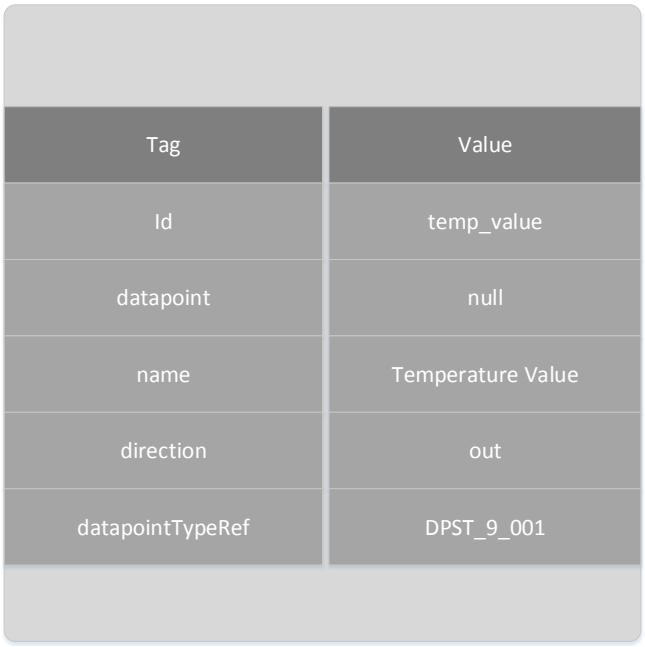

Figure 2.5: KNX model of a datapoint

The specification provides a mapping from the *KNX information model* to each of the supported WS technologies such as oBIX. An exemplary REST GET response from an oBIX based *KNX Gateway* is shown in Listing 2.3.

```
<obj name="example" href="/installation/example/" is="/knx/Installation" displayName="
    Example">
  <list name="views" href="views" of="obix:ref /knx/View">
    <ref name="view_heating" href="view_heating" is="/knx/View"/>
    <ref name="view_alarm" href="view_alarm" is="/knx/View"/>
  </list>
</obj>
```

Listing 2.3: KNX Gateway oBIX response

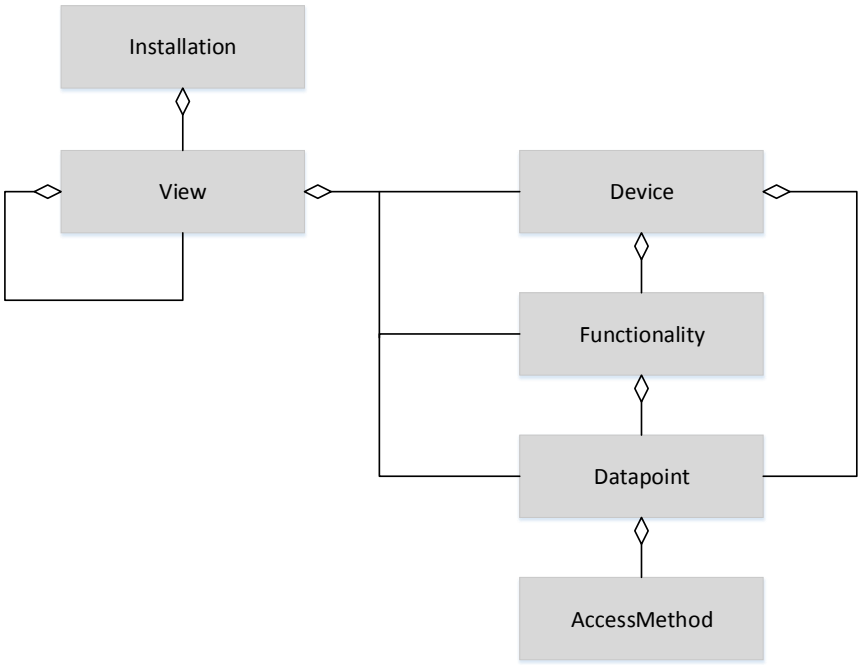

Figure 2.6: Abstract model of KNX network [35]

2.4.2 ETS

The Engineering Tool Software (ETS) is used to develop and deploy KNX installations. The software allows to model the building hierarchy, network topology and to create group objects in order to obtain the desired functionality [36]. It is also possible to name and describe the group objects with meaningful names. The ETS additionally provides the means to deploy an application on a given device.

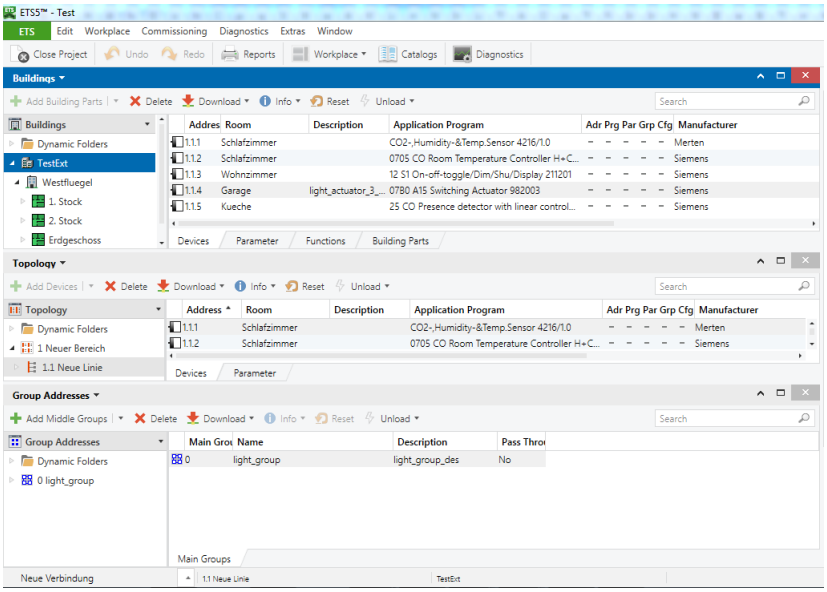

Figure 2.7: ETS user interface

The ETS provides an XML export functionality for projects. The files exported contain all DPs used in the project as well as DPTs, the building structure, the group objects. These data can be transformed to oBIX as shown in [36] or [37] or to OPC UA as shown in [38].

2.5 Semantic Web

The Semantic Web provides the means to categorize and classify items and to reason about the relationships between these items. The idea of the Semantic Web as presented by Berners-Lee in 2001 [39] is to extend the current Web, where most of the available data is designed for humans, and to improve the structure of the data by giving the provided information a well defined meaning empowering advanced H2M and M2M cooperation opportunities. This provided meaning in which items are logically structured and connected enables the interoperability of systems [40].

The following features of the Semantic Web are observable: anybody can say anything about any topic (*AAA slogan*), the provided information is never complete as new data can be added any time (*open world/closed world*) and that semantically equal things can be named differently in different contexts (*nonunique naming*) [41].

The World Wide Web Consortium (W3C) has developed a set of technologies and languages to share meaning across the Semantic Web. The usage of Internationalized Resource Identifiers (IRIs) [42] in order to uniquely identify items is essential to the Semantic Web. Basically, an IRI is an extension of a URI as it allows a wider range of UNICODE characters. As IRIs have a global scope, anyone can reference resources

```
1 <Thermostat_1> <isLocatedIn> <Living Room>
```

Listing 2.4: RDF statement

associated with them [40]. This allows to semantically reuse or extend existing concepts and to model relationships amongst them. In the beginnings of the Semantic Web, most of the proposed technologies were based on the Extensible Markup Language (XML) [43] which for itself does not provide any means to model semantics. In the meantime, a wider spectrum of serialization formats has been introduced.

The Resource Description Framework (RDF) [11] is a framework for expressing information about resources which can be anything like documents, people or abstract concepts. It is designed for scenarios where data should be processed by applications instead of displayed to humans. RDF allows us to make statements about resources. The statements or triples consist of a *subject, predicate and object*. An exemplary statement is shown in Listing 2.4. *Subjects* and *objects* are resources. The *predicate* or *property* models the relationship between the resources and is always directed from a *subject* to an *object*. Triples can be visualized as a directed graph as shown in Figure 2.8. They are usually stored in an RDF store. RDF supports a variety of serialization formats e.g. Turtle [44], N-3 [45].

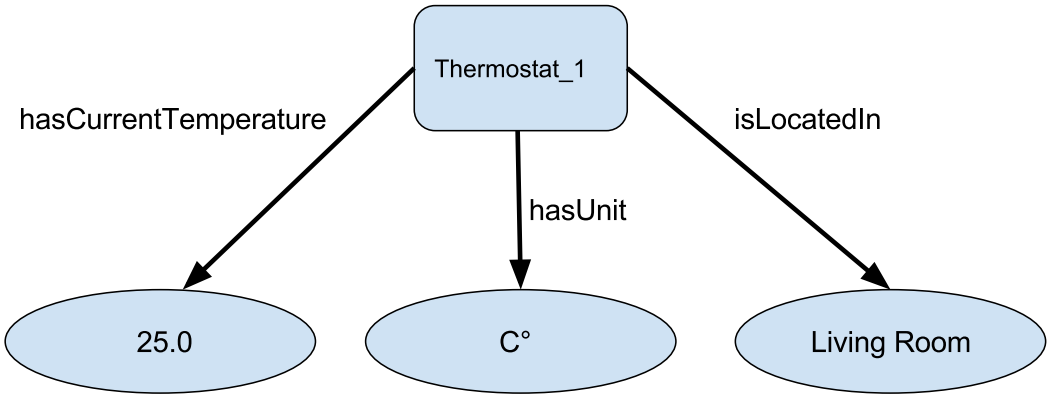

Figure 2.8: RDF graph visualization

RDF makes statements about resources, but it does not allow to make any statements about the nature of the resources or what they stand for. In order to enable the classification of resources, the RDF Schema [46] language was presented. The RDF Schema introduces the notion of a *class* expressed through the type *property*, which allows to build hierarchies of classes, subclasses, properties and subproperties. This concept is comparable to classes in object oriented programming languages. However, adding new properties to existing classes in object oriented programming languages means changing

```
<Thermostat_1> <type> <Thermostat>
<Thermostat> <subClassOf> <Device>
<Living_Room> <type> <Location>
<isLocatedIn> <type> <Property>
<isLocatedIn> <range> <Location>
<isLocatedIn> <domain> <Device>
```

Listing 2.5: RDF Schema triples

the class, whereas the RDF Schema allows to adapt an existing class to new requirements without changing the original class [47]. Restrictions on types are expressed through the *domain* and *range* properties. A simple RDF Schema class model is shown in Listing 2.5. The same hierarchy is visualized as graph in Figure 2.9.

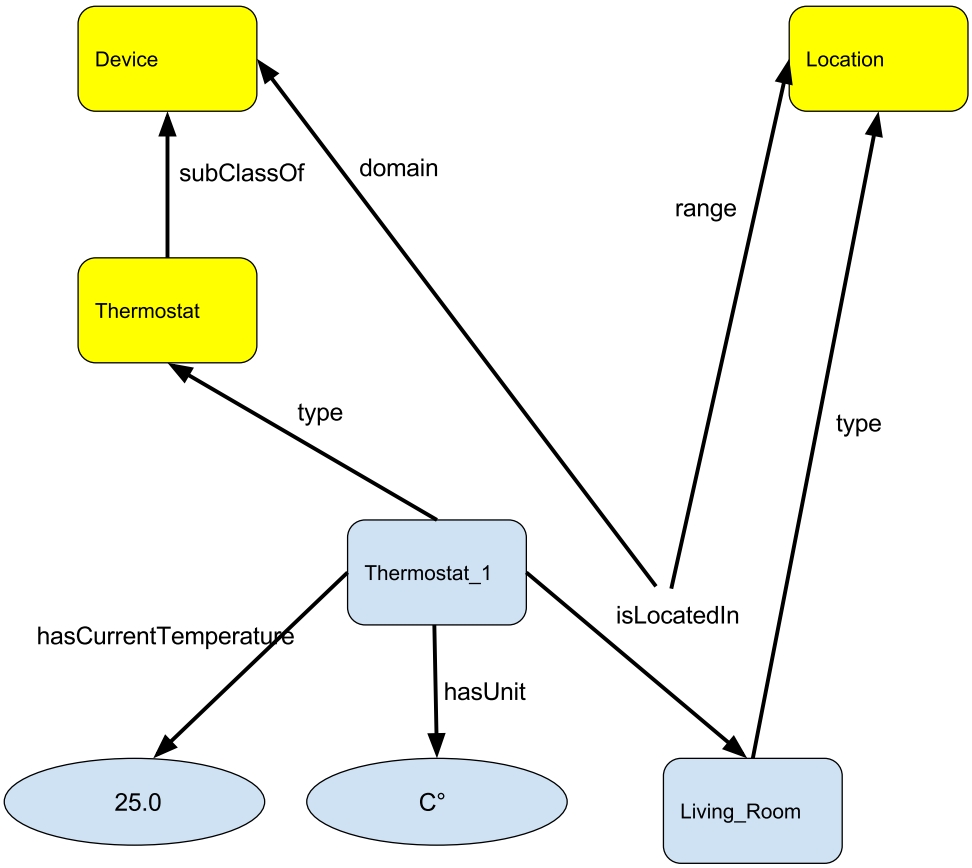

Figure 2.9: RDF Schema graph visualization

RDF Schema is a primitive ontology language [47]. Ontologies are collections of informa-

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 SELECT ?device
3 WHERE {
4       ?device isLocatedIn Living_Room .
5       ?device rdf:type Thermostat
6     }
```

Listing 2.6: Simple SPARQL query

tion which formally define relations among terms [39]. As both RDF and RDF Schema are rather restricted in their expressiveness, the Web Ontology Languagy (OWL) [9] was introduced. It is designed to model knowledge about things, groups of things and relations between them, which can be exploited by computer programs. OWL ontologies consist of classes, properties, individuals and data values. Classes define a concept while the properties model the relations between these concepts. Individuals are the instantiations of these concepts. The main exchange syntax for OWL is RDF/XML. Additionally to the features already supported by RDF Schema, it provides the means to express equality/inequality of classes, to specifically exclude membership from classes (disjoint classes), or to apply cardinality restrictions on properties. OWL distinguishes between datatype properties and object properties. The range of datatype properties is always a datatype such as string or date. Object properties always point to another resource object. The relation between RDF/RDF Schema and OWL is depicted in Figure 2.10 according to [47]. The hierarchy from Figure 2.9 after translation to OWL is shown in Listing 2.8.

An important aspect of the Semantic Web is the possibility to search for semantic data. Therefore, the W3C has provided the SPARQL Protocol and RDF Query Language (SPARQL) [12]. This allows to query and even to update RDF stores. SPARQL provides an HTTP and SOAP binding which allow to remotely execute SPARQL queries. In many aspects, SPARQL is comparable to the Structured Query Language (SQL) used with relational databases. The basic primitives of SPARQL queries are the *triple patterns* which are similar to normal RDF triples, but instead of the *subject, predicate* or *object* they contain a variable placeholder prefixed with the character '?'. Combining multiple *triple patterns* which all have to be satisfied is called a *group pattern*. Group patterns are enclosed by curly brackets [41]. The simple SPARQL query shown in Listing 2.6 searches for all thermostats located in the living room. As already mentioned, it is also possible to update RDF stores with a SPARQL Update [48]. Updates contain a delete and insert clause specifying which triples to delete and insert, respectively. The simple SPARQL update shown in Listing is moving a thermostat from the kitchen to the living room.

```
1
2  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
3  DELETE
4  {
5    ?device isLocatedIn Kitchen
6  }
7  INSERT
8  {
9    ?device isLocatedIn Living_Room
10 }
11 WHERE
12 {
13   ?device isLocatedIn Kitchen .
14   ?device rdf:type Thermostat
15 }
```

Listing 2.7: Simple SPARQL update

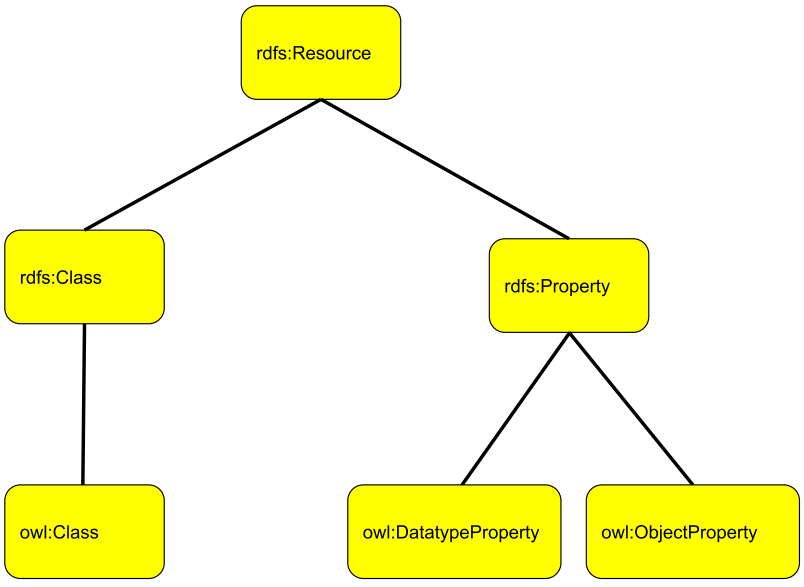

Figure 2.10: RDF/RDF Schema relation to OWL [47]

```
1
2  <owl:Class rdf:about="&testowl;Device">
3     <rdfs:label rdf:datatype="&xsd;string">Device</rdfs:label>
4  </owl:Class>
5
6  <owl:Class rdf:about="&testowl;Thermostat">
7    <rdfs:label rdf:datatype="&xsd;string">Thermostat</rdfs:label>
8    <rdfs:subClassOf rdf:resource="&testowl;Device"/>
9  </owl:Class>
10
11 <owl:Class rdf:about="&testowl;Location">
12   <rdfs:label rdf:datatype="&xsd;string">Location</rdfs:label>
13 </owl:Class>
14
15 <owl:NamedIndividual rdf:about="&testowl;Living_Room">
16   <rdf:type rdf:resource="&testowl;Location"/>
17 </owl:NamedIndividual>
18
19 <owl:ObjectProperty rdf:about="&testowl;isLocatedIn">
20   <rdfs:range rdf:resource="&testowl;Location"/>
21   <rdfs:domain rdf:resource="&testowl;Device"/>
22 </owl:ObjectProperty>
23
24 <owl:NamedIndividual rdf:about="&testowl;Thermostat_1">
25  <rdf:type rdf:resource="&testowl;Thermostat"/>
26  <isLocatedIn rdf:resource="&testowl;Living_Room"/>
27 </owl:NamedIndividual>
```

Listing 2.8: OWL ontology example

OWL comes with two sublanguages: OWL Full and OWL DL (descriptions logic). OWL Full is syntactically fully compatible to RDF Schema and is theoretically undecidable. OWL DL is syntactically restricted and thus less expressive, but due to its restrictions is decidable, which allows the implementation of reasoners e.g. Pellet [49] that are able to answer with *true* or *false*. Reasoners can be used to check if an ontology is logically consistent and further might be utilized to gain new knowledge.

2.5.1 ThinkHome

The major goal of the ThinkHome Smart Home System (SHS) [50] [15] is to increase the energy efficiency of smart homes while assuring a desired level of user comfort. The system consists of an extensive knowledge base formally specified as an OWL ontology and an intelligent multi-agent system controlling the smart home functions (cf. Figure 2.11). The agents are capable of making logical decisions on grounds of the knowledge sourcing from the knowledge base. They might also interact with other agents in order to perform complex control functions using the ontology as a common vocabulary.

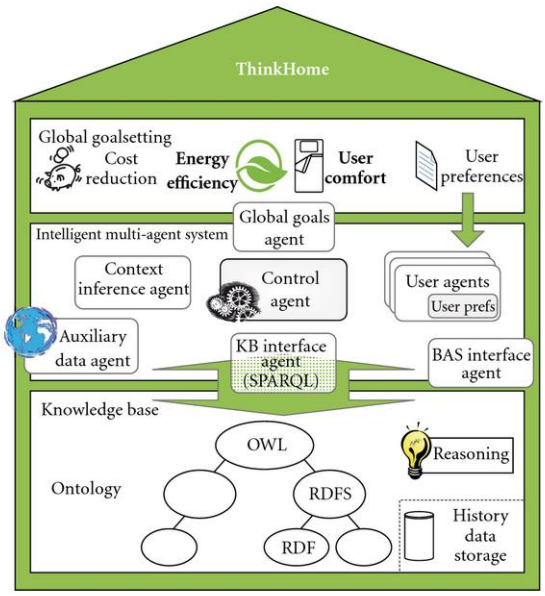

Figure 2.11: ThinkHome smart home system [15]

The ontology provided is composed of different modules each of them modelling another important aspect relevant to the energy efficient operation of residential buildings. Eventually the full ThinkHome ontology consists of five different ontologies as shown in Figure 2.12. The *building* ontology describes all relevant details related to the structure of the building which are important to allow an energy efficient control of the building. Therefore, the Green Building XML (gbXML) Schema [51] is integrated into the *building* ontology part as presented in [52]. The *user preferences* ontology allows to model expectations of users on the system such as a desired temperature at a given schedule. The *weather and exterior* part of the knowledge base provides the means to include exterior parameters as current weather conditions into the SHS. The set of features and processes which are available to the SHS e.g. heating, opening windows are expressed by the *processes* ontology. The *energy and resources* module provides information on the available low-level building automation technologies together with a model of energy related parameters such as energy tariffs and energy providers [50].

The energy and resource ontology contains models of devices commonly used in BASs such as temperature sensors, presence sensors or light switches. Furthermore, it allows to locate these devices within a building structure.

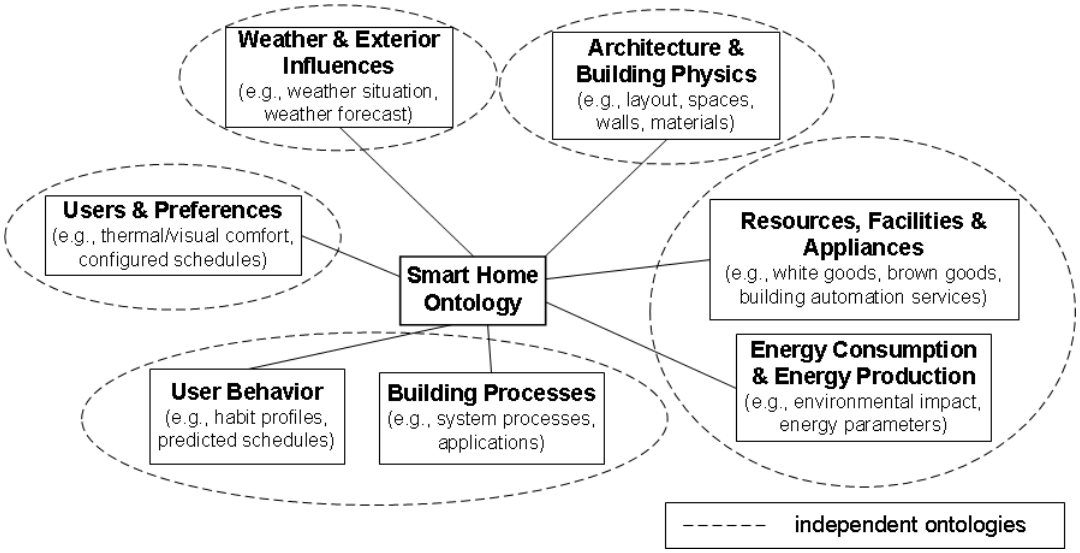

Figure 2.12: ThinkHome ontology [53]

2.6 IoTSys

IoTSys is an integration middleware for the IoT with focus on building automation technologies [13]. It is published as an open source [54] Java framework based on OSGi, consisting of various bundles allowing to specifically tailor the system to a given building automation scenario. The IoTSys architecture is depicted in Figure 2.13. It provides a transparent IPv6 based multi-protocol gateway interface for various building automation technologies such as KNX and BACnet, offering bindings to different data-link and physical layers. The protocol adapters like the KNX adapter are crucial elements of the architecture as they provide the binding to BAS specific protocols. The gateway interface maps different technologies to the oBIX object model and abstracts from the underlying BAS specific application protocol. The oBIX handler takes care of oBIX Read, Write and Invoke requests and is independent of the underlying BAS protocol. It provides access to the oBIX *watch* mechanism used to monitor changes of runtime data. The presentation of the oBIX *lobby* object is also in the scope of the oBIX handler [55].

Additionally to an HTTP binding to oBIX, IoTSys provides a Constrained Application Protocol (CoAP) [56] binding which supports asynchronous communication between a server and a client. Further, IoTSys supports the Efficient XML Interchange (EXI) encoding of the oBIX model. The EXI parser component is responsible for compression and decompression of the oBIX data between the oBIX handler and the CoAP/HTTP handler which offer a centralized oBIX compliant network interface [55].

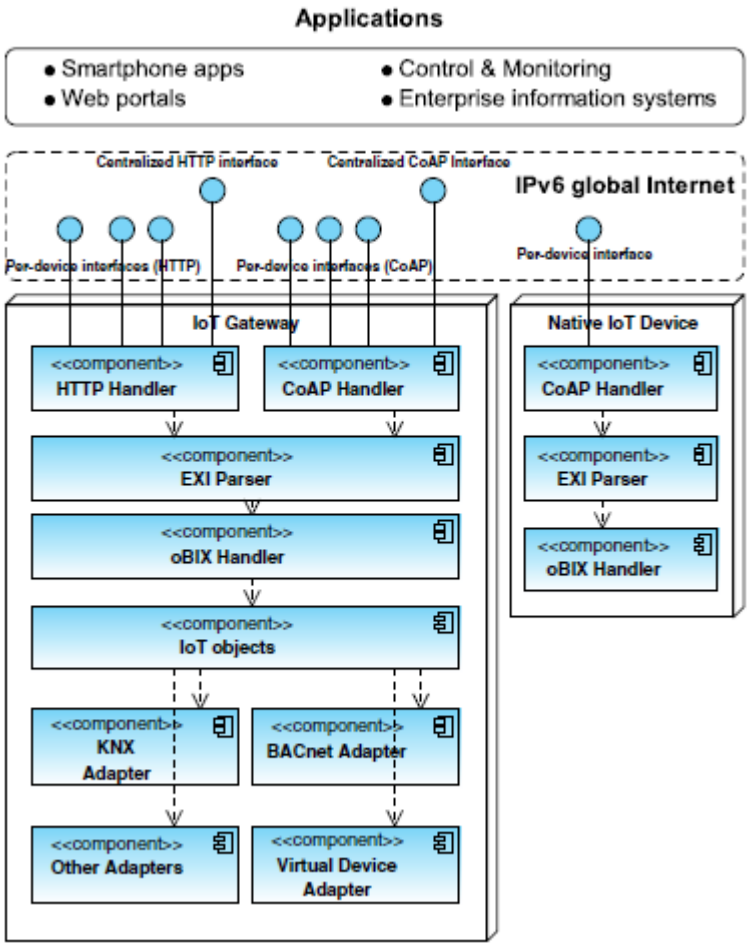

Figure 2.13: IoTSys architecture [55]

Due to the fact that reasoning about the semantics of oBIX contracts requires human interaction, as laid down in [55], it is proposed to enrich the oBIX data with semantic annotations, that allow a transformation to OWL ontologies targeting the building automation sector. As described in [13] [37], IoTSys supports the automated configuration of KNX projects by importing the ETS export file. This approach allows to use the ETS directly for the semantic annotation of oBIX objects. In the course of the mapping of KNX to oBIX, the KNX datapoint types are transformed to oBIX objects and the according datapoint type is preserved in the oBIX model as a contract.

CHAPTER 3

Semantic Interoperability Layer for oBIX (SILo)

In order to enable semantic interoperability that allows to exchange semantically consistent data unambiguously between different machines on basis of oBIX, a transformation of oBIX resources is required. The data presented at the oBIX interface is syntactically formalized but the meaning of the representation is not implicit. Semantic Web technologies provide the means to model the meaning implicitly. Therefore, a transformation of the oBIX representation to an OWL ontology is proposed. This ensures that the static structure of the BAS as provided at the oBIX interface is semantically well defined. This provides the means to use the resulting ontology as a common vocabulary for autonomous agents, to execute SPARQL queries and to gain new insights through semantic reasoning.

Nevertheless, the runtime data of BASs like sensor and actuator values require an additional consideration. The transformed ontology depicts the state of a BAS at the moment of the transformation. An energy efficient operation of BASs which satisfies the requirements related to comfort necessitates an access to runtime data. Therefore, a synchronization mechanism needs to be implemented which ensures that the data presented by the OWL ontology are always up-to-date.

The proposed Semantic Interoperability Layer for oBIX (SILo) as shown in Figure 3.1 is tailored to the oBIX REST binding [30] syntactically encoded in XML. It provides a SPARQL binding that allows to query and update the values of the BAS in control. The binding to oBIX is based on HTTP. It provides the means to read the actual data at the oBIX interface and further to write values of the BAS by exchanging REST calls between the oBIX server and the SILo implementation.

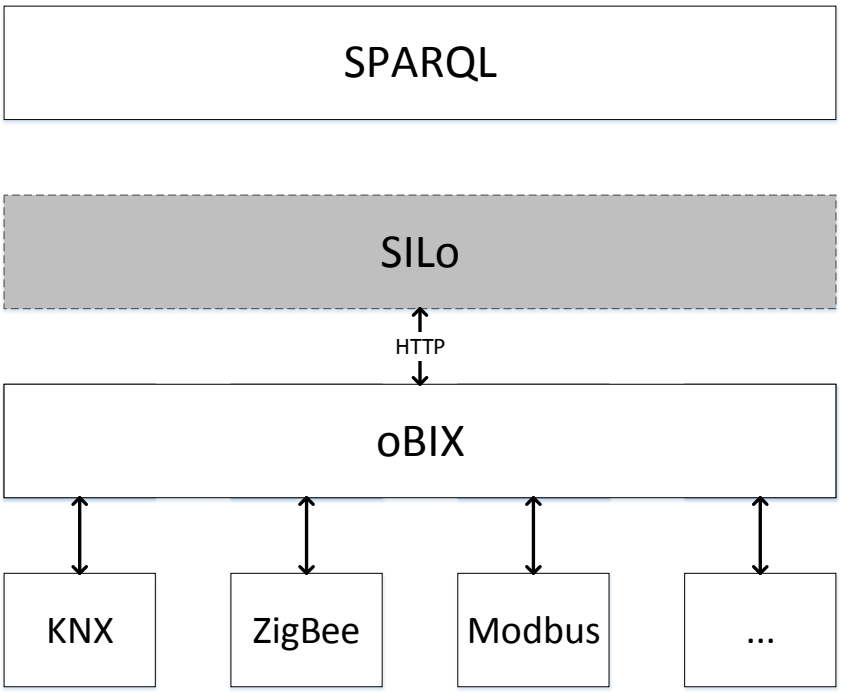

Figure 3.1: SILo stack

3.1 Model transformation

Herein, a general concept of the transformation from oBIX to an arbitrary OWL ontology, providing the means to model a building automation scenario, shall be introduced.

Following observations have an immediate impact on the transformation concept:

- oBIX provides a very **flexible object model** that allows to describe a BAS. Different implementations of oBIX servers might represent the same BAS differently both on the syntactic level and at a different level of abstraction.
- The objects are syntactically well defined by the standard, but the specification itself shows a **lack of inherent semantic information**.
- It is expected that in most cases the transformation will need to provide a **lift in the level of abstraction** i.e. the resulting ontology will provide a model of the BAS at a higher level of abstraction, thus providing a wider set of controlling features.

- An oBIX resource is **uniquely identified** by its URI. OWL classes, properties and individuals are uniquely identified by an IRI which is an extension of the former.
- As oBIX complies to the RESTful paradigm, information about various resources of the BAS might not be accessible via a single WS call but may require **multiple interactions** in order to gather all the relevant data related to a building automation scenario.

As both the oBIX source and the resulting ontology can be encoded in XML, the transformation is based on XSLT. Due to the flexible object model of oBIX, it can be concluded that an approach as presented in [57] including a mapping of the XML Schema Definition Language (XSD) [58] Schema to an ontology is not useful as it would allow only to model the formal syntactical model of oBIX. This concept does not provide the means to model server specific oBIX *contracts* which provide explicit semantic information extremely useful to the transformation. It further can be argued that every specific oBIX server implementation, as long as there is no standardized oBIX interface, will require a customized transformation implementation regardless of the used ontology.

As the oBIX standard does not provide the means to model semantic information implicitly, it is up to server developers to communicate such. Therefore, it can not be generally expected for such information to be provided by the oBIX server. If required semantic information is missing, a transformation is only possible provided that the required semantic information is obtained from a different source. Commonly, providing semantic information requires a human interaction and is considered as a necessary part of the transformation engineering process. This needs to be adapted to different oBIX server implementations. The higher the gap in the level of abstraction between the oBIX server implementation and the targeted ontology, the more semantic information needs to be provided in order to allow a transformation, i.e., if the oBIX server delivers information on the level of datapoints and the targeted ontology models the BAS at the same level, no or little additional semantic annotations will be required. On the other hand, if the ontology assigns datapoints to devices, an information will be required as to define which datapoints should be grouped together to model a device.

As a result of the REST paradigm, which is a core concept in oBIX and the supported reference feature, it might be necessary to traverse the complete or multiple parts of the oBIX object tree in the course of the transformation process in order to obtain all the required information. A specific WS call might return only the relevant data to this specific resource whereas information about its child objects might be provided as references. These child references have to be resolved by separate WS calls. Depending on the specific oBIX server implementation it might be required to traverse the whole oBIX object tree, starting from the oBIX lobby object, which acts as an entry point to the services provided by an oBIX server. This might result in an extensive oBIX document, which contains a lot of non-essential information and thus adds complexity to the XSL stylesheet implementation. In order to reduce the complexity of the transformation, it should be possible to specify a set of relevant oBIX object tree nodes which are traversed

```
1  <obj name="example" href="/installation/example/" is="/knx/Installation" displayName=
       "Example">
2    <list name="views" href="views" of="obix:ref /knx/View">
3      <ref name="view_devices" href="view_devices" is="/knx/View"/>
4    </list>
5  </obj>
```

Listing 3.1: oBIX REST response to GET request

and transformed independently and the resulting artefacts are combined to an OWL ontology representation.

On the grounds of the provided arguments, a generic transformation concept as depicted in Figure 3.2 for each of the relevant oBIX object nodes is proposed. According to that, the transformation is performed in an iterative three step process which is described in the following:

1. *Traverse and Combine* - In this step, the oBIX server object tree is traversed, starting from a specified node via multiple WS requests. Subsequently, the responses are combined into a *complete oBIX document*, which provides a complete view on the relevant parts of the oBIX representation of the BAS.

2. *Annotate* - This optional step allows to introduce semantic data annotations to the *complete oBIX document*. This can be omitted if the oBIX server of interest is already providing the required information. The result of this step is a *complete semantic oBIX document*. The semantic annotations should be attached to required resources via the oBIX contract mechanism.

3. *Transform* - In a final step, an XSL transformation is performed resulting in the desired OWL ontology. It is recommended to preserve oBIX URIs as unique identifiers for OWL individuals during XSL transformation.

Traverse and Combine

Starting from a defined start node of the oBIX object tree, identified by its URI, the results of single REST calls (GET) have to be combined into the *complete oBIX document*. If an oBIX object node contains a reference object (cf. Listing 3.1) this has to be replaced by its instantiation. This allows the implementation of a recursive algorithm which replaces all reference objects and returns the desired result as shown in Listing 3.2.

Annotate

The annotation step allows to mark relevant resources with semantical meta-data that eases the XSL transformation. This step can be omitted if the oBIX interface already

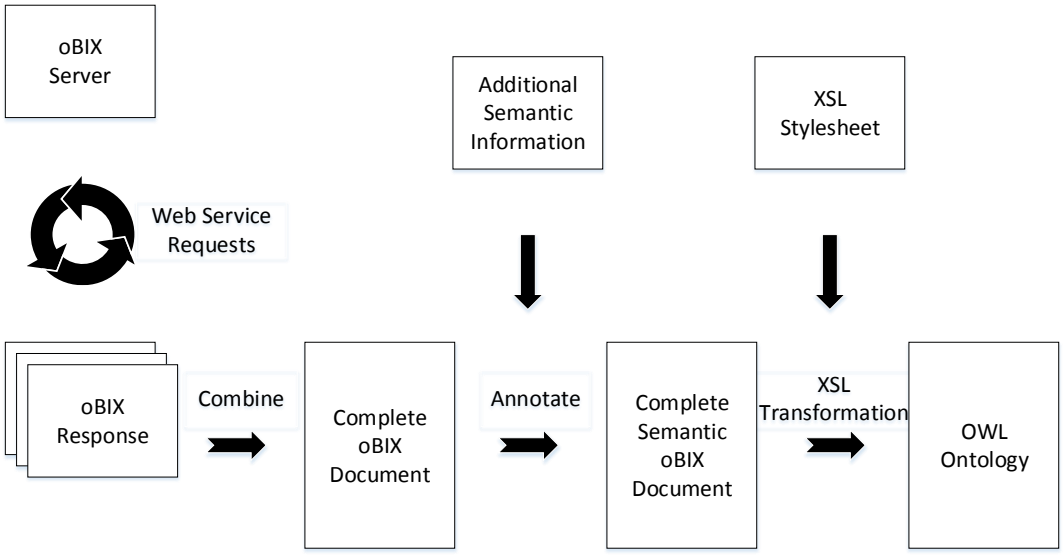

Figure 3.2: Transformation concept overview

provides semantically enriched data, e.g., a human reader will probably assume that the *temp_sensor* object from Listing 3.2 is actually a temperature sensor. Although, this cannot be inferred unambiguously, as the naming is controlled by the oBIX server and partly the BAS engineer. A naming convention can be defined that allows to mark relevant resources with meaningful names in order to annotate resources without doubt. This convention can be distributed to BAS engineers which could annotate various systems accordingly during development thus supporting reusability of the provided transformation. If the presented information at the oBIX interface does not provide any semantical annotations that can be used to implement a transformation, the engineer should be provided with the possibility to annotate oBIX resources. An example of a semantic annotation is shown in Listing 3.3 where a class of the target ontology is used as annotation.

Transform

In this final step, the *complete semantic oBIX document* is transformed to the target ontology by utilizing XSLT. Different target ontologies require customized transformations. If there is a one-to-one mapping between an oBIX resource and the resulting OWL individual the URI of the oBIX resource should be used as unique identifier for the

```
<obj name="example" href="/installation/example/" is="/knx/Installation" displayName=
    "Example">
  <list name="views" href="views" of="obix:ref /knx/View">
    <obj name="view_devices" href="view_devices" is="/knx/View"/>
      <list name="devices" href="devices" of="/knx/Device">
        <obj name="temp_sensor" href="temp_sensor" is="/knx/Device">
          .....
        </obj>
      </list>
    </obj>
  </list>
</obj>
```

Listing 3.2: Complete oBIX document example

```
<obj name="sensor" href="sensor" is="/knx/Device &EnergyResourceOntology;
    TemperatureSensor">
  .....
</obj>
```

Listing 3.3: oBIX semantic annotation example

```
<owl:NamedIndividual rdf:about="http://www.obixserver.org/installation/example/views/
    view_devices/devices/sensor">
  <rdf:type rdf:resource="&EnergyResourceOntology;TemperatureSensor"/>
  <isIn rdf:resource="&EnergyResourceOntology;LivingRoom1"/>
</owl:NamedIndividual>
```

Listing 3.4: Resulting OWL ontology example

individual thus reducing the complexity of the implementation. An exemplary output of the XSLT is shown in Listing 3.4.

3.2 Data synchronisation

The SILo implementation has to assure that the data that is provided by its SPARQL interface is up to date as to allow an adequate and reliable operation of the BAS. It is assumed that the data available from the oBIX server represents an actual state of the underlying BAS. Therefore, the task of the synchronization mechanism is to merge differences between the data provided by the oBIX server and the data presented by the SPARQL interface within a reasonable time frame.

If data values are changed at the SILo SPARQL interface, the internal representation, i.e., the OWL ontology has to be updated. Additionally, the affected values at the oBIX

server have to be updated in a timely manner. A single SPARQL update might require several HTTP requests to be transmitted.

One possibility to keep the OWL ontology actualized is to poll the data provided by the oBIX server recurrently. However, this approach does not scale well as a possibly large number of sensor and actuator values would produce a high network load even if the values have not changed. Because of that, the utilization of the oBIX *watch* mechanism is proposed. This approach still requires polling due to the fact that HTTP does not allow the server to contact the client, but reduces the network load substantially as only updated data values are transmitted. The oBIX Websocket binding [32] would obviate the usage of the watch service as it allows the server to contact the client on demand. The utilization of the watch service is depicted in Figure 3.3. As soon as an updated value is provided by the oBIX watch service, the internal representation needs to be updated accordingly. The polling interval between successive *pollChanges* calls should be adjustable as to allow different control scenarios to be realized.

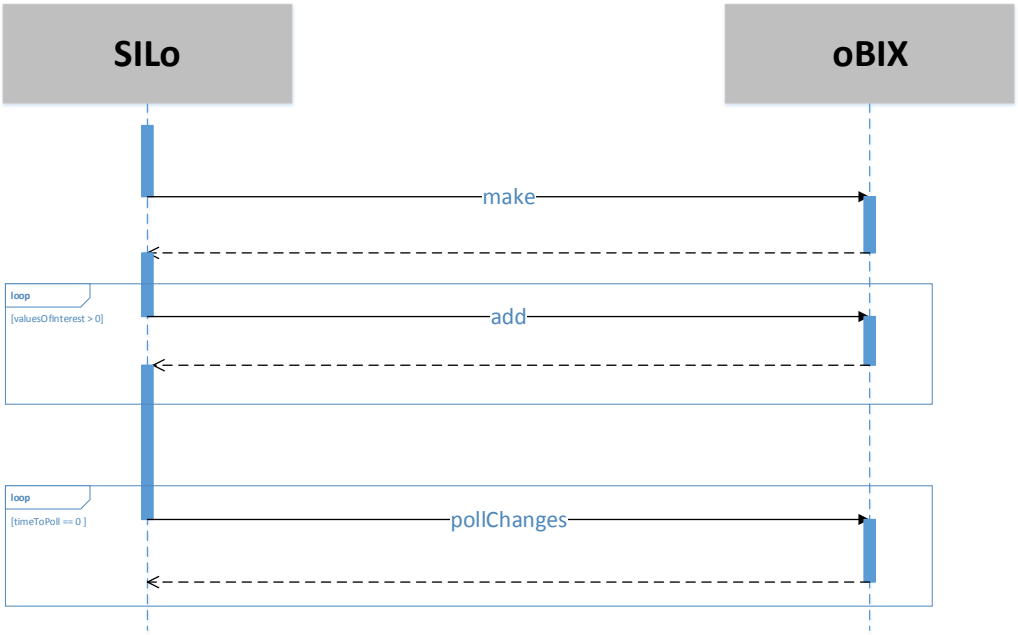

Figure 3.3: SILo synchronisation

3.3 Error handling

Two categories of errors are of relevance to SILo. For one, there is always the possibility of an HTTP error like 403 Forbidden or 404 Not-Found. Potentially, oBIX errors like *BadUriErr*, *PermissionErr* or *UnsupportedErr* which are proposed by standard or any custom error message provided by the oBIX server implementation might arise as well. As both are out of scope of the SILo implementation there are little remedy actions that could be taken from this side. Generally, any unexpected behaviour between the SILo implementation and the oBIX server should be signalized at the SPARQL interface in order to inform the users that the service is currently not available.

CHAPTER 4

Implementation - Case Study for IoTSys and ThinkHome

4.1 Model transformation

4.1.1 Engineering process

The proposed concept allows the implementation of algorithms performing the transformation. Nevertheless, a human interaction is required for different implementations of oBIX servers and different target ontologies. As soon as an implementation is provided for a given oBIX server and a target ontology, different BASs can be transformed automatically. Still, different BASs might require own semantic annotations if they are not provided by the oBIX server.

The engineering process for the transformation is outlined in the following:

1. Define target OWL ontology and model required extensions
2. Check oBIX server output for semantic information
3. Semantically annotate oBIX objects relevant to target ontology (optional)
4. Generate *complete semantic oBIX documents*
5. Implement an XSL transformation for each of the *complete semantic oBIX documents* to target ontology

A specification of the oBIX interface such as the KNX Web Service specification proposed in [35] reduces the engineering effort as it provides a consistent interface to oBIX. Due to the flexible and extendable nature of the oBIX object model, such specifications appear

desirable as they empower the M2M communication. Provided an existing transformation implementation for a specific target ontology, the engineering process is reduced to the semantic annotation of oBIX resources. By utilizing the ETS export functionality, it is even possible to fully automatize this process.

For the proof of concept implementation the steps described above included the following:

1. The ThinkHome ontology was chosen as target ontology. Some minor extensions were required in order to enable the transformation process.

2. Due to the fact that the used IoTSys oBIX server presents an export of the ETS model at its oBIX interface, most of the devices required no annotations as a naming convention was used for the KNX groups names.

3. Some devices not providing enough semantical information were annotated during the transformation process.

4. Traversing the IoTSys oBIX object tree starting from the lobby resulted in a very large *complete oBIX document* which added some complexity to the implementation of the XSL transformation. The information required for the transformation is provided by two distinctive oBIX nodes related to the building structure and functions provided by the BAS. Therefore, two smaller *complete semantic oBIX documents* for each of these nodes are created, resulting in less effort for the XSL transformation implementation.

5. For each of the two *complete semantic oBIX documents*, an individual XSL transformation is implemented.

4.1.2 Ontology

In order to implement Semantic Web features for automation systems, an ontology is required. As the reuse of ontologies is desirable in context of ontology design [59], the ThinkHome EnergyResourceOntology was chosen as a starting point. This ontology provides the basic means for modelling a diversified set of devices used in the sphere of home and building automation such as temperature controllers, light intensity sensors or switches. Additionally, it allows to locate these devices within a building model. As an example the *OnOffLightSwitch* class is shown in Listing 4.1.

Various minor extensions to the EnergyResourceOntology are required in order to enable a semantic interoperablity layer for oBIX. A new datatype property *webservicePayload* (cf. Listing 4.2) proved to be valuable as it allowed to model an oBIX content template to be matched against a received message or to be sent to the oBIX server.

Due to the modelling approach used by the EnergyResourceOntology, where the current state value of controllable devices is reflected by a unidirectional object property, it was not possible to operate on the according state value instance alone. Every state value

```
<!-- https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
    EnergyResourceOntology.owl#OnOffLightSwitch -->
<owl:Class rdf:about="&EnergyResourceOntology;OnOffLightSwitch">
  <rdfs:label rdf:datatype="&xsd;string">OnOffLightSwitch</rdfs:label>
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
        <rdf:Description rdf:about="&EnergyResourceOntology;Switch"/>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&EnergyResourceOntology;hasFunctionality"/>
          <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;
              NetworkFunctionality"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&EnergyResourceOntology;hasFunctionality"/>
          <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;
              OnOffNotificationFunctionality"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&EnergyResourceOntology;hasState"/>
          <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;OnOffState"/>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="&EnergyResourceOntology;controlledObject"/>
          <owl:allValuesFrom rdf:resource="&EnergyResourceOntology;
              LightingSystemResource"/>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:comment xml:lang="en">OnOffSwitch for Lights only, derives from ZigBee HA
      specifications</rdfs:comment>
</owl:Class>
```

Listing 4.1: EnergyResourceOntology;OnOffLightSwitch

```
<owl:DatatypeProperty rdf:about="&EnergyResourceOntology;webservicePayload">
  <rdfs:label rdf:datatype="&xsd;string">webservicePayload</rdfs:label>
  <rdfs:comment xml:lang="en">The payload of a web service call.</rdfs:comment>
  <rdfs:domain rdf:resource="&EnergyResourceOntology;StateValue"/>
</owl:DatatypeProperty>
```

Listing 4.2: EnergyResourceOntology;webservicePayload

change implied a complete traversal of the device RDF graph to find the instance to be updated. To overcome this limitation and improve performance, a new object property *isValueOf* is introduced, which is attached to a state value instance and provides a bidirectional connection to the device, i.e., a *Controllable* instance, which is the base class of all devices. This is depicted in Figure 4.1.

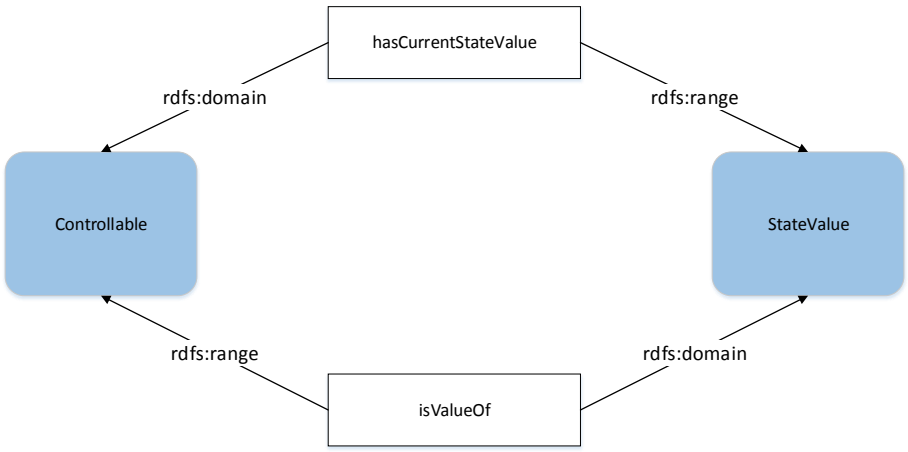

Figure 4.1: isValueOf ObjectProperty

For those devices which report more than just one variable e.g. a CO2 sensor which measures the temperature and the CO2 value, two new object properties were introduced *functionOf* and its inverse *providesFunction* as shown in Figure 4.2. This was necessitated by the approach chosen by the EnergyResourceOntology how devices are located within a building. Only the device instance itself is located within a room or a building. These properties allow to locate multiple functionalities within a building without having to parse the complete device RDF graph.

A new device as shown in Listing 4.3 was introduced which allows to model a simplified temperature controller, i.e,. the only variable it supports is the desired temperature.

Additional parameters of interest, which were not already present in the EnergyResourseOntology were added to the ontology. These parameters provide means to monitor the working hours of a switch, the current load of a switch and the number of switching cycles. The classes and properties required to monitor the current load are displayed in Figure 4.3. The other two parameters are modelled accordingly.

In order to enable the signalization of error states a simple hierarchy of error classes as depicted in Figure 4.4 was introduced. The base class *SILoError* provides a description of the error as a string via the *hasSILoErrorDescription* property. The subclasses *SILoObixError* and *SILoHttpError* are used to distinguish between HTTP and oBIX errors.

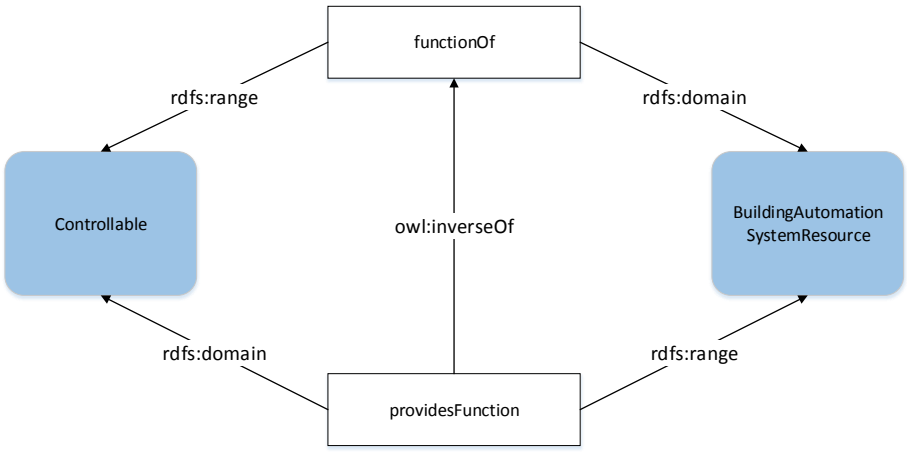

Figure 4.2: providesFunction ObjectProperty

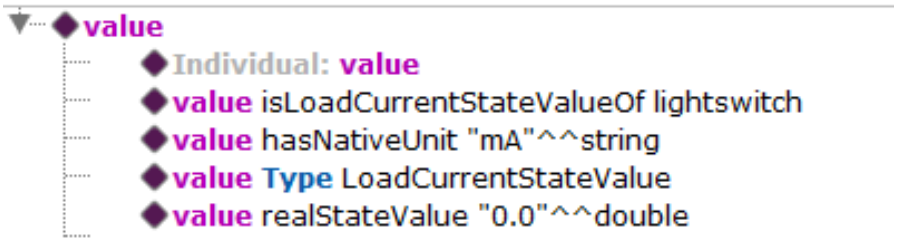

Figure 4.3: EnergyResourceOntology;LoadCurrentStateValue

Another change related to the building model was the necessity to make both object properties *contains* and its inverse *isIn* transitive in order to reflect the fact that if a room is on the second floor of a building, it is still in the building.

4.1.3 Location

In order to meet the requirements of this work, it is essential to localize all the devices we want to control. The IoTSys oBIX server used as a reference implementation throughout this work, parses the KNX ETS export project file and provides its contents via a web interface. Therefore, the building model that is provided by the used oBIX server implementation corresponds to the ETS building model. An exemplary representation of an ETS building model is shown in Figure 4.5.

The building model presented by the used oBIX server is displayed in Listing 4.4. During the transformation process each oBIX room type is mapped to an EnergyResourceOntology class. The simple mapping is shown in Table 4.1.

During the XSL transformation the according building model is transformed to an OWL

```
<owl:Class rdf:about="&EnergyResourceOntology;TemperatureController">
  <rdfs:label rdf:datatype="&xsd;string">TemperatureController</rdfs:label>
  <rdfs:subClassOf rdf:resource="&EnergyResourceOntology;
      BuildingAutomationSystemResource"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&EnergyResourceOntology;hasState"/>
      <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;TemperatureState"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&EnergyResourceOntology;hasFieldOfApplication"/>
      <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;
          TemperatureFieldOfApplication"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&EnergyResourceOntology;hasFunctionality"/>
      <owl:someValuesFrom rdf:resource="&EnergyResourceOntology;
          TemperatureRegulationFunctionality"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:comment xml:lang="en">A mechanism that allows controlling a temperature.</rdfs:
      comment>
</owl:Class>
```

Listing 4.3: EnergyResourceOntology;TemperatureController

```
<list href="part/" is="obix:Range">
  <obj name="building" href="part/building" displayName="Building"/>
  <obj name="buildingpart" href="part/buildingpart" displayName="BuildingPart"/>
  <obj name="floor" href="part/floor" displayName="Floor"/>
  <obj name="room" href="part/room" displayName="Room"/>
  <obj name="corridor" href="part/corridor" displayName="Corridor"/>
  <obj name="stairway" href="part/stairway" displayName="Stairway"/>
  <obj name="distributionboard" href="part/distributionboard" displayName="
      DistributionBoard"/>
</list>
```

Listing 4.4: oBIX Building model

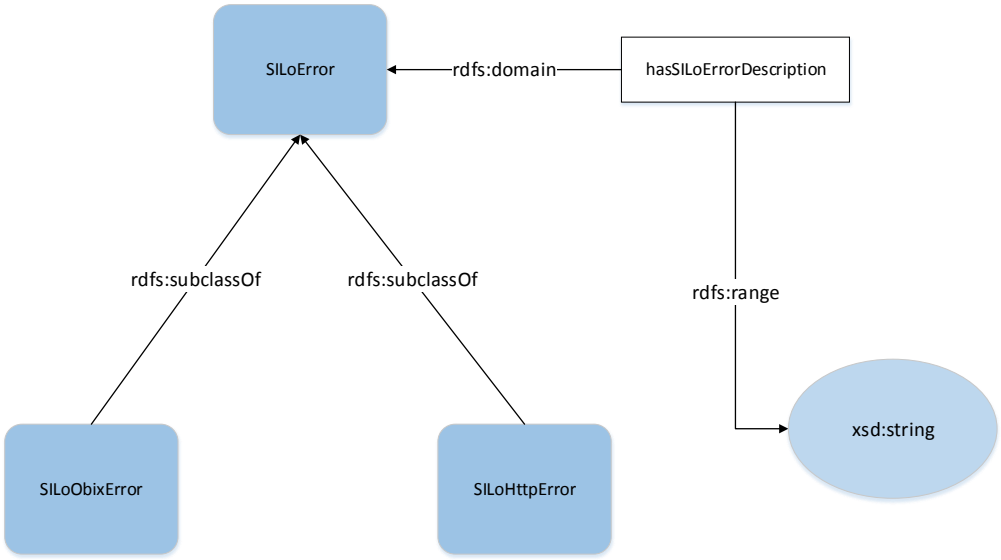

Figure 4.4: SILoError model

oBIX	EnergyResourceOntology
building	&EnergyResourceOntology;Building
buildingpart	&EnergyResourceOntology;BuildingPart
floor	&EnergyResourceOntology;BuildingStorey
room	&EnergyResourceOntology;Room
corridor	&EnergyResourceOntology;Corridor
stairway	&EnergyResourceOntology;Stairway
distributionboard	&EnergyResourceOntology;DistributionBoard

Table 4.1: Building Mapping

building model that is represented by EnergyResourceOntology individuals. The layout of the building is preserved by the transformation. Additionally, all the devices found in the building are localized within the OWL building model. An example outcome of the transformation is shown in Figure 4.6. Every OWL class, property or individual is uniquely identified by an *rdf:about* attribute. During the transformation the absolute oBIX server URL is used as identifier for OWL instances.

It has to be noted that the XSL transformation implementation is dependent on the oBIX

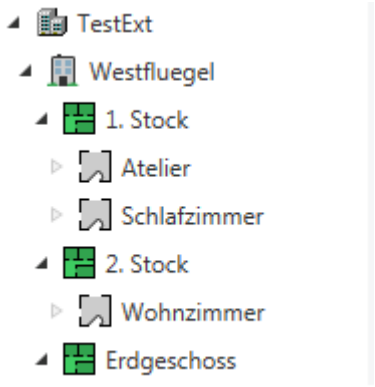

Figure 4.5: ETS building model

```
<obj name="P-013F-0_BP-3" href="a_lab/" is="knx:Part" displayName="A-Lab">
  <enum name="type" href="a_lab/type" val="room" range="/enums/part"/>
  <list name="instances" href="a_lab/instances" of="knx:InstancePart">
    <obj name="P-013F-0_DI-53" href="a_lab/instances/4" is="knx:InstancePart">
      <ref name="reference" href="/networks/e183_1/entities/presence_detector_up_258_21
          /1" is="knx:Entity" displayName="Presence detector UP 258/21"/>
    </obj>
    <obj name="P-013F-0_DI-61" href="a_lab/instances/7" is="knx:InstancePart">
      <ref name="reference" href="/networks/e183_1/entities/
          knx_co2___feuchte___und_temperatursensor/1" is="knx:Entity" displayName="
          KNX CO2-, Feuchte- und Temperatursensor"/>
    </obj>
    <obj name="P-013F-0_DI-62" href="a_lab/instances/8" is="knx:InstancePart">
      <ref name="reference" href="/networks/e183_1/entities/switching_actuator_n_562_11
          /1" is="knx:Entity" displayName="Switching Actuator N 562/11"/>
    </obj>
  </list>
</obj>
```

Listing 4.5: oBIX building example

server used. Slight differences in regards of the building model representation would require an adaptation to the implementation. Furthermore, using a different ontology would also require an adaptation to the XSLT implementation.

4.1.4 Devices

In a first step, the building model is transformed and all devices i.e. all sensors, switches or controllers are localized within the building model. In order to draw semantic conclusions about an heterogeneous automation scenario, all these devices need to be transformed to OWL individuals as well. The transformation process for entities is implemented as an

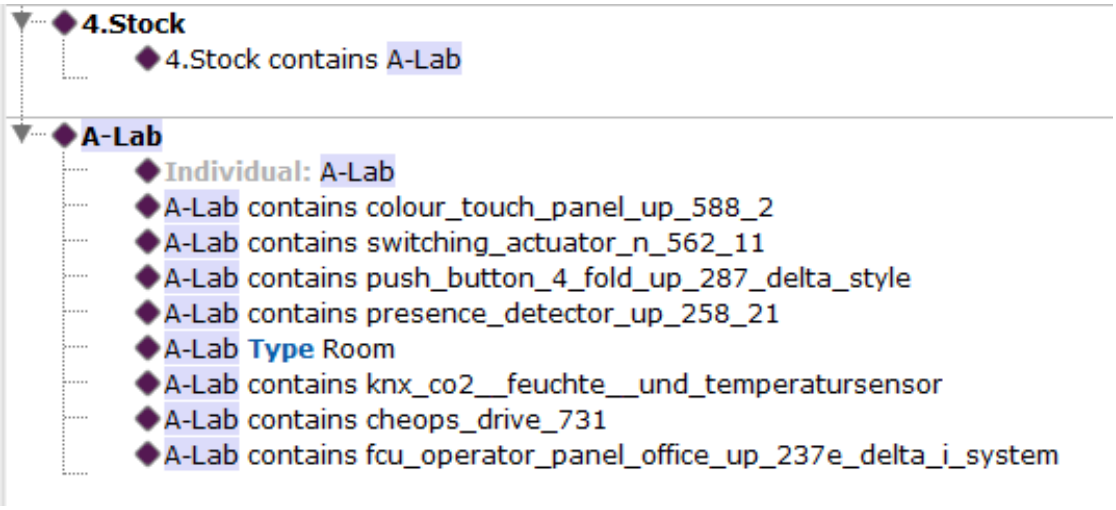

Figure 4.6: OWL building model example

CO2 sensor
Humidity Sensor
Light Sensor
On/Off Light Switch
Presence Sensor
Temperature Sensor
Temperature Controller

Table 4.2: Supported Device Types

XSL transformation similar to the building model transformation.

The transformation itself is dependent on the specific device type like light switch, presence detector or humidity sensor. A set of device types supported in this work is shown in Table 4.2.

A reference to each KNX device used in a KNX automation scenario is published via the oBIX server as an entity. Such an entity can support multiple functions e.g. a KNX switch actuator 4-fold is represented as a single entity which itself provides up to 4 logical functions. The different logical functions are defined by the KNX groups to which the device is programmed. It should be noted that in the used OWL building model only the entity is localized but not the logical function. Therefore, we need a connection between the logical groups and the device in order to localize each of the functions within a building. The interconnections between entities, buildings and logical functions is shown in Figure 4.7.

In order to determine all logical functions of a device, a manual engineering process is required. In ETS it is possible to provide each of the KNX groups with a meaningful

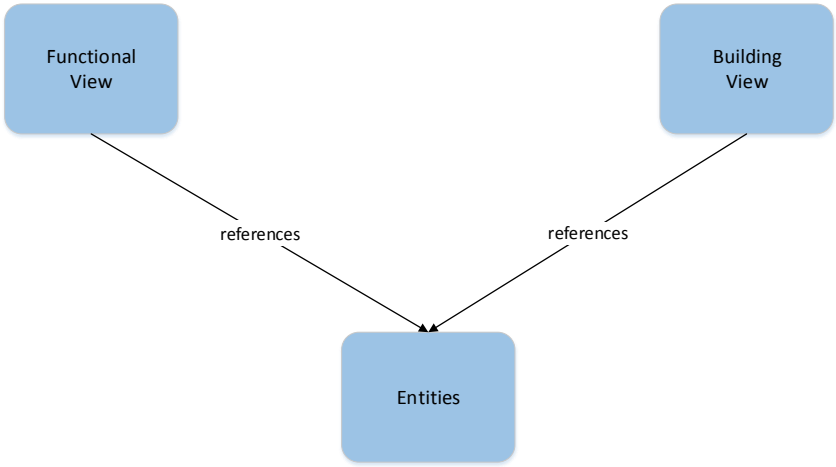

Figure 4.7: IoTSys oBIX model

Keyword	Device Type
co2	CO2 sensor
relfeuchte	Humidity Sensor
licht	On/Off Light Switch
hk_raumtemperatur_ist	Temperature Sensor
hk_raumtemperatur_soll	Temperature Controller

Table 4.3: Device type naming convention

name. With regard to automating the transformation, a naming convention was set up which is used to determine the logical function of a KNX group and thus the device type itself. The used group names and according device type are shown in Table 4.3. During the transformation process each group name is evaluated to determine the according device type. If a group name contains a specific keyword, it is mapped to the according device type individual. Each logical function that is provided by an entity is modelled as an own device type i.e. if a switch actuator 4-fold is programmed to 4 different KNX groups after the transformation there will be 4 OWL switch individuals representing each of them.

In order to allow the tracking of the current load, switching cycle count and the operating hours count, *StateValue* instances are connected to matching *Switch* individuals. The detection of these *StateValue* individuals is also based on a naming convention as shown in Table 4.4.

A naming convention is useful if a new KNX project is started and this convention is followed throughout the whole project. In such a case, an existing XSL transformation could be reused multiple times. If the KNX project already exists and it requires

Keyword	*StateValue*
laststrom	Load Current State
betriebsstunden	Operating Hours State
schaltspielzahl	Switching Cycles State

Table 4.4: State value naming convention

```
1 http://localhost:8080/networks/e183_1/entities/presence_detector_up_258_21/1/datapoints/
      output_brightness=EnergyResourceOntology;LightSensor
2 http://localhost:8080/networks/e183_1/views/functional/groups/e183_1/groups/iotsensoren/
      groups/indoorbrigthnesssensor=EnergyResourceOntology;LightSensor
3
4 http://localhost:8080/networks/e183_1/entities/presence_detector_up_258_21/1/datapoints/
      output_presence=EnergyResourceOntology;PresenceSensor
5 http://localhost:8080/networks/e183_1/views/functional/groups/e183_1/groups/iotsensoren/
      groups/presencedetector=EnergyResourceOntology;PresenceSensor
```

Listing 4.6: Semantic annotation properties

additional semantic information it should be verified, whether it is easier to adapt an existing XSL transformation than to rewrite the whole KNX project to comply with a naming convention.

As we used an existing ETS project, which lacked any semantical information for *PresenceSensor* and *LightSensor* instances, an annotation process was required in order to enable the transformation for these device types. The according oBIX nodes identified by their URL are enhanced with semantical information as shown in Listing 4.6. This annotation process results in a complete semantic oBIX document, where the according oBIX nodes are annotated via the oBIX contract mechanism. This information can be used throughout the XSL transformation in order to define the according device type.

In the following, the supported device types and state values that are the result of XSL transformation shall be presented.

CO2 Sensor

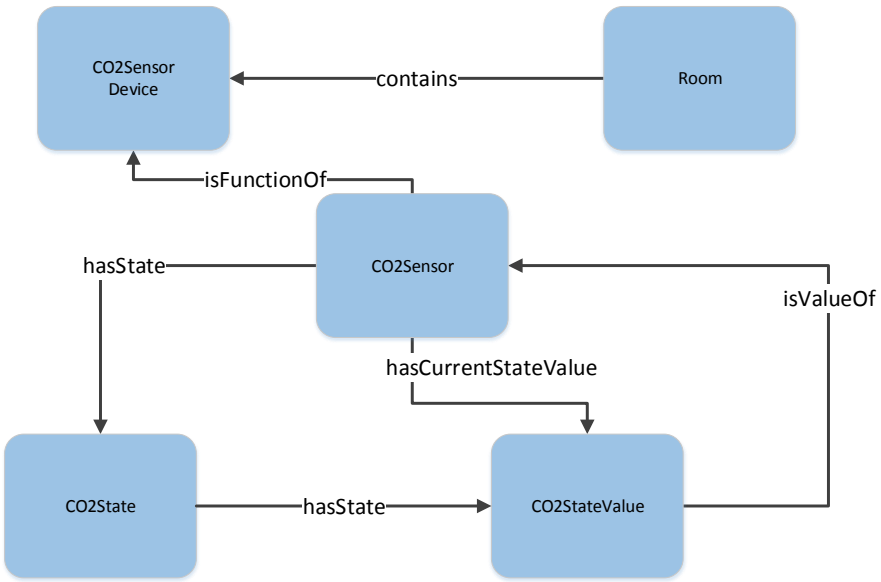

Figure 4.8: CO2Sensor

The *CO2Sensor* is connected to *CO2State* instance via the *hasState* property and to the *CO2StateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.8. In the EnergyResourceOntology used throughout this work the *StateValue* instances always contain the measurements. Mostly they have a property *realStateValue* containing a numerical value and a *hasNativeUnit* property holding the unit of the value. In this case, the unit used is parts per million (PPM). The *CO2Sensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. The *CO2StateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message.

Humidity Sensor

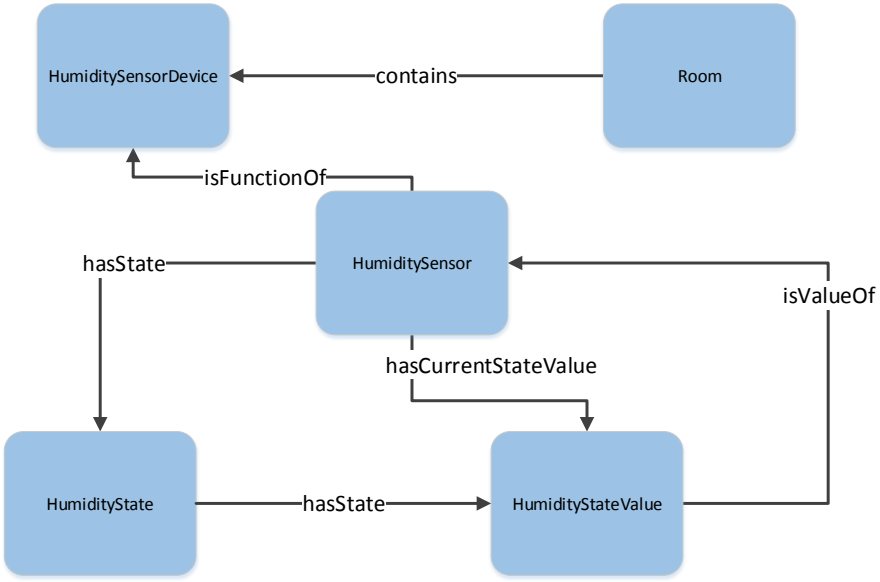

Figure 4.9: HumiditySensor

The *HumiditySensor* is connected to *HumidityState* instance via the *hasState* property and to *HumidityStateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.9. An *HumidityStateValue* instance contains the measurements and has a property *realStateValue* containing a numerical value and a *hasNativeUnit* property holding the unit of the value. In this case, the unit used is percent (%). The *HumiditySensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. The *HumidityStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message.

Light Sensor

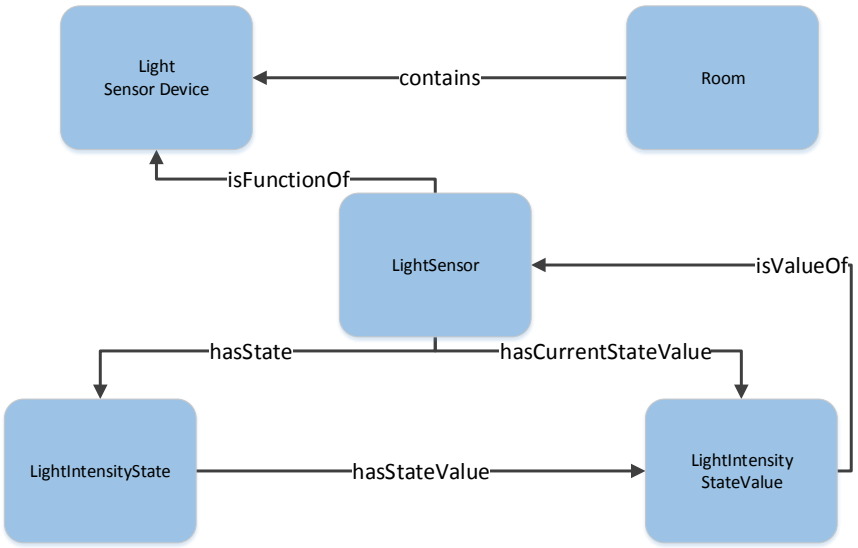

Figure 4.10: LightSensor

The *LightSensor* is connected to *LightIntensityState* instance via the *hasState* property and to *LightIntensityStateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.10. A *LightIntensityStateValue* instance contains the measurements and has a property *realStateValue* containing a numerical value and a *hasNativeUnit* property holding the unit of the value. In this case, the unit used is *lux*. The *LightSensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. The *LightIntensityStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message.

Presence Sensor

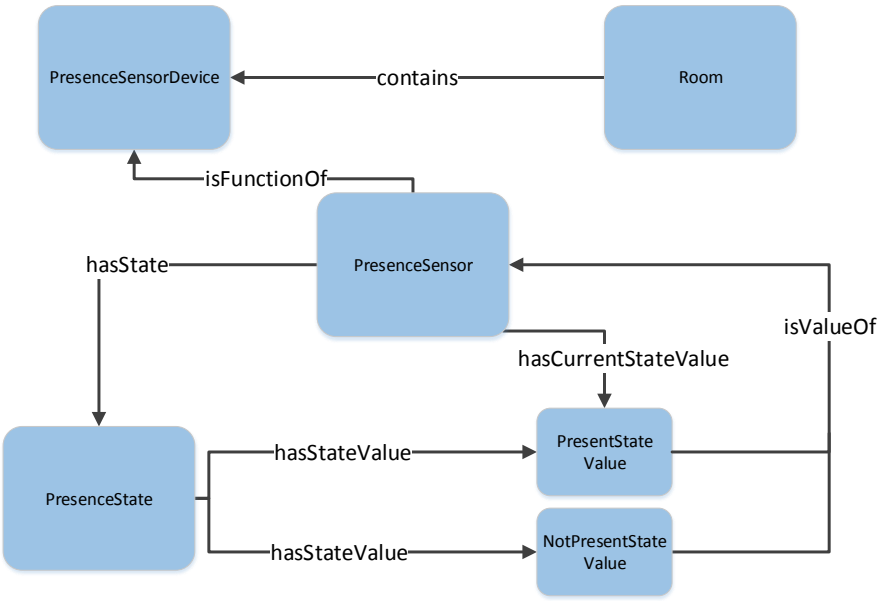

Figure 4.11: PresenceSensor

The *PresenceSensor* is connected to *PresenceState* instance via the *hasState* property and either to a *PresentState* or *NotPresentState* instance via the *hasCurrentStateValue* property as shown in Figure 4.11. This sensor provides just two states the *PresentState* and *NotPresentState* which have no units. The *PresenceSensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. Both, the *PresentState* and *NotPresentState* instances contain a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message and to decide if someone is present or not.

Temperature Sensor

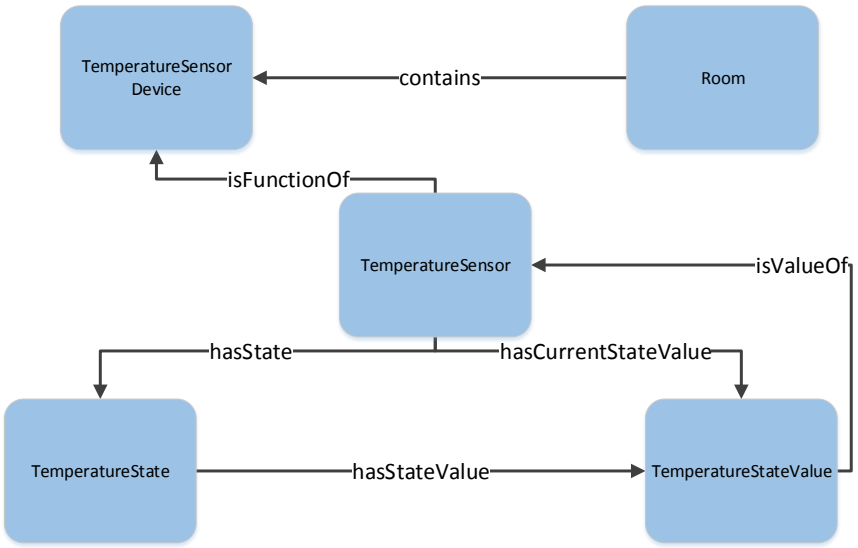

Figure 4.12: TemperatureSensor

The *TemperatureSensor* is connected to *TemperatureState* instance via the *hasState* property and to *TemperatureStateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.12. A *TemperatureStateValue* instance contains the measurements and has a property *realStateValue* containing a numerical value and a *hasNativeUnit* property holding the unit of the value. In this case, the unit used is Celsius (°C). The *TemperatureSensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. The *TemperatureStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message.

Temperature Controller

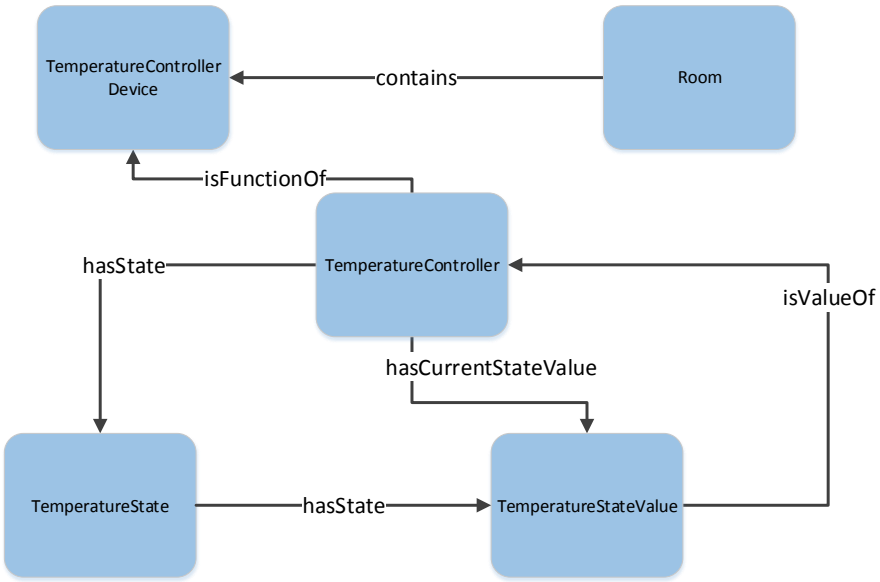

Figure 4.13: TemperatureController

The *TemperatureController* is very similar to the *TemperatureSensor* in regards of its structure, but its purpose is different as it is used to set a desired temperature. The controller is connected to *TemperatureState* instance via the *hasState* property and to *TemperatureStateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.13. A *TemperatureStateValue* instance contains the measurements and has a property *realStateValue* containing a numerical value and a *hasNativeUnit* property holding the unit of the value. In this case, the unit used is Celsius (°C). The *TemperatureSensor* is localized within the building via the device to which it is attached via the *isFunctionOf* property. The *TemperatureStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server, i.e., to be able to retrieve the value from the oBIX message.

Light Switch

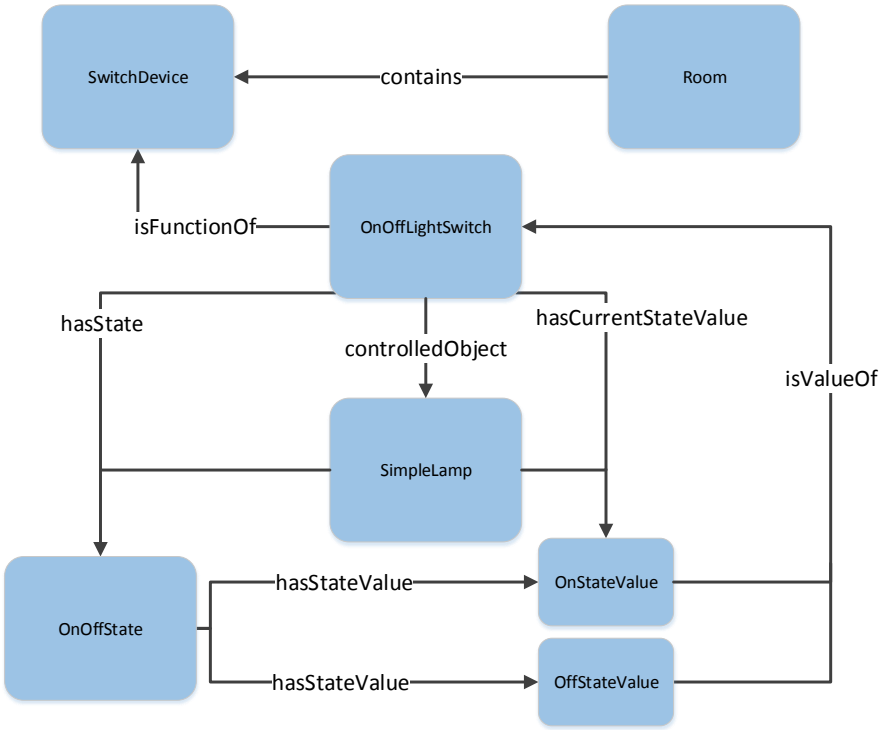

Figure 4.14: LightSwitch

The *OnOffLightSwitch* is connected to *OnOffState* instance via the *hasState* property and either to an *OnStateValue* or *OffStateValue* instance via the *hasCurrentStateValue* property as shown in Figure 4.14. This switch provides just two state values the *OnStateValue* and *OffStateValue* which have no units. The *OnOffLightSwitch* is localized within the building via the device to which it is attached via the *isFunctionOf* property. Both, the *OnStateValue* and *OffStateValue* instances contain a *webservicePayload* property which is used as a template to send messages to the oBIX server, i.e., to be able to turn the lights on or off. An important aspect of the *OnOffLightSwitch* is that it is connected to a SimpleLamp individual via the *controlledObject* property. Both, the switch and the lamp instances point to the same individual of type *OnOffState* and both show to the same *StateValue* instance via the *hasCurrentStateValue* property. This has to be considered during execution of SPARQL updates as both individuals need to be updated accordingly.

Operating Hours *StateValue*

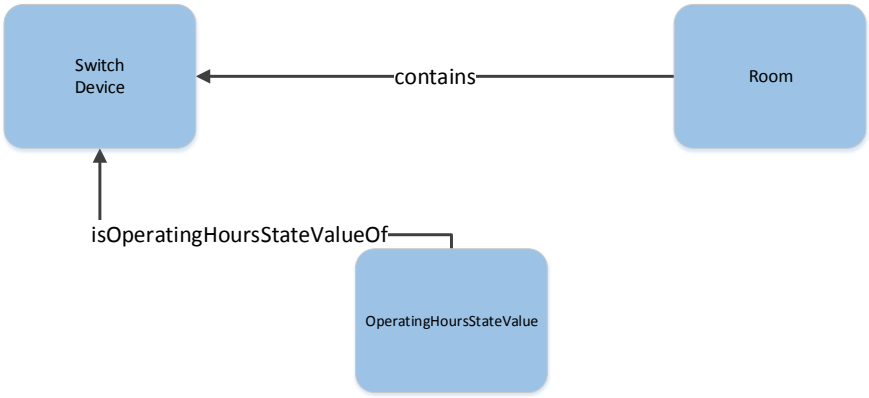

Figure 4.15: Operating hours count state value

This is a *StateValue* instance that should return the amount of operating hours of a corresponding switch. As shown in Figure 4.15, the hours are stored within an instance of *OperatingHoursStateValue*. This instance is attached to the actual switch via an object property *isOperatingHoursStateValueOf* which has an inverse property *hasOperatingHoursStateValueOf*. The corresponding switch is localized within a building part. The *OperatingHoursStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server.

Switching Cycles *StateValue*

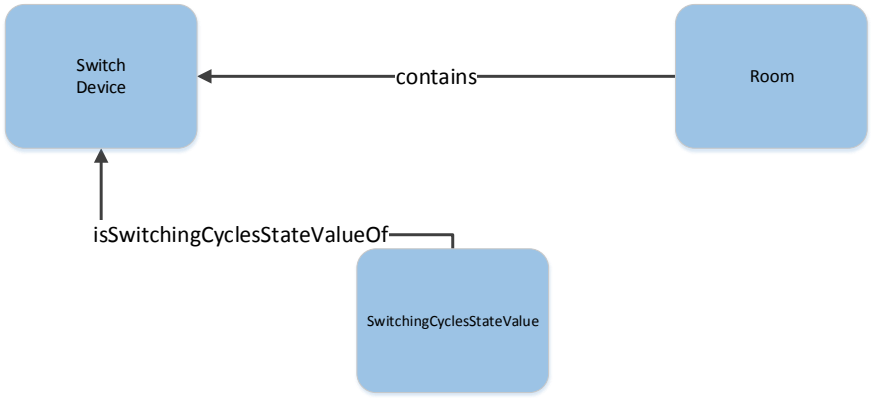

Figure 4.16: Switching cycles count state value

This is a *StateValue* instance that should return the switching cycle count of a corresponding switch. As shown in Figure 4.16, the amount of switching cycles is stored within an instance of *SwitchingCycleStateValue*. This instance is attached to the actual switch via an object property *isSwitchingCycleStateValueOf* which has an inverse property *hasSwitchingCycleStateValueOf*. The corresponding switch is localized within a building part. The *SwitchingCycleStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server.

Load Current *StateValue*

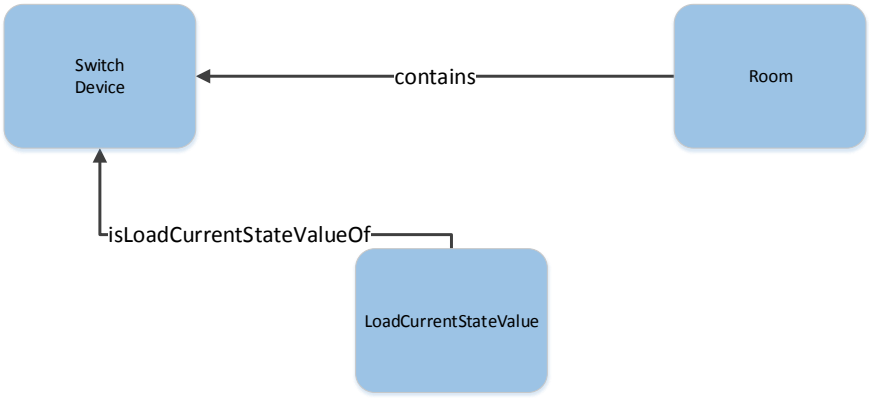

Figure 4.17: Load current state value

This is a *StateValue* instance that should return the actual load current of a corresponding switch. As shown in Figure 4.17, the current is stored within an instance of *LoadCurrentStateValue*. This instance is attached to the actual switch via an object property *isLoadCurrentStateValueOf* which has an inverse property *hasLoadCurrentStateValueOf*. The corresponding switch is localized within a room. The *LoadCurrentStateValue* instance contains a *webservicePayload* property which is used as a template to match received messages from the oBIX server.

4.2 SILoTool

The SILoTool is a software application implemented in Java. The intention of this application is to support the engineer throughout the transformation process and to allow a user to execute SPARQL queries and updates in order to control a BAS regardless of the heterogeneous technologies used at the field level. As shown in Figure 4.18, it consists of different modules which provide different features. The modules are shortly described herein:

```
java SILoTool
--create --serverUrl= [--startUrl=] [--recursive=] [--target=]
--annotate --serverUrl= [--startUrl=] [--recursive=] [--annotationProperties
    =] [--target=]
--transform --serverUrl= [--annotationProperties=] --transformationProperties
    = [--target=]
--run --ontology= --serverUrl= --transformationProperties= [--
    annotationProperties=] [--watchPeriod=] [--ontologyHandlerPackage=]
```

Listing 4.7: SILoTool usage

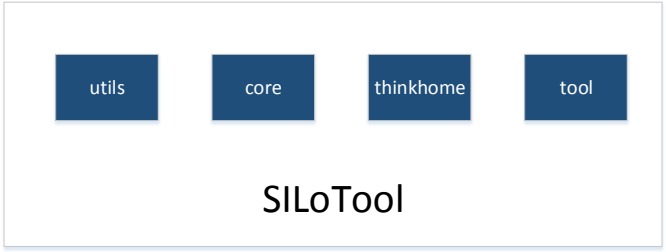

Figure 4.18: SILoTool modules

utils This module provides the means to perform an XSL transformation. Additionally, it supports HTTP communication via GET, POST and PUT requests. If offers an XML parsing functionality as well.

core This module contains the core components of the SILoTool. It implements an oBIX client used to communicate with the oBIX server supporting the oBIX watch mechanism. Additionally, it provides a SILo framework used to adapt oBIX messages to RDF statements of the target ontology and store them in an Apache Jena [60] RDF triple store. Finally, it supports the binding to different target ontologies via the *SILoOntologyHandler* interface.

thinkhome This module is the *SILoOntologyHandler* interface implementation for the ThinkHome ontology. It provides the means to update the RDF triple store as to generate oBIX update requests.

tool The tool module allows to run the SILoTool in order to perform different tasks such as to generate a *complete oBIX document*, a semantically annotated *complete semantic oBIX document*, to perform an XSL transformation or to run the SILo Web crawler.

The usage of the SILoTool is shown in Listing 4.7 and is described in the following:

create

The create command allows to create a *complete oBIX document*.

serverUrl The address of the oBIX server.

startUrl The optional address of the specific oBIX server object node for which the document should be created.

recursive An optional parameter indicating whether oBIX references should be resolved to the objects they reference.

target File name where the result should be stored to. If no target is provided the result is printed to *System.out*.

annotate

The annotate command allows to create a *complete semantic oBIX document*.

serverUrl The address of the oBIX server.

startUrl The optional address of the specific oBIX server object node for which the document should be created.

recursive An optional parameter indicating whether oBIX references should be resolved to the objects they reference.

annotationProperties Java property file defining how specified nodes should be annotated via the oBIX contracts mechanism (cf. Listing 4.6).

target File name where the result should be stored to. If no target is provided the result is printed to *System.out*.

transform

The transform command allows to perform XSL transformations for specified oBIX object nodes.

serverUrl The address of the oBIX server.

annotationProperties Optional Java property file defining how specified nodes should be annotated via the oBIX contracts mechanism (cf. Listing 4.6).

transormationProperties Java property file defining how specified nodes should be transformed to the target ontology. The format of the file is shown in Listing 4.8. It specifies which XSL transformation file to use for which oBIX object node, whether to recursively traverse the oBIX object node and to resolve references.

```
1
2 0_xsl_http=http://localhost:8080/networks/e183_1/views/building/
3 0_xsl_file=buildings.xsl
4 0_xsl_traverse=false
5
6 1_xsl_http=http://localhost:8080/networks/e183_1/views/functional/groups
7 1_xsl_file=groups.xsl
8 1_xsl_traverse=false
```

Listing 4.8: SILoTool transformation properties

target File name where the result should be stored to. If no target is provided the result is printed to *System.out*.

run

The run command allows to perform a SILo transformation and to run the SILo Web crawler in order to control a BAS via a SPARQL interface.

ontology File name of the target ontology.

serverUrl The address of the oBIX server.

annotationProperties Optional Java property file defining how specified nodes should be annotated via the oBIX contracts mechanism (cf. Listing 4.6).

transormationProperties Java property file defining how specified nodes should be transformed to the target ontology. The format of the file is shown in Listing 4.8. It specifies which XSL transformation file to use for which oBIX object node, whether to recursively traverse the oBIX object node and to resolve references.

watchPeriod An optional parameter specifying the delay between two consecutive oBIX watch *pollChanges* requests.

ontologyHandlerPackage An optional parameter specifying where to search for the *SILoOntologyHandler* interface implementation. By default the crawler searches in *silo.ontology.handler.**.

The SILo Web Crawler behaves as an oBIX client and connects to an oBIX server as shown in Figure 4.19. The oBIX server is responsible for abstracting from a heterogeneous automation scenario and providing a consolidated view on the situation via Web services. The information gathered from the oBIX server is stored in a Knowledge Base (Apache Jena OWL model) which is accessible to a user via SPARQL. This Knowledge Base can be both queried via SPARQL and also updated via SPARQL. An update operation in SPARQL is forwarded to the oBIX server and allows to actively change values of the BAS, e.g., the desired temperature.

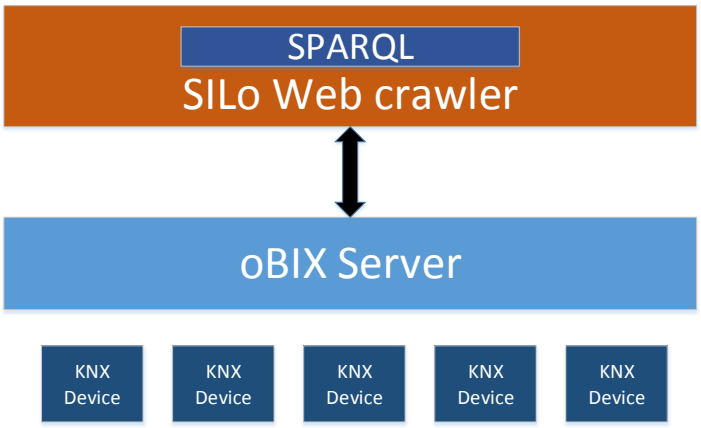

Figure 4.19: SILo Web crawler

4.2.1 Apache Jena

Apache Jena [60] is a free and open source Java framework for the Semantic Web. It provides several APIs that interact together in order to process RDF data. Figure 4.20 depicts its general architecture. At the core of Jena there is the RDF Application Programming Interface (API) which allows manipulating RDF graphs. The contents of an RDF triple store are presented in a container named *Model* which is a high-level abstraction of a directed graph.

Jena also provides an Ontology API which allows to handle RDF Schema and OWL ontologies. Jena supports all ontology languages from the most expressive OWL to the weakest RDF Schema and provides a consistent interface across the different language variants. Ontologies are stored in an *ontology model* which is an extension of the presented *Model*. Nevertheless the *ontology model* is stored in an RDF store as the Ontology API does not change the way how the data is persisted underneath. All Ontology API interactions are mapped to RDF triple store functions.

Jena provides a set of internal reasoners and additionally enables external reasoners such as Pellet [49] to attach to Jena through its Inference API. The internal reasoners usually extend the basic *ontology model* with inferred RDF triples and thus provide a consistent

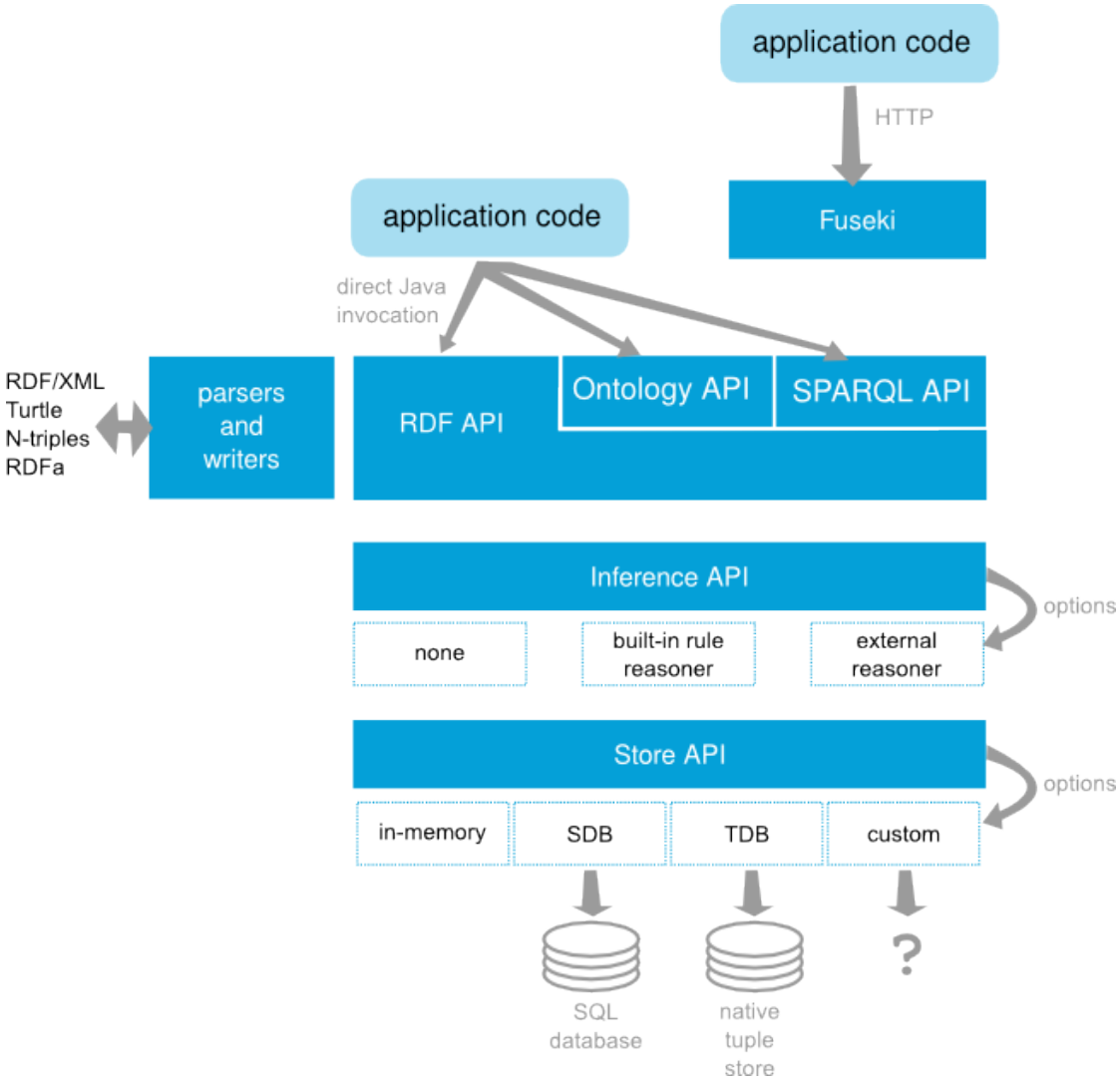

Figure 4.20: Apache Jena architecture [60]

view on the extended *ontology model* which allows to access inferred triples exactly the same way as basic triples.

Jena also provides support for SPARQL and SPARQL Update. It even has implemented a standalone SPARQL server named Fuseki which allows to execute SPARQL queries over the Web.

The Jena Store API is responsible for the storage of RDF data. It is possible to keep the data in memory, but also to utilize a mechanism named SDB which stores the data in a relational database. TDB is a high performant storage solution for RDF stores which directly stores the data to disk and fully supports SPARQL.

4.2.2 Architecture

The general architecture overview of the SILo Web crawler is shown in Figure 4.21.

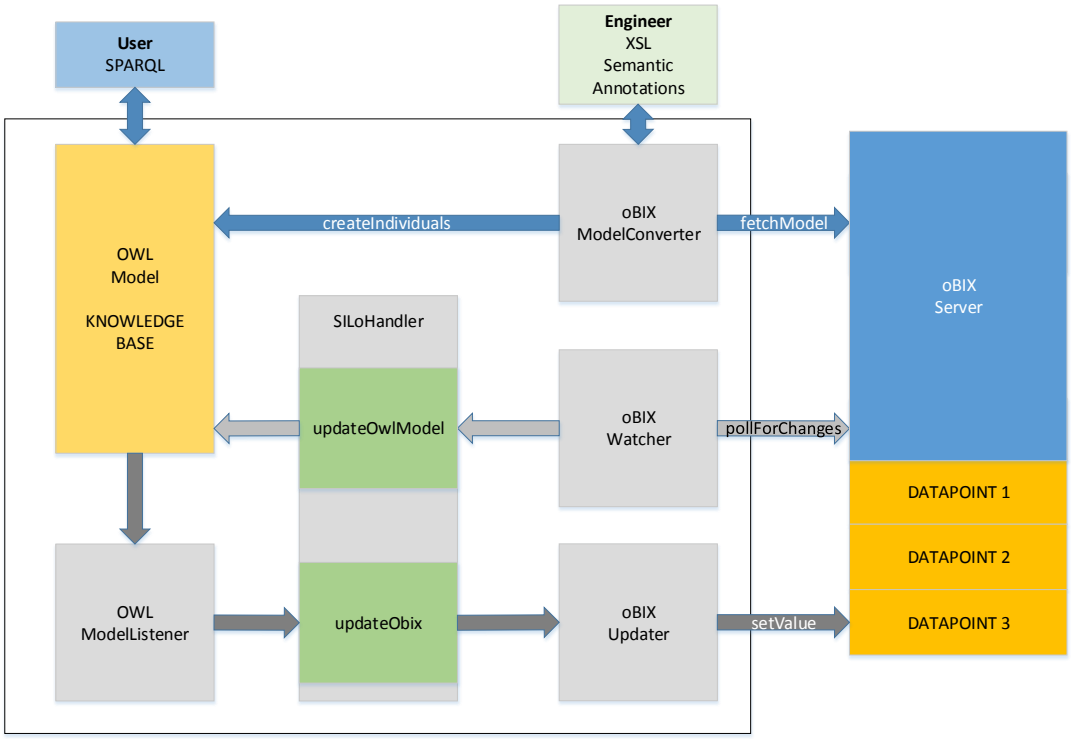

Figure 4.21: SILo Web crawler architecture

The central core component of the SILo Web crawler is the *SILoHandler* which acts as an interface between the OWL model and the oBIX server. Every request is handled by the singleton *SILoHandler* instance. Basically the *SILoHandler* is just a wrapper for the target ontology specific *SILoOntologyHandler* implementation to which every call is forwarded. The *SILoHandler updateOwlModel* method shown in Listing 4.9 is called when an update was signalized by the oBIX watch mechanism and takes care of the synchronization. The *SILoHandler* is the only component which is allowed to update the RDF triple store.

The *OwlModelListener* is called whenever the OWL model changes. It is used to propagate changes of the OWL model to the oBIX server. On every change the control is forwarded to the *SILoHandler* via the *updateObix* method which delegates the handling to the target ontology specific *SILoOntologyHandler*. The *SILoHandler* provides the means to decide whether a change is relevant to the oBIX server.

The *ObixWatcher* is used to track changes on oBIX server side. It polls for changes from the oBIX server. The delay between two consecutive *pollChanges* executions is

```
1  public void updateOwlModel(ObixWatchOut obixWatchOut) {
2    for (ObixWatchOutListItem item : obixWatchOut.getList()) {
3      SILoDeviceControl siLoDeviceControl = controlledDevices.get(item.getHref()
           );
4      try {
5        siLoDeviceControl.getSemaphore().acquire();
6        siLoDeviceControl.setObixUpdateRequired(false);
7        List<OwlModelUpdateOperation> owlModelUpdateOperations = ontologyHandler.
             getOwlModelUpdateOperations(item);
8        updateOntologyModel(owlModelUpdateOperations);
9        siLoDeviceControl.setObixUpdateRequired(true);
10     } catch (Exception e) {
11       SILoErrorHandler.handleError(e);
12     } finally {
13       siLoDeviceControl.getSemaphore().release();
14     }
15   }
16 }
```

Listing 4.9: SILoHandler.updateOwlModel

configurable. In case of a change, the *SILoHandler* instance is informed and performs all further tasks required. The *ObixUpdater* is utilized in order to send oBIX requests to the oBIX server.

The *ObixModelConverter* fetches the oBIX data and performs a SILo transformation. The results are stored to the RDF triple store.

4.2.3 Functionality

After startup, the adapted EnergyResourceOntology which represents the TBox is loaded and an OWL model is created. Therefore, the already mentioned Apache Jena framework is used. In order to enable inferring of logical conclusions within the ontology, the Pellet [49] reasoner is instantiated.

Afterwards, the *ObixModelConverter* fetches the oBIX server and performs the SILo transformation according to the specified annotation and transformation properties. The result of the SILo transformation is a representation of the oBIX model as a set of target ontology individuals which are eventually stored into the RDF triple store. This mechanism is depicted in Figure 4.22.

In a separate step, the units used by the oBIX server such as PPM or lux are retrieved and published locally via the *ObixUnitFactory*. This factory instance is used to resolve the units received from oBIX datapoints.

Eventually, the *SILoHandler* and the *SILoOntologyHandler* are instantiated and initialized. The *ObixWatcher* class provides the means to create an oBIX watcher for a datapoint and to query the oBIX server for any updates. As soon as a watcher has been created

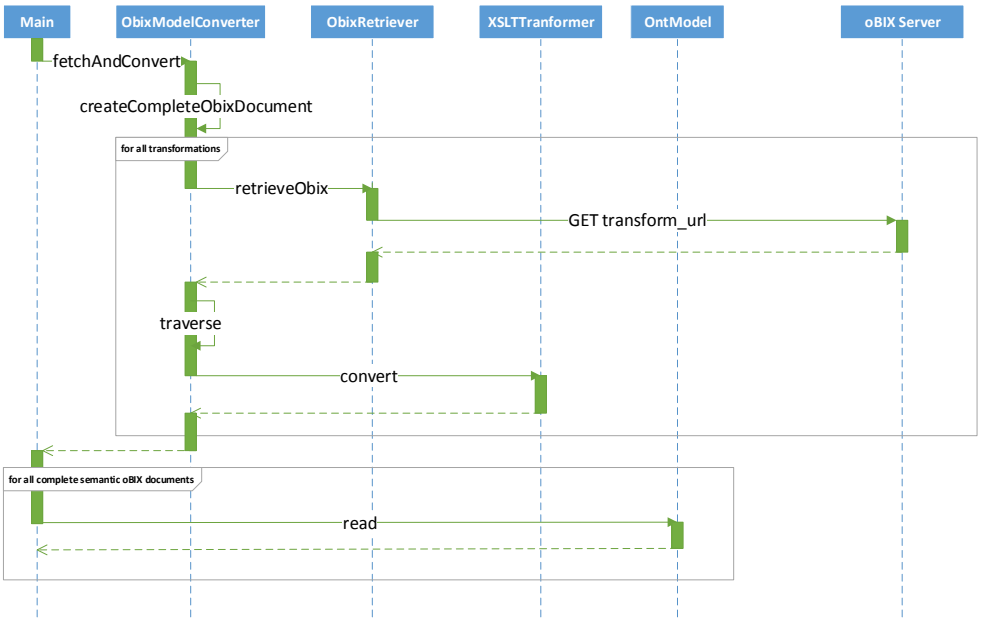

Figure 4.22: SILo Web crawler startup

for a single device, a timer task is started which polls the server for any changes with a configured delay. This functionality is displayed in Figure 4.23.

The *OwlModelListener* class extends the Apache Jena *ObjectListener* class. Its implementation allows to react to changes of the OWL model. In case of a SPARQL update, the listener is called and delegates to the *SILoHandler* in order to update the oBIX server accordingly, as shown in Figure 4.24. If the change happened on the oBIX server, the listener is informed as well, but in this case the oBIX server should not be updated as it was the actual source of the change. The *SILoHandler* takes care that the propagation of the update is stopped via the *obixUpdateRequired* member as displayed in Figure 4.25.

The *SILoErrorHandler* singleton class is responsible for the error propagation. In case of an HTTP or oBIX error, a *SILoError* object is instantiated and the *SILoErrorHandler* enters the error mode. While in error mode, the user is informed about the error state in case of SPARQL query or update executions. This allows the user to take measures suited to leave the error condition state.

4.2.4 SILoOntologyHandler

Generally, the *SILoOntologyHandler* depicted in Figure 4.26 performs the following task:

Figure 4.23: ObixWatcher Functionality

- Specify all the devices that need to be watched by the oBIX watch mechanism.
- Create oBIX update requests that are to be sent to the oBIX server in case of SPARQL updates.
- Create model update operations in order to update the OWL model after a change on the oBIX server was observed.

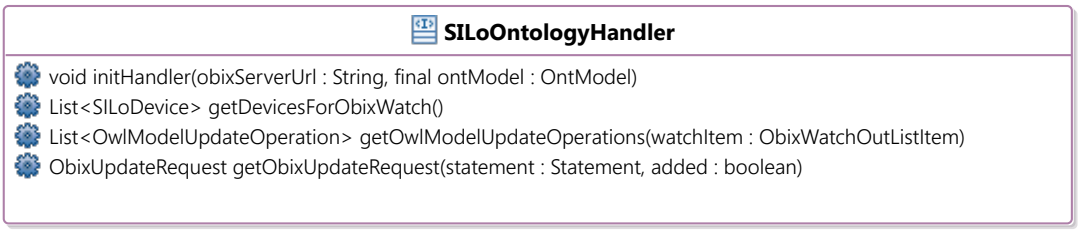

Figure 4.26: SILoOntologyHandler

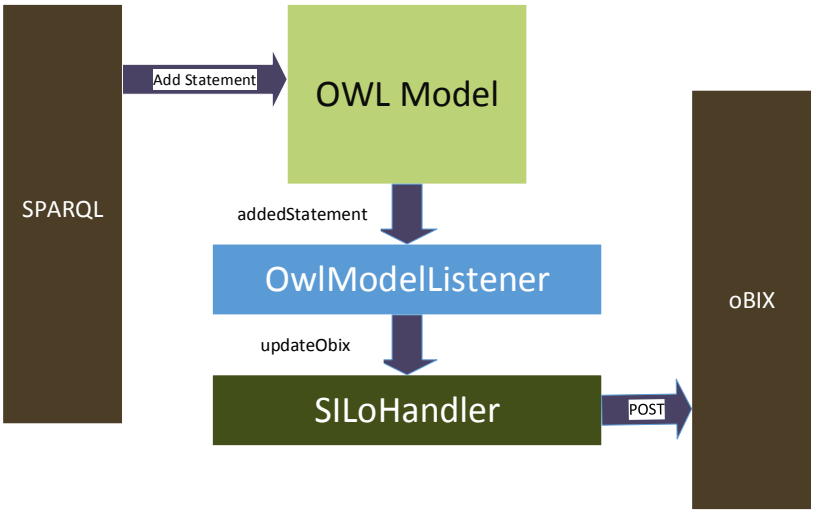

Figure 4.24: SPARQL Update

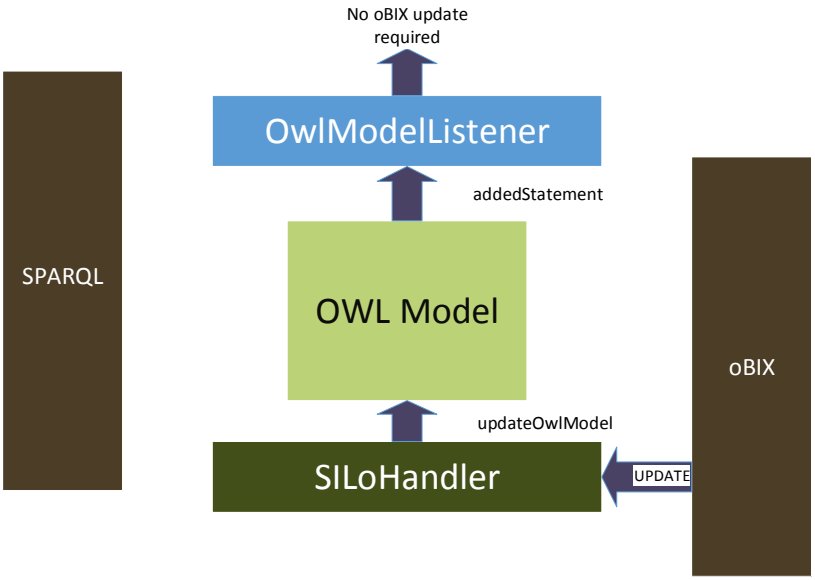

Figure 4.25: oBIX Update

All these tasks are dependent on the target ontology. By implementing the *SILoOntologyHandler* interface it is possible to bind various target ontologies to the SILoTool. By default, the SILoTool searches the *silo.ontology.handler.** package for implemen-

tations of the *SILoOntologyHandler* interface which are eventually instantiated and injected to the SILo Web crawler. It is also possible to configure the SILoTool to look in an arbitrary package by providing the *ontologyHandlerPackage* startup parameter. The ThinkHome *SILoOntologyHandler* implementation provided with the SILoTool is a reference implementation of the *SILoOntologyHandler* interface for the ThinkHome ontology.

The *ObixSewoaHandler* class is the ThinkHome implementation of the *SILoOntologyHandler* interface. It allows to register *ObixDeviceHandler* implementations which handle requests for specific device types or state values. During initialization, every *ObixDeviceHandler* searches the OWL model for individuals of its designated OWL class and returns their URLs, which are used as an identifier for the specific device. The *ObixSewoaHandler* maintains a mapping between these devices and the corresponding *ObixDeviceHandler* implementation. In case of an *updateOwlModel* or *updateObixRequest* call, it consults the map in order to find the responsible *ObixDeviceHandler* and delegates the execution to it.

As the *SILoHandler* is the only component allowed to update the OWL model, the *ObixDeviceHandler* returns a list of *OwlModelUpdateOperation* instances to be executed in order to update the model in case of an oBIX update. Currently four different types of operations for an OWL individual are supported:

- Add an object type property statement.
- Add a data type property statement.
- Remove a specific property statement.
- Remove all property statements.

In order to create *OwlModelUpdateOperation* instances, the *OwlModelUpdateOperation* factory can be utilized. Eventually the *SILoHandler* executes these operations asserting an atomic execution. This approach eases the implementation of the *SILoOntologyHandler* interface as the *SILoHandler* is required to take care of the synchronization.

In case of a SPARQL update, the *ObixSewoaHandler* delegates the generation of *ObixUpdateRequest* instances to the responsible *ObixDeviceHandler*. It is up to the *ObixDeviceHandler* instance to decide whether an update is necessary. If the oBIX server needs to be informed about a change triggered by a SPARQL update execution, it must return an *ObixUpdateRequest* instance, specifying the HTTP request type, the target URL and the payload, which is forwarded to the *ObixUpdater* in order to send the request.

CHAPTER 5

Test Lab

The functionality of the SILo Web crawler was evaluated in the A-Lab of the Institute of Computer Aided Automation at the Vienna University of Technology. It provides a well-equipped laboratory which supports different automation field protocols such as KNX and BACnet with various sensors and actuators, such as a presence detector, a light sensor and a temperature controlling unit. The general setup for this evaluation is depicted in Figure 5.1.

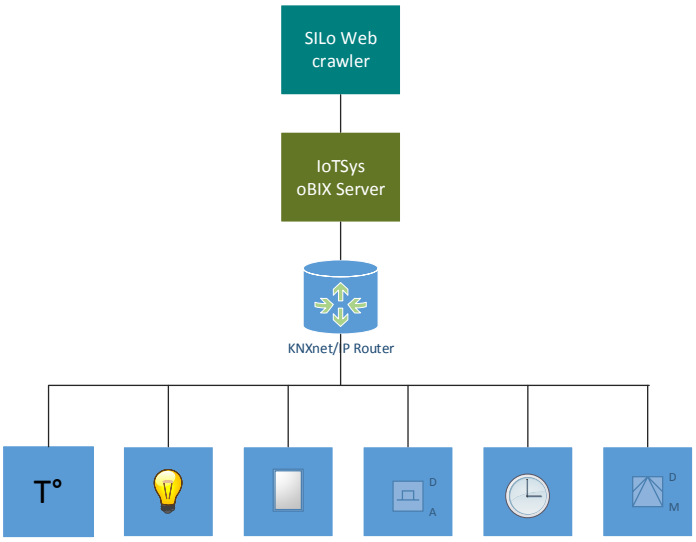

Figure 5.1: A-Lab Setup

Additionally, a KNX suitcase as shown in Figure 5.2, providing additional devices, such as a CO2 sensor and a humidity sensor, was attached to the KNX installation.

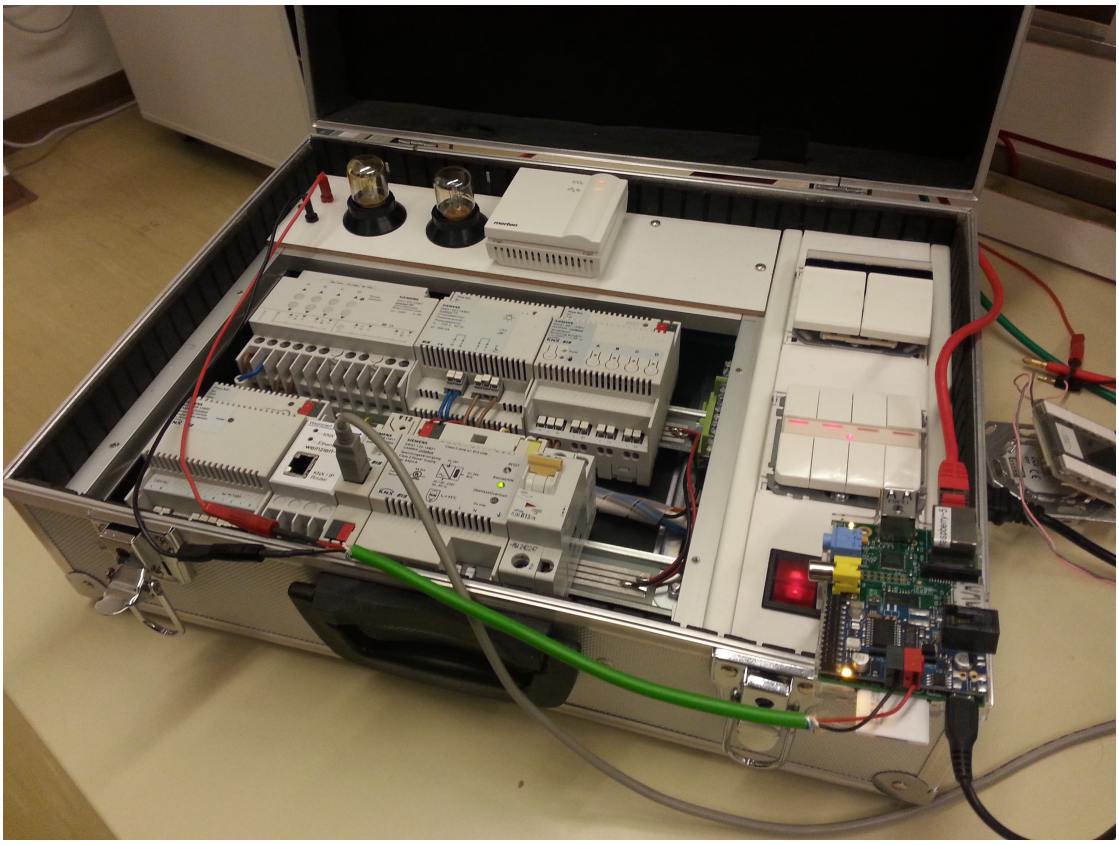

Figure 5.2: A-Lab KNX Suitcase

In order to demonstrate the functionality of the SILo Web crawler, the following use cases, already mentioned in the problem statement of this work, are considered:

- Is every switching actuator of a distinct floor in "off" state?
- How many rooms of a building are occupied?
- Which lamps in a building have exceeded a distinct operating time?
- Turn on the light in a specific room!

For evaluation purposes, a set of SPARQL queries and updates, adapted to the A-Lab setting, was prepared and will be described herein.

In order to learn whether all lights of a specific floor are switched off, the query shown in Listing 5.1 may be used. It searches for rooms where the light is still on, if there

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX EnergyResourceOntology: <https://www.auto.tuwien.ac.at/downloads/thinkhome/
    ontology/EnergyResourceOntology.owl#>
SELECT ?switch ?location
WHERE {
  ?switch rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
      EnergyResourceOntology.owl#OnOffLightSwitch> .
  ?switch EnergyResourceOntology:hasCurrentStateValue ?type .
  ?type rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
      EnergyResourceOntology.owl#OnStateValue> .
  ?switch EnergyResourceOntology:functionOf ?device .
  ?device EnergyResourceOntology:isIn+ <http://localhost:8080/networks/e183_1/views/
      building/parts/treitlstrasse/parts/4stock> .
  ?device EnergyResourceOntology:isIn ?location
}
```

Listing 5.1: SPARQL all lights on floor in "off" state query

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX EnergyResourceOntology: <https://www.auto.tuwien.ac.at/downloads/thinkhome/
    ontology/EnergyResourceOntology.owl#>
SELECT (count(distinct *) as ?count)
WHERE {
  ?switch EnergyResourceOntology:hasCurrentStateValue ?type .
  ?type rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
      EnergyResourceOntology.owl#PresentState> .
  ?switch EnergyResourceOntology:functionOf ?device .
  ?device EnergyResourceOntology:isIn+ <http://localhost:8080/networks/e183_1/views/
      building/parts/treitlstrasse> .
}
```

Listing 5.2: SPARQL count occupied rooms query

is no such room, it must be concluded that the lights are switched off on the whole floor. Therefore, it searches for all individuals of type *OnOffLightSwitch* which have as current state value an individual of type *OnStateValue*. Further, the search is restricted to individuals which are bound via the *functionOf* property to devices, which are located transitively via the *isIn* property on a given floor.

The query shown in Listing 5.2 counts the occupied rooms of a building by searching for all rooms in a building, which contain a presence sensor that has as current state value a *PresentStateValue* individual attached. As the presence sensor is a function of a device, which is located within a building, the *functionOf* property is used to get the device which is transitively found via the *isIn* property.

The query shown in Listing 5.3 looks for light switches which have exceeded a specified

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX EnergyResourceOntology: <https://www.auto.tuwien.ac.at/downloads/thinkhome/
    ontology/EnergyResourceOntology.owl#>
SELECT ?switch ?value ?unit
WHERE {
  ?switch rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
      EnergyResourceOntology.owl#OnOffLightSwitch> .
  ?switch EnergyResourceOntology:hasOperatingHoursStateValue ?type .
  ?type rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
      EnergyResourceOntology.owl#OperatingHoursStateValue> .
  ?type EnergyResourceOntology:realStateValue ?value .
  ?type EnergyResourceOntology:hasNativeUnit ?unit .
  ?switch EnergyResourceOntology:functionOf ?device .
  ?device EnergyResourceOntology:isIn+ <http://localhost:8080/networks/e183_1/views/
      building/parts/treitlstrasse> .
  FILTER(?value >= 1000)
}
```

Listing 5.3: SPARQL exceeded working hours query

amount of operating hours. In detail, the search is restricted to look for individuals of type *OnOffLightSwitch*, which have as a current state value, bound by the *hasCurrentStateValue* property, an individual of type *OperatingHoursStateValue*. Again, the *OperatingHoursStateValue* is a function of a device located within a building via the transitive *isIn* property. The *OperatingHoursStateValue* individual is attached to the count of operating hours by the *hasRealStateValue* property, which is used to limit the search result only to those devices which have exceeded 1000 operating hours.

The SPARQL update query shown in Listing 5.4 is used to turn on the lights in a given room. This is achieved by deleting the RDF triple connected by the *hasCurrentStateValue* property with an *OnStateValue* instance and inserting a new triple with this property where the object of the property is an individual of type *OffStateValue*. Both the *OnOffLightSwitch* and the corresponding *SimpleLamp* individuals have to be updated. The attached device is retrieved via the *functionOf* property and is eventually located via the transitive *isIn* property. As soon as this update is executed, an oBIX request is sent to set the corresponding values of the oBIX server.

The queries described worked as expected during the evaluation phase. It could be demonstrated that it is feasible to control a complex building automation scenario with the SILo Web crawler via SPARQL. This requires extensive knowledge of the ontology in use, but allows to draw advanced logical conclusions about a building automation scenario. By using the SILo Web crawler a heterogeneous automation scenario could be semantically integrated regardless of the field technologies used on lower levels of abstraction.

The SILoTool currently supports only SPARQL queries submitted via the command line.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX EnergyResourceOntology: <https://www.auto.tuwien.ac.at/downloads/thinkhome/
       ontology/EnergyResourceOntology.owl#>
3  DELETE
4    {
5      ?switch EnergyResourceOntology:hasCurrentStateValue ?v
6      ?lamp EnergyResourceOntology:hasCurrentStateValue ?v
7    }
8    INSERT
9    {
10     ?switch EnergyResourceOntology:hasCurrentStateValue ?statevalue
11     ?lamp EnergyResourceOntology:hasCurrentStateValue ?statevalue
12   }
13   WHERE {
14     ?switch EnergyResourceOntology:hasCurrentStateValue ?v .
15     ?v rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
           EnergyResourceOntology.owl#OffStateValue> .
16     ?switch EnergyResourceOntology:hasState ?state .
17     ?state EnergyResourceOntology:hasStateValue ?statevalue .
18     ?statevalue rdf:type <https://www.auto.tuwien.ac.at/downloads/thinkhome/ontology/
           EnergyResourceOntology.owl#OnStateValue> .
19     ?switch EnergyResourceOntology:functionOf ?device .
20     ?device EnergyResourceOntology:isIn+ <http://localhost:8080/networks/e183_1/views/
           building/parts/treitlstrasse/parts/4stock/parts/aufputzkasten_a_lab_rechts> .
21     ?switch EnergyResourceOntology:controlledObject ?lamp
22 }
```

Listing 5.4: SPARQL update to turn on lights

This has proven not to be an optimal user interface for SPARQL as the queries and updates are rather complex. Referring to the usability of the SILoTool, there is room for improvement, like a graphic user interface which supports the user during the query crafting.

From the scalability point of view, the SILo Web crawler provides a solution which scales well to an increasing number of field level devices with regards of monitoring. This is achieved by the utilization of the oBIX watch mechanism which allows to monitor multiple devices via a single HTTP request. Even if the count of field level devices is further increased, there is no increase of the communication traffic. Still, this approach requires a recurrent polling of the oBIX server. In order to control multiple devices, multiple network requests are required, as oBIX complies to the REST paradigm, which is resource oriented. In this case, an increasing number of devices results in a higher network load as well. Relying on the oBIX REST binding, this is the optimum that can be reached in terms of scalability.

One shortcoming of the current implementation is that an update of the oBIX server configuration, i.e., adding or removing of devices, is not automatically detected by the

SILo Web crawler. Such a scenario imposes a high probability of oBIX errors. Such errors are detected by the crawler and provided to the user upon execution of further queries or updates.

CHAPTER 6

Conclusion

6.1 Summary

The proposed SILo provides the possibility to dynamically map an arbitrary oBIX server representation to an OWL ontology, which can be used as a common vocabulary for distributed autonomous agents. The presented transformation process can fully be automated, once an XSL transformation for a specific oBIX server implementation and a target ontology is developed, provided that the oBIX server yields consistent semantic information for various BASs. If the oBIX server is not able to provide such information, an annotation step, as presented in this work, might allow to reuse an existing XSL transformation, thus allowing a less complex semi-automated process. Due to the flexibility of the oBIX standard, distinct oBIX server implementations will require a customized transformation, as long as the oBIX specification provides no means for explicit semantic information. For different target ontologies, own transformations have to be developed. It seems also feasible to reuse an existing transformation for a specific target ontology and eventually perform the mapping to another OWL ontology by applying Semantic Web technologies. This approach requires a verification, though.

The proof of concept implementation, presented in this work, that is based on the IoTSys oBIX server and the ThinkHome ontology, supports the engineer throughout the different phases of the SILo transformation process. It provides the means to generate a *complete oBIX document*, which resolves oBIX references to object instances. Further, it permits the annotation of oBIX object nodes with semantic information, resulting in a *complete semantic oBIX document*. Eventually, an XSL transformation can be performed on the gained data, yielding OWL individuals of the target ontology. The proposed transformation process is extremely flexible, as it allows to perform the SILo transformation in a single iteration on a *complete oBIX document* starting with the oBIX lobby object. However, this might result in a more complex XSL transformation implementation, as the generated oBIX document might include a lot of non-essential

information useless for the transformation. The SILo transformation can also be executed in an iterative process on multiple smaller *complete oBIX documents* just including the nodes that contain the information relevant for the transformation. This allows the implementation of less complex XSL style-sheets tailored to single oBIX object nodes of interest.

Furthermore, the prototype implementation provides a SILo Web crawler, which can be used to control a heterogeneous BAS. Therefore, the individuals obtained by the SILo transformation process are stored within the Apache Jena RDF triple store and made accessible via the SPARQL protocol. The SILo Web crawler basically resolves SPARQL queries and updates to according oBIX request and takes care of the synchronization between the oBIX server the OWL model. Additionally, it provides a simplified error handling, where potential HTTP and oBIX errors are communicated to the user upon execution of SPARQL queries or updates. Besides, it provides the means to use different target ontologies by implementing the *SILoOntologyHandler* interface.

During the evaluation, it was demonstrated that new insights about the current state of a BAS can be gained by the utilization of SPARQL queries. Location based queries allowed to determine whether all lights on a given floor are switched off, or to count the occupied rooms in a building. Likewise, it was shown that it is possible to control various devices of a BAS, on grounds of their location within a building, by executing SPARQL updates. Summarizing, it can be stated that SILo provides the means for a comprehensive control of BASs.

6.2 Further work

As shown in Figure 6.1, it can be concluded that the features currently provided by the SILo Web crawler are not sufficient to guarantee the state of global interoperability. One of the steps missing in order to reach this goal is a common description, i.e., a widely adopted ontology that is reused by different software agents. However, the SILo Web crawler can easily be adapted to such an ontology, as soon as it is available. Furthermore, an interface is required that allows autonomous agents to interconnect via SILo, such as a Java API or a network binding. This would enable the involved agents to programmatically access the information stored in the RDF triple store, thus increasing their context-awareness and eventually contributing to a more energy efficient control and operation of buildings.

Referring to the vision of a SWoT [24] where autonomous distributed agents are collectively controlling a BAS, questions related to transaction management consequently arise. Multiple agents concurrently controlling a BAS via SILo might lead to an inconsistent state of the BAS due to a lack of transaction management features. As a RESTful service is characterized through stateless requests, the lack of transaction support is inherent in the system. The oBIX specification itself does not provide any suggestions on transaction management on a higher level. An oBIX server implementation might implement such features, but this is currently out of scope of the oBIX standard. In order to ensure

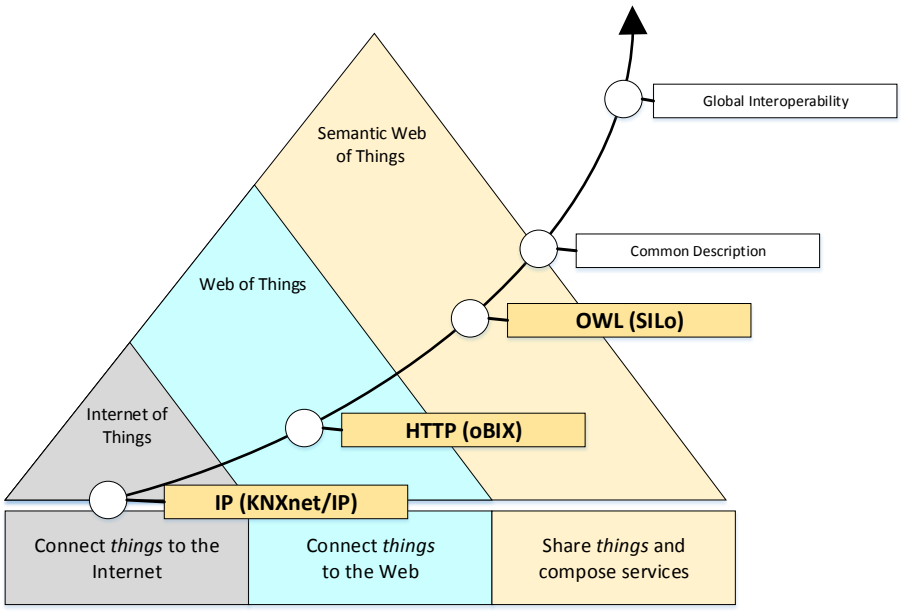

Figure 6.1: Semantic Web of Things (adapted from [24])

data integrity, as one of the most critical properties of a system, transactions should provide means for ACID (atomicity, consistency, isolation and durability) [61]. As a single SPARQL update might result in several HTTP requests, ensuring atomicity is in the scope of SILo. The SILo implementation might be improved to allow that either all involved requests are successfully executed or none. The oBIX batch mechanism is of no use here, as it relinquishes the handling of partial failures to the implementors. Consistency, isolation and durability have to be enabled by the oBIX implementation, as multiple SILo agents might concurrently access the oBIX interface as to control the underlying BAS.

Security aspects of SILo require further examination. As user authorization and authentication of oBIX servers are out of scope of the oBIX standard, security features of oBIX servers will differ from implementation to implementation, hence hindering a general approach. Generally, it seems desirable to encrypt the whole communication between the oBIX server and the SILo Web crawler by utilizing HTTPs. Authorization and authentication can be ensured by application of Semantic Web technologies, thus allowing the users to execute specific SPARQL queries or updates only in case they are authorized to do so.

Supporting the oBIX Websocket binding would provide better means for scalability, as this binding allows a bidirectional connection between oBIX server and the SILo Web crawler without the necessity to poll. Likewise, connecting SILo to an oBIX server, supporting the CoAP protocol, as IoTSys, would be beneficial in terms of scalability. However, the current oBIX standard does not provide such a binding, yet.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 59, 60, 74

BAS Building Automation System. 1–4, 14, 24, 25, 27–32, 35, 36, 55, 58, 73–75

CoAP Constrained Application Protocol. 25

EHS European Home System. 14

EIB European Installation Bus. 14

ETS Engineering Tool Software. 16–18, 26, 36, 39, 42, 43, 45, 77

EXI Efficient XML Interchange. 13, 25

H2M Human-to-Machine. 8, 18

HTTP Hypertext Transfer Protocol. 9, 10, 13, 14, 21, 25, 27, 32, 33, 38, 56, 63, 66, 71, 74, 75, 78

HTTPs HTTP Secure. 75

IoE Internet of Energy. 10

IoT Internet of Things. xv, 1, 2, 4, 7–10, 15, 25

IRI Internationalized Resource Identifier. 18, 29

IT Information Technology. 15

JSON Javascript Object Notation. 10, 13

KNX Konnex. 4, 14–17, 25, 26, 35, 36, 39, 43, 44, 67, 68, 77

M2M Machine-to-Machine. 8–10, 18, 36

OASIS Organization for the Advancement of Structured Information Standards. 10

oBIX open Building Information eXchange. xv, 1–4, 10–13, 15–18, 25–36, 38–58, 61–66, 70–75, 77, 78, 80

OSGi Open Service Gateway Initiative. 25

OWL Web Ontology Languagy. 2–5, 21–23, 26–33, 35, 39, 41–44, 58, 59, 61–64, 66, 73, 74, 77

RDF Resource Description Framework. 3, 4, 19–22, 38, 56, 59–62, 70, 74, 77

REST Representational State Transfer. 10, 13, 16, 27, 29, 30, 71, 74

SHS Smart Home System. 23, 24

SILo Semantic Interoperability Layer for oBIX. xv, 2–4, 27, 28, 32–34, 56, 58, 59, 61–63, 66–68, 70–75, 77, 78

SOAP Simple Object Access Protocol. 13, 21

SPARQL SPARQL Protocol and RDF Query Language. 2–4, 8, 21, 22, 27, 32, 34, 52, 55, 58, 60, 63–66, 68–71, 74, 75, 78

SQL Structured Query Language. 21

SWoT Semantic Web of Things. 8, 9, 74

URI Universal Resource Identifier. 10, 13, 18, 29–31

URL Universal Resource Locator. 41, 45, 66

W3C World Wide Web Consortium. 18, 21

WS Web Services. 1, 2, 7, 10, 16, 29, 30

XML Extensible Markup Language. 4, 10, 13, 18, 19, 21, 27, 29, 56

XSD XML Schema Definition Language. 29

XSL Extensible Stylesheet Language. 3, 4, 29, 30, 35, 36, 39, 41, 42, 44, 45, 56–58, 73, 74, 80

XSLT XSL Transformation. 3, 4, 29, 31, 32

Bibliography

[1] Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. KNX and OPC UA. In *Konnex Scientific Conference*, November 2010.

[2] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[3] Payam Barnaghi, Wei Wang, Cory Henson, and Kerry Taylor. Semantics for the Internet of Things: Early progress and back to the future. *Int. J. Semant. Web Inf. Syst.*, 8(1):1–21, January 2012.

[4] IEA. Building - ES - International Energy Agency, 2013. [Last accessed 26-March-2015].

[5] O. Hersent. KNX. In *The Internet of Things: Key Applications and Protocols*. Wiley Publishing, 2011.

[6] Wolfgang Kastner, Lukas Krammer, and Andreas Fernbach. State of the Art in Smart Homes and Buildings. In Richard Zurawski, editor, *Industrial Communication Technology Handbook, Second Edition*, chapter 55. CRC Press, Inc., 2014.

[7] OBIX Version 1.1. OASIS Committee Specification 01. [Last accessed 26-September-2016].

[8] Wolfgang Kastner, Andreas Fernbach, Wolfgang Granzer, and Markus Jung. KNX and the Semantic Web of Automation. In *Proceedings of the KNX Scientific Conference*, November 2014.

[9] OWL 2 Web Ontology Language Primer (Second Edition). W3C Recommendation, 2012. [Last accessed 26-March-2015].

[10] KNX Specifications. Konnex Assoc. Diegem, Belgium, 2014. Ver. 2.1.

[11] Resource Description Framework (RDF). W3C Recommendation, 2014. [Last accessed 26-March-2015].

[12] SPARQL. Query Language for RDF. W3C Recommendation, 2008. [Last accessed 26-March-2015].

[13] Markus Jung, Jomy Chelakal, Jürgen Schober, Wolfgang Kastner, Luyu Zhou, and Giang Ky Nam. IoTSyS: an integration middleware for the Internet of Things. In *Proceedings of the 4th International Conference on the Internet of Things (IoT 2014)*, Cambridge, MA, USA, October 2014. Demo abstract.

[14] OPC Foundation. OPC Unified Architecture (UA), 2008. [Last accessed 26-March-2015].

[15] Mario Kofler. *An ontology as shared vocabulary for distributed intelligence in smart homes*. PhD thesis, Vienna University of Technology, 2013.

[16] Semantic Sensor Network XG Final Report. W3C Incubator Group Report, 2011. [Last accessed 26-March-2015].

[17] Open Geospatial Consortium. Sensor Model Language (SensorML), 2014. [Last accessed 26-March-2015].

[18] Dennis Pfisterer, Kay Romer, Daniel Bimschas, Henning Hasemann, Manfred Hauswirth, Marcel Karnstedt, Oliver Kleine, Alexander Kroeller, Myriam Leggieri, Richard Mietz, Max Pagel, Alexandre Passant, Ray Richardson, and Cuong Truong. SPITFIRE: toward a semantic Web of Things. *Communications Maagazine, IEEE*, 58(11), October 2011.

[19] INFSO D.4 0. Networked Enterprise and RFID INFSO G.2 Micro and Nanosystems, in: Co-operation with the Working Group RFID of the ETP EPOSS, Internet of Things in 2020, Roadmap for the Future, Version 1.1, 2008. [Last accessed 26-September-2016].

[20] N. Bui, A. P. Castellani, P. Casari, and M. Zorzi. The internet of energy: a web-enabled smart grid system. *IEEE Network*, 26(4):39–45, July 2012.

[21] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W. David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page, Alexandre Passant, Amit Sheth, and Kerry Taylor. The {SSN} ontology of the {W3C} semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25 – 32, 2012.

[22] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-sparql: Sparql for continuous querying. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 1061–1062, New York, NY, USA, 2009. ACM.

[23] Javier D. Fernández, Miguel A. Martínez-Prieto, Claudio Gutiérrez, Axel Polleres, and Mario Arias. Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19(0), 2013.

[24] Antonio J. Jara, Alex C. Olivieri, Yann Bocchi, Markus Jung, Wolfgang Kastner, and Antonio F. Skarmeta. Semantic web of things: An analysis of the application semantics for the iot moving towards the iot convergence. *Int. J. Web Grid Serv.*, 10(2/3):244–272, April 2014.

[25] N. Bui, A. P. Castellani, P. Casari, and M. Zorzi. The Internet of Energy: a web-enabled smart grid system. *IEEE Network*, 26(4):39–45, July 2012.

[26] Stamatis Karnouskos. The cooperative Internet of Things enabled smart grid. In *Proceedings of the 14th IEEE international symposium on consumer electronics (ISCE2010), June*, pages 07–10, 2010.

[27] B Initiativ. Internet of energy-ict for energy markets of the future, 2008.

[28] ASHRAE. Proposed Addendum C to Standard 135-2004, BACnet, 2004. [Last accessed 26-September-2016].

[29] Internet Engineering Task Force. RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax, January 2005. `http://www.ietf.org/rfc/rfc3986.txt`. [Last accessed 26-September-2016].

[30] Bindings for OBIX: REST Bindings Version 1.0. Edited by Craig Gemmill and Markus Jung. 14 September 2015. OASIS Committee Specification 01. [Last accessed 26-September-2016].

[31] Bindings for OBIX: SOAP Bindings Version 1.0. Edited by Markus Jung. 14 September 2015. OASIS Committee Specification 01. [Last accessed 26-September-2016].

[32] Bindings for OBIX: WebSocket Bindings Version 1.0. Edited by Matthias Hub. 14 September 2015. OASIS Committee Specification 01. [Last accessed 26-September-2016].

[33] KNX Basics. `http://www.knx.org/media/docs/Flyers/KNX-Basics/KNX-Basics_en.pdf`. [Last accessed 26-September-2016].

[34] M. Neugschwandtner, G. Neugschwandtner, and W. Kastner. Web services in building automation: Mapping KNX to oBIX. In *2007 5th IEEE International Conference on Industrial Informatics*, volume 1, pages 87–92, June 2007.

[35] KNX System Specifications - Web Services. Konnex Assoc. Diegem, Belgium, 2016. Draft.

[36] Bovet, Gerome and Hennebert, Jean. Introducing the Web-of-Things in Building Automation: A Gateway for KNX installations. In *10th international Conference on Informatics in Control, Automation and Robotics (ICINCO 2013)*, 2013.

[37] Daniel Schachinger and Wolfgang Kastner. Model-driven integration of building automation systems into Web service gateways. In *Factory Communication Systems (WFCS), 2015 IEEE World Conference on*, pages 1–8. IEEE, 2015.

[38] Andreas Fernbach, Wolfgang Granzer, Wolfgang Kastner, and Paul Furtak. Mapping ETS4 Project Structure to OPC UA using ETS4 XML Export. In *KNX Scientific Conference*, November 2012.

[39] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.

[40] Nigel Shadbolt, Tim Berners-Lee, and Wendy Hall. The semantic web revisited. *IEEE intelligent systems*, 21(3):96–101, 2006.

[41] Dean Allemang and James Hendler. *Semantic web for the working ontologist: effective modeling in RDFS and OWL*. Elsevier, 2011.

[42] M. Dürst; M. Suignard. Internationalized Resource Identifiers (IRIs). January 2005. RFC. `http://www.ietf.org/rfc/rfc3987.txt`. [Last accessed 26-September-2016].

[43] W3C. Extensible Markup Language (XML). `http://www.w3.org/XML`, 2015. [Last accessed 1-October-2016].

[44] RDF 1.1 Turtle. W3C Recommendation. 25 February 2014. `https://www.w3.org/TR/turtle/`. [Last accessed 26-September-2016].

[45] RDF 1.1 N-Triples. W3C Recommendation. 25 February 2014. `http://www.w3.org/TR/n-triples/`. [Last accessed 26-September-2016].

[46] RDF Schema 1.1. W3C Recommendation. 25 February 2014. `http://www.w3.org/TR/rdf-schema/`. [Last accessed 1-October-2016].

[47] Grigoris Antoniou and Frank Van Harmelen. *A semantic web primer*. MIT press, 2004.

[48] W3C. SPARQL 1.1 Update. W3C Recommendation. 21 March 2013. `http://www.w3.org/TR/sparql11-update/`, 2015. [Last accessed 1-October-2016].

[49] Pellet. Pellet, 2016. [Last accessed 26-March-2016].

[50] Mario Kofler, Christian Reinisch, and Wolfgang Kastner. A semantic representation of energy-related information in future smart homes. *Energy and Buildings*, 47:169–179, 2012.

[51] Green building xml (gbxml) schema. green building xml (gbxml) schema, inc. `http://www.gbxml.org`, 2016. [Last accessed 26-September-2016].

[52] M. J. Kofler and W. Kastner. A knowledge base for energy-efficient smart homes. In *Energy Conference and Exhibition (EnergyCon), 2010 IEEE International*, pages 85–90, Dec 2010.

[53] ThinkHome Website.Institute of Computer Aided Automation. Automation Systems Group. https://www.auto.tuwien.ac.at/projectsites/thinkhome/ontologies.html. [Last accessed 1-October-2016].

[54] M. Jung et al. Github iotsys code. https://github.com/mjung85/iotsys, 2015. [Last accessed 26-September-2016].

[55] Markus Jung, Jürgen Weidinger, Christian Reinisch, Wolfgang Kastner, Cedric Crettaz, Alex Olivieri, and Yann Bocchi. A transparent ipv6 multi-protocol gateway to integrate building automation systems in the Internet of Things. In *Green Computing and Communications (GreenCom), 2012 IEEE International Conference on*, pages 225–233. IEEE, 2012.

[56] Constrained application protocol (coap).draft-ietf-core-coap-18. https://tools.ietf.org/html/draft-ietf-core-coap-18, 2013. [Last accessed 26-September-2016].

[57] Hannes Bohring and Sören Auer. Mapping XML to OWL Ontologies. *Leipziger Informatik-Tage*, 72:147–156, 2005.

[58] XML Schema Definition Language (XSD) 1.1. W3C Recommendation. 5. April 2012. https://www.w3.org/TR/xmlschema11-1/. [Last accessed 04-October-2016].

[59] E Paslaru Bontas, Malgorzata Mochol, and Robert Tolksdorf. Case studies on ontology reuse. In *Proceedings of the IKNOW05 International Conference on Knowledge Management*, volume 74, 2005.

[60] Apache. Jena. https://jena.apache.org, 2016. [Last accessed 26-March-2016].

[61] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys (CSUR)*, 15(4):287–317, 1983.