

A FLEXIBLE SEMANTIC SERVICE COMPOSITION FRAMEWORK
FOR PERVASIVE COMPUTING ENVIRONMENTS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

MUSTAFA ÖZPINAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

AUGUST 2014

**A FLEXIBLE SEMANTIC SERVICE COMPOSITION FRAMEWORK
FOR PERVASIVE COMPUTING ENVIRONMENTS**

Submitted by **MUSTAFA ÖZPINAR** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems, Middle East Technical University** by,

Prof. Dr. Nazife Baykal
Director, **Informatics Institute**

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department, **Information Systems**

Assist. Prof. Dr. P. Erhan Eren
Supervisor, **Information Systems, METU**

Examining Committee Members:

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems, METU

Assist. Prof. Dr. P. Erhan Eren
Information Systems, METU

Prof. Dr. Nazife Baykal
Information Systems, METU

Assoc. Prof. Dr. Aysu Betin Can
Information Systems, METU

Dr. Nail Çadallı
Karel A.Ş.

Date: 15.08.2014

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Mustafa Özpınar

Signature : _____

ABSTRACT

A FLEXIBLE SEMANTIC SERVICE COMPOSITION FRAMEWORK FOR PERVASIVE COMPUTING ENVIRONMENTS

Özpinar, Mustafa

M.Sc., Department of Information Systems

Supervisor: Assist Prof. Dr. P. Erhan Eren

August 2014, 66 pages

With the advances in technology, high-speed connections, powerful and low cost devices have become available. It is estimated that there will be tens of billions of devices connected to the Internet by 2020. However, for the effective use of such an outstanding number of devices, they should be able to communicate with each other in different scenarios. A commonly agreed structure should be adopted to overcome the communication problem of heterogeneous devices. Web of Things (WoT) is a vision about quickly connecting devices and services by reusing the Web standards. In this way, the communication protocol is provided, but service definition, composition and resolving the meaning of data stay as challenging problems. The goal of this thesis study is to investigate the area of semantic service composition, to propose a composition architecture, and to design a proof-of-concept system using heterogeneous networked devices. Accordingly, a lightweight ontology is constructed in order to define services semantically. A service registry solution considering WoT constraints is presented for publishing and discovering services, and a rule-based flexible semantic service composition framework is proposed for composing services to achieve a goal. A proof-of-concept system including some features of the proposed solution is implemented on real devices in order to assess its feasibility. In

addition, the proposed solution is compared with existing frameworks by examining scenarios and features.

Keywords: Semantic Service Composition Framework, Ontology, Web of Things

ÖZ

YAYGIN BİLİŞİM ORTAMLARI İÇİN ESNEK SEMANTİK SERVİS BİRLEŞTİRME ÇERÇEVESİ

Özpınar, Mustafa

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Yrd. Doç. Dr. P. Erhan Eren

Ağustos 2014, 66 sayfa

Teknolojinin ilerlemesiyle birlikte, yüksek hızlı bağlantılar, güçlü ve düşük maliyetli cihazlar ortaya çıktı. 2020’de İnternet’e bağlı on milyarlarca cihazın olacağı tahmin edilmekte. Bununla birlikte, bu kadar fazla cihazın daha etkin kullanılabilmesi için, bu cihazların farklı senaryolar dahilinde haberleşebilmeleri gerekiyor. Farklı yapıdaki cihazların haberleşme problemlerinin üstesinden gelebilmek için üzerinde anlaşılmış ortak yapılar benimsenmelidir. Nesnelerin Ağı (Web of Things), cihazları ve servisleri Web standartlarını yeniden kullanarak hızlıca birbirleriyle haberleştirme vizyonudur. Bu şekilde haberleşme protokolü çözülmüş oluyor fakat servis tanımlama, birleştirme ve verinin anlamının çözümlenmesi hala zorlu bir problem olarak duruyor. Bu tez çalışmasının amacı; semantik servis birleştirme alanını araştırmak, bir birleştirme mimarisi ortaya koymak ve farklı yapıdaki birbirine bağlı cihazların kullanıldığı çalışabilir bir sistem tasarımı yapmaktır. Servisleri semantik olarak tanımlayabilmek için karmaşık olmayan bir ontoloji oluşturuldu. Servislerin yayınlanması ve keşfedilmesi için Web of Things kısıtları düşünülerek bir servis kayıt çözümü sunuldu. Bir amaç dahilinde servisleri birleştirebilmek için kural tabanlı, esnek, semantik servis birleştirme çerçevesi ortaya konuldu. Çözüme ait bazı özellikleri içeren çalışabilir bir sistem gerçek cihazlar üzerinde gerçekleştirildi. Önerilen çözüm senaryo ve özelliklerin incelenmesiyle varolan çerçevelerle karşılaştırıldı.

Anahtar Kelimeler: Semantik Servis Birleřtirme erevesi, Ontoloji, Nesnelerin Ađı

To My Love and Daughter

ACKNOWLEDGEMENTS

First of all, I would like to thank to Assist. Prof. Dr. P. Erhan Eren for his supervision, motivation and guidance through the development of this thesis.

I also want to express my gratefulness to my wife Sevde for all her patience, love, support and tolerance. Without her understanding this work would never been possible. To my sweet daughter, Elif Beyza, I thank her for understanding my frequent absences.

TABLE OF CONTENTS

ABSTRACT	vi
ÖZ	viii
DEDICATION	x
ACKNOWLEDGEMENTS	xi
TABLE OF CONTENTS	xii
TABLE OF FIGURES	xv
CHAPTER	
1. INTRODUCTION	1
1.1. Motivation	1
1.2. Scope of the Thesis	3
1.3. Our Approach	3
1.4. Contributions	3
1.5. Thesis Organization	4
2. RELATED WORK	5
2.1. Advantages of Semantic Service Definition	5
2.2. Ontology	7
2.3. Service Definition Methods : WSDL & RESTful Services	7
2.4. Linked Data and Linked Open Services	8
2.5. Why is Composition Important?	9
2.6. Existing Composition Frameworks	9
2.6.1. iServe [19, 27]	9

2.6.2. Yahoo Pipes [28]	10
2.6.3. Microsoft Popfly [29]	11
2.6.4. ClickScript [6]	11
2.6.5. SOA4All [5]	12
2.6.6. Amigo [30]	12
2.6.7. Flexible Composition of Smart Device Services [31].....	12
2.6.8. EasyApp [4]	12
2.6.9. Sense2Web [7].....	13
2.6.10 SensorMasher [9].....	13
2.6.11. Smart Home [12].....	13
2.6.12. A Goal-Based Service Framework [3].....	13
3. PROPOSED FRAMEWORK.....	15
3.1. Service Definition	17
3.2. Endpoint Repository.....	20
3.3. Composition Framework.....	22
4. PROTOTYPE IMPLEMENTATION	29
4.1. Scope of Prototype.....	29
4.2. Composition Framework.....	31
4.2.1. Implementation of Server Side.....	31
4.2.2. Implementation of Client Side.....	34
4.2.3. Implementation of Shared Part.....	35
4.2.4. Graphical User Interface (GUI) of Composition Framework.....	35
4.3. Android Application.....	42

4.4. Arduino Application.....	42
5. EVALUATION AND RESULTS	45
5.1. Feature-Based Comparison	45
5.2. Scenario-Based Comparison.....	46
6. CONCLUSION.....	49
REFERENCES.....	51
APPENDICES.....	55
A. EASYLIFE ONTOLOGY	55
B. SAMPLE SERVICE DEFINITION FILE	59

TABLE OF FIGURES

FIGURE

1. “Knowledge Hierarchy” in the context of IoT [2]	2
2. Semantics at different levels in IoT [2].....	6
3. Evolution of the Web [25]	8
4. iServe Framework [27]	10
5. A sample mashup with Yahoo Pipes	11
6. Main components of the Goal-Based Framework	14
7. Service platform architecture	15
8. Architecture of EasyLife	16
9. RESTful method declaration.....	20
10. Event declaration	20
11. Basic composition scenario	22
12. Definition of goal, task and service [3].....	26
13. Sample goal, task and service definition.....	27
14. Sample class inheritance	28
15. Components of prototype implementation.....	30
16. Server and client architecture of EasyLife.....	31
17. Method based composition	33
18. Event based composition	34
19. EasyLife composition framework GUI	36
20. Tools pane in composition framework	36
21. Endpoint services in composition framework.....	37
22. Event in composition.....	37
23. Defining parameter	38
24. Processing over parameter	38
25. Process types over parameters.....	38
26. Sample composition.....	39

27. Sample composition	40
28. Sample composition	41
29. Rule creation	41
30. Android application	42
31. Arduino board setup	43
32. Framework comparison based on features	45
33. Required features and comparisons for Scenario 1	47
34. Required features and comparisons for Scenario 2	48

CHAPTER 1

INTRODUCTION

1.1. Motivation

“The number of transistors on integrated circuits doubles approximately every two years” says Moore’s Law. So far, this estimation seems to be accurate over the years. Microcontrollers are used in TV sets, remote controls, video recorders, car electronics, scanners, washing machines, mobile phones, and other devices. It is estimated that there will be around 25 billion devices connected to the Internet by 2015 and 50 billion by 2020 [1]. Such a stunning number of highly distributed and heterogeneous devices will need to interconnect and communicate in different scenarios autonomously.

Internet of Things refers to the uniquely identifiable objects and their virtual representations in an Internet-like structure. Things in such an environment may communicate each other with different protocols such as Bluetooth, RFID, HTTP, UDP etc.

Web of Things is a vision inspired from Internet of Things where everyday objects equipped with microcontrollers such as wireless sensor networks, household appliances, RFID tagged objects are connected by over the Web. Web of Things is about reusing the Web standards (URI, HTTP, REST, etc.) to connect quickly, expanding the ecosystem of devices and services built into everyday smart objects.

Exponentially growing raw data are not valuable unless processed, analyzed and turned into actionable knowledge. As a result, composition of distributed services and interconnecting devices is a prominent field in Web of Things. Fusion of data creates situation awareness and enables applications, machines and human users to better understand their surrounding environments. Understanding of a situation, or context, potentially enables services and applications to make intelligent decisions and respond to the dynamics of their environments. Figure 1 depicts the evolution of raw data. It gains structure when annotated with semantics. The structure becomes important when the produced data are intended to be consumed by different systems. Existence of commonly agreed structures (or semantic annotations) makes data exchange easier. With semantics, the meaning of data is clear to stakeholders. After intelligent analysis of semantic data, actionable knowledge can be produced. For example, readings from meters can be used to better predict and balance power consumption in smart grids; analyzing combination of traffic, pollution, weather and congestion sensory data records can provide better traffic and city management; monitoring

and processing sensory devices attached to patients or elderly can provide better remote healthcare.

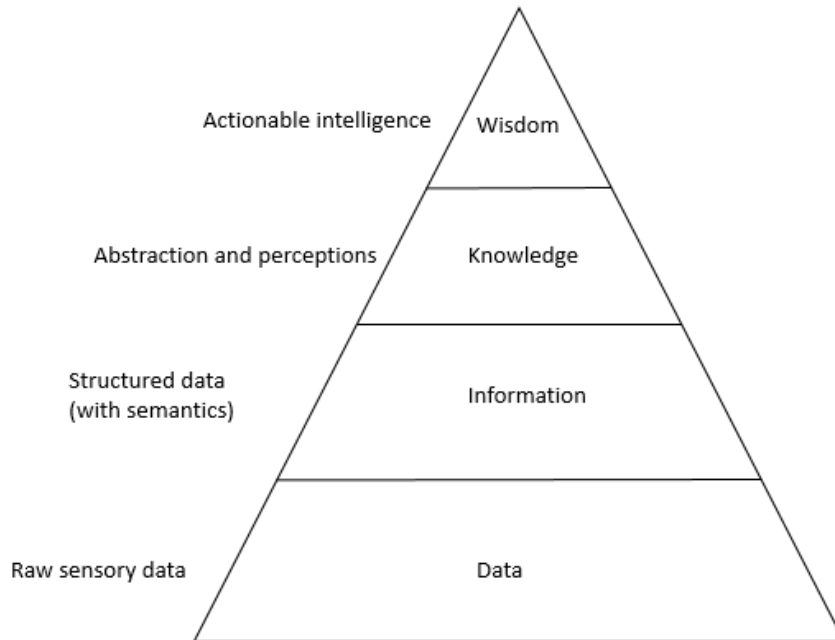


Figure 1: "Knowledge Hierarchy" in the context of IoT [2]

All of these scenarios are possible with an infrastructure that has the ability of searching, discovering and composing distributed services around a purpose. Service composition provides the composer for achieving high level and complex goals by combining services of sub goals. Event support in such an infrastructure is a powerful feature that contributes to gain the following high level knowledge about the situation:

- Anomaly detection,
- Situation awareness
- Pattern recognition

In the domain of Web of Things, such intelligent decisions depend on the usage and composition of different data sources and distributed data providing services. However, "Things" may produce structured or raw data. Some of them may be directly connected to network, while others use proxies. Communication protocols or technologies may also differ among them. Things may also be resource-constrained devices or powerful machines. The heterogeneity makes interoperability among them a challenging problem, which prevents generic solutions from being adopted on a global scale.

The focus of this thesis is identifying the problems and providing solutions for the composition of services in the domain of Web of Things. The main goal of the thesis is to produce a semi-automatic flexible composition framework which can help improve situation awareness and can be adopted easily.

1.2. Scope of the Thesis

In this thesis, it is aimed to develop a composition framework in the domain of Web of Things that has the ability of integrating services. One of the most challenging concerns in this domain is the heterogeneity of billions of devices. In addition, resource-constrained devices should be taken into account. Therefore, providing a solution that applies to definition of services is one of the requirements of this thesis.

Service composition frameworks should be able to search and discover existing services. Finding meaningful and exact results in search process is possible with semantics. Therefore another requirement of the thesis is semantics based search.

Events are primary actors for composition. However, they are only valuable in some context. Users of this system should be able to create rules about events. Therefore rule definition is also a requirement of the thesis.

As a result, scope of the thesis is defined as providing service definition, searching and discovering services, and a composition framework with rule definition ability.

1.3. Our Approach

In this thesis, ontology enabled smart services are defined and consumed by a generic composition framework for achieving a user defined goal. Ontology promotes semantic definition of services and removes ambiguities. A generic ontology is needed for communication standards.

Our overall system has the following parts:

- First, services are defined with ontologies.
- Second, service providers are registered to endpoint repositories.
- Third, composition framework searches endpoint repositories.
- Finally, composition framework enables user to create compositions and run them.

1.4. Contributions

The main contributions of this thesis are as follows:

- A generic service definition ontology is defined and advantages of this ontology are examined in detail.
- A powerful composition framework with the following abilities is created:
 - consume semantically described services
 - linked data support
 - rule definition support.
 - semantic search support.
 - context aware
- Distributed endpoint repositories are provided. These repositories support semantic search queries.

1.5. Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 presents related work on Web of Things and service composition. Ontology based service definition is compared with traditional service definition. Composition approaches are discussed and details of some example works are explained to address where our work stands in the literature.

In Chapter 3, the conceptual design of the proposed service composition framework is presented.

Chapter 4 provides information about prototype implementation.

In Chapter 5, evaluation of the proposed solution is presented by comparing with existing frameworks. Some possible real-life scenarios are also explained in this chapter.

Chapter 6 concludes the study by giving a summary of the work done. It also mentions possible future work to guide researchers in this area.

CHAPTER 2

RELATED WORK

Related work can be classified into semantic service definition, ontology, service definition methods, linked data and existing composition frameworks.

2.1. Advantages of Semantic Service Definition

Semantics is the study of meaning. It allows defining high-level abstractions on description based on commonly accepted ontological schema so that machines and humans can interpret links and relations between different attributes [7,12]. Semantic annotation of data (for example, with domain knowledge) can provide machine-interpretable descriptions on what the data represents, where it originates from, how it can be related to its surroundings, who is providing it, and what are the quality, technical, and non-technical attributes [2]. Utilizing and reasoning semantic information enables integration of data on a wider scale, known as networked knowledge [7]. Semantic data supports reasoning and inference by incorporating entailment rules in expressive representation. These attributes make semantic data amenable for flexible and complex manipulation, thus enabling many advanced processing capabilities such as automated processing and knowledge discovery, and novel application scenarios such as data sharing, reuse, integration, and situation aware assistance [12]. In [2], semantic definition through machine perception from IoT data is stated as key enabler for developing situation-aware applications that can intelligently respond to changes in real world.

Study at [2] highlights the benefits of semantic annotation and interoperability. Semantic technologies based on machine-interpretable representation formalism have shown promise for describing objects, sharing and integrating information, and inferring new knowledge together with other intelligent processing techniques.

Addition of semantics has also helped create machine-interpretable and self-descriptive data in IoT domain. However, dynamic and resource-constrained nature of IoT requires special design considerations to be taken into account to effectively apply semantic technologies on real world data.

The term “Semantic interoperability” is defined in [2] as “different stakeholders can access and interpret the data unambiguously”. Networked objects on IoT, or the “Things”, need to exchange data among each other and with other users. Automated interactions in IoT depend

on unambiguous data descriptions provided in a way that can be processed and interpreted by machines and software agents. Figure 2 gives information about semantics at different levels in IoT. At the bottom the real world objects (i.e. “Things”) reside. At the next level, data provided by objects are semantically annotated. Machines can interpret high-level abstractions and serve data as services or to applications.

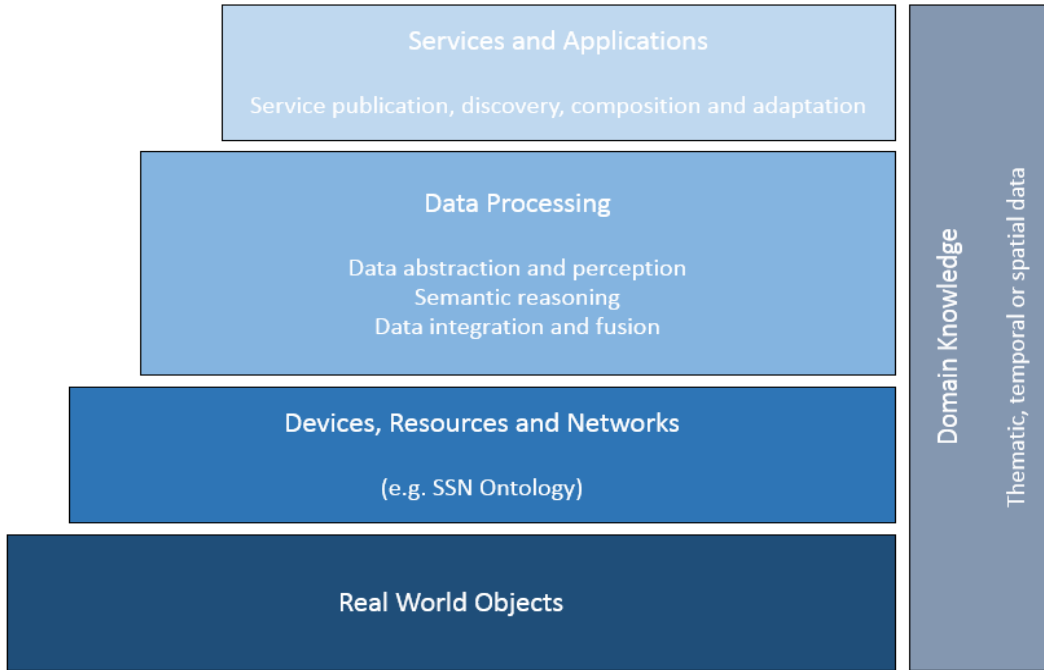


Figure 2: Semantics at different levels in IoT [2]

The main problem of Smart Home (SH) technologies is identified in [12] as the missing commonly agreed self-descriptive data model at both data and application levels. Data heterogeneity hinders seamless exchange, integration and reuse of data. Application heterogeneity disallows reuse of middleware services in different scenarios without support of formal data models. Lack of semantics and inability of data sharing and integration reduce the potential to carry out deep, intelligent data analysis and knowledge discovery from multiple data sources, such as trend discovery, pattern recognition and knowledge-based decision making. This ultimately leads to difficulty of developing and deploying systematic SH solutions with seamless data integration and advanced high-levels of intelligent capabilities.

2.2. Ontology

Ontology is used to denote the types, properties and relations using a shared vocabulary. If a commonly-agreed ontology is defined and used by stakeholders, data can be interoperable without ambiguity. To achieve global scale semantic interoperability, common semantic annotation frameworks, ontology definitions, and adaptation are key issues [2].

Considering the resource constrained environments in WoT, it is a requirement for ontology to be simple and lightweight [2, 8]. Designing lightweight semantic description models [15] and effective representation frameworks such as Binary RDF Representation [16] are some of the recent works that can provide effective semantic data representations for IoT domain. The complexity involved in describing semantic web services with ontologies (i.e. such as OWL-S [17] or WSMO [18]) has hindered the widespread adoption of comprehensive semantic models in IoT. Looking at the future prospect of using semantics in IoT domain, lightweight and easy-to-use ontologies seem to have a better chance of being widely adopted and reused in order to create an interoperable platform across different domains and applications.

2.3. Service Definition Methods : WSDL & RESTful Services

Currently, services on web are in the form of two main groups: on the one hand “classical” web services based on WSDL and SOAP, on the other hand RESTful services [19]. WSDL is used to provide structured descriptions for services, operations and endpoints, while SOAP is used to wrap XML messages exchanged between service consumer and provider. A large number of additional specifications such as WS-Addressing, WS-Messaging and WS-Security complement the stack of technologies. On the other hand, an increasing number of popular Web and Web 2.0 applications as offered by Facebook, Google, Flickr and Twitter offer easy-to-use, publicly available Web APIs, also referred to as RESTful services (properly when conforming to the REST architectural principles [20]). RESTful services are centered around resources, which are interconnected by hyperlinks and grouped into collections, whose retrieval and manipulation is enabled through a fixed set of operations commonly implemented by using HTTP. In contrast to WSDL-based services, Web APIs build upon a light technology stack relying almost entirely on the use of URIs, for both resource identification and interaction, and HTTP for message transmission.

Study at [21] identifies disadvantages of WSDL. WSDL description is used to generate module source code automatically, which is then compiled into a larger program. If description changes, the program no longer works, even if such a change leaves the functionality intact. A concrete example of such brittleness is the switch from 32 to 64 bit integer identifiers that occurred at some point in Google’s AdWords API, a small change in the service’s WSDL file that required the complete recompilation of the relevant pieces of source code [22]. This indicates that WSDL is not well adapted to real-world circumstantial changes. Therefore, WSDL cannot offer automatic service discovery at runtime and this is why we should investigate other possibilities.

In RESTful APIs, resources are identified by URIs [23]. A resource is to be differentiated from its representation. For example, a set of RDF triples (the resource) might be represented in different serializations (syntaxes), such as RDF/XML or Turtle. Manipulation of any of the representations should carry sufficient information to manipulate

the original resource. All messages need to be self-descriptive, for example, media type of a message needs to make clear what can be done with this message. Each representation needs to communicate relevant related resources, or next steps the client can take at each state.

2.4. Linked Data and Linked Open Services

Linked Data (Web of Data) describes a method of publishing structured data so that it can be interlinked and become more useful. It shares information over HTTP as RDF and enables querying over data. Figure 3 shows the evolution of web and position of Linked Data.

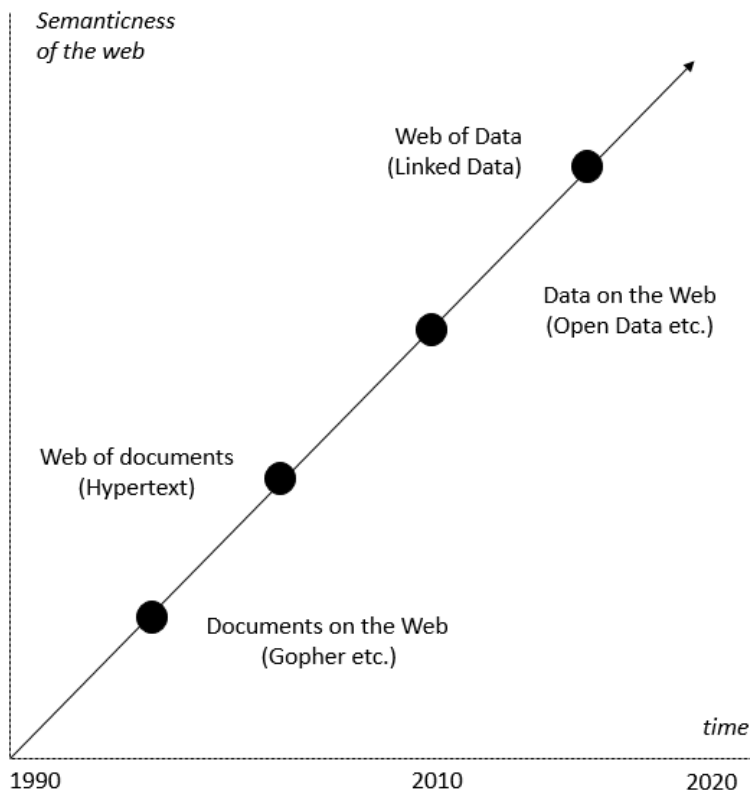


Figure 3: Evolution of the Web [25]

Main problem with current data is that they are tightly coupled with applications. However, application and data should be independent. Main benefit of Linked Data is that data are managed and updated independently from consumer. No change is required on the consumer side either content or links of data are updated.

Web of Data is based upon four simple principles, known as the Linked Data Principles, which are [2, 24]:

- Use URIs (Uniform Resource Identifiers) as names for things.
- Use HTTP URIs so that people can look up those names.

- When someone looks up a URI, provide useful information, using standards (RDF*, SPARQL).
- Include links to other URIs, so that they can discover more things.

Using URIs to identify things removes ambiguity and ensures that everybody points to the same thing with the same URI. Relations with other data on the cloud enable high-level and useful inferences.

The Linked Open Service (LOS) Principles [25] encourage the following:

- allowing RDF-encoded messages for input/output;
- reusing URIs from Linked Data source for representing features in input and output messages;
- making the semantic relationship between input and output explicit

2.5. Why is Composition Important?

Semantics is a great step towards global interoperability between services but is not enough alone. Discovery, management of data and supporting autonomous interactions have to be solved as well [2]. Semantic annotations do not eliminate the key role of information analytics and intelligent methods, which can process and interpret data and create meaningful abstractions. In real world, useful information can only be acquired by combination of two or more services. Considering a security service at home, an opened window does not mean anything alone, but it does by combining with the location of inhabitant.

Fusion of data by event processing and intelligent data processing methods provides composer with the following high level knowledge about situation:

- Anomaly detection,
- Situation awareness
- Pattern recognition

2.6. Existing Composition Frameworks

2.6.1. iServe [19, 27]

iServe is defined in [27] as “service warehouse which unifies service publication, analysis, and discovery through the use of lightweight semantics as well as advanced discovery and analytic capabilities”. Most important feature of iServe is that it can annotate traditional web services and publishes Semantic Web Services as Linked Data. As can be understood from Figure 4, WSDL-based service definitions are wrapped with annotations based on some ontologies (SAWSDL, WSMO-Lite, MicroWSMO or OWL-S) and enabled to consume and produce Linked Data. By this way, advanced service analysis and discovery techniques are enabled. iServe provides a semantic annotation tool. RDF data can be accessed via RESTful API, web browser and as linked data. Semantic queries are also supported by iServe platform.

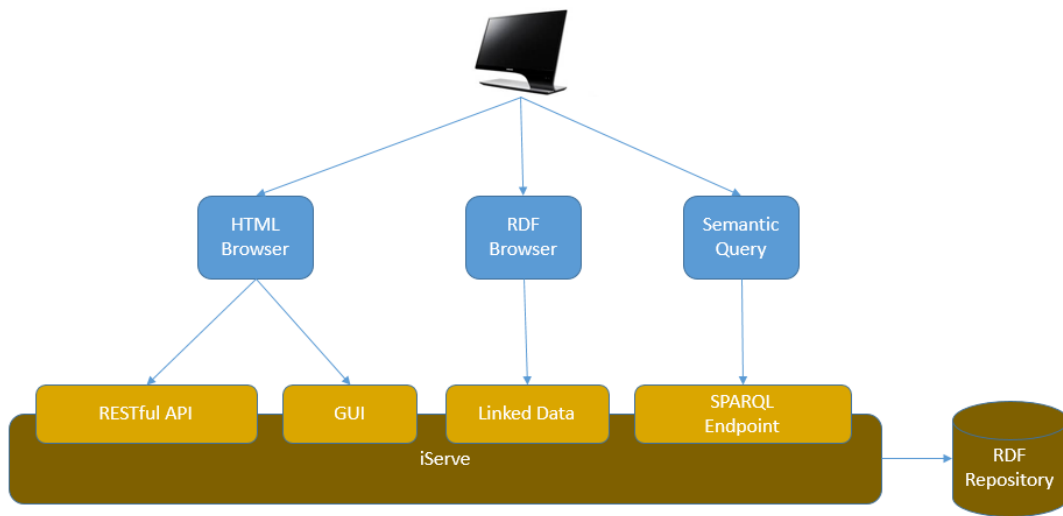


Figure 4: iServe Framework [27]

iServe does not provide a service composition infrastructure. It aims to serve data and services with semantic descriptions.

2.6.2. Yahoo Pipes [28]

Yahoo Pipes is a powerful composition tool to aggregate, manipulate, and mashup content from around the web. Framework has the ability to combine many feeds into one, then sort, filter and translate it.

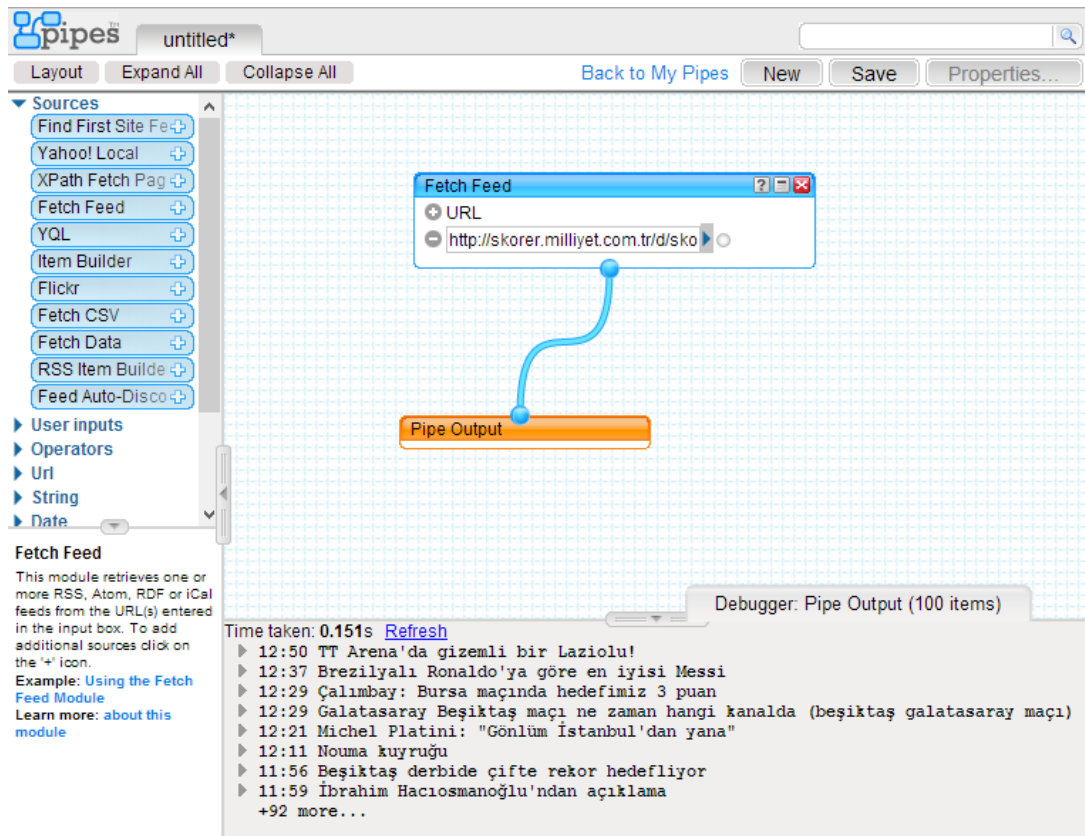


Figure 5: A sample mashup with Yahoo Pipes

Yahoo Pipes does not enable users to discover services other than provided by the framework. Since services are not semantically annotated, it is not a promising application in the context of Web of Things.

2.6.3. Microsoft Popfly [29]

Microsoft Popfly was a Web site that allowed users to create web pages, program snippets, and mashups using Microsoft Silverlight rich internet applications runtime and a set of online tools provided. It was discontinued on August 24, 2009. As with Yahoo Pipes, Popfly does not enable users to discover services other than provided by the framework too.

2.6.4. ClickScript [6]

ClickScript project aims to create Web mashups by connecting web resources with some simple operations. It is a browser plugin written on top of the Dojo AJAX library and provides users with visual programming. Since it is written in JavaScript, ClickScript cannot use resources based on WS-* Web Services or low-level proprietary service protocols, but it can easily access RESTful services available on the Web. Thus, creation of ClickScript building blocks (or widgets) based on Web of Things devices is straightforward.

2.6.5. SOA4All [5]

In this European funded project, it is aimed to create a standard infrastructure to define, discover and analyze services. Services are semantically defined with RDF using many ontologies (Functional Classifications, Execution, Auditing Logs, eGovernment, hRESTS ontologies). RDF data is consumed and produced by services. For composition, a tool that can only be used by developers or expert users of system is provided. Composition framework is not simple enough for non-expert end users. No Linked Data integration and goal-based composition infrastructure exists in the framework. Since many ontologies are used to define services, users should know the details to create a composition or provide a new service to the environment.

2.6.6. Amigo [30]

As stated in project deliverables, the main goal of Amigo project is to merge traditionally separated domains of home automation, consumer electronics, mobile communications, and personal computing, to offer home individual residents user-friendly, intelligent, and meaningful interfaces to handle home information and services. In project, OWL-S extended ontologies are created in computer electronics, PC, mobile and domotic areas. Services are defined with these strict ontologies. Amigo is based on zero-configuration. It provides an infrastructure for well-defined services to be integrated without user intervention. Developers can develop new plugins for Amigo system. A conversation is a composition of service definitions in Amigo system. Conversation can be integrated into system and matching services are used in the configuration.

Amigo system does not enable user to create new compositions. The system is specific to Smart Home systems. Strict ontologies do not permit to insert new devices and services into the ecosystem.

2.6.7. Flexible Composition of Smart Device Services [31]

Article is mainly about the service composition based on task and executing the task according to dynamically changing context. If a task is created by selecting required service definitions, then real services can be selected by execution system and invoked accordingly. No implementation is provided for this solution.

2.6.8. EasyApp [4]

EasyApp is a goal-driven service flow generator based on semantic web service annotation and discovery technologies. The purpose of EasyApp is providing application creation environment for software programmers to make new application semi-automatically by enabling the semantic composition of web services on the Web.

At the beginning, user inputs a goal and finally gets a service flow satisfying the goal. Goal is analyzed and decomposed to sub-goals. Service discovery component finds relevant services and generates a template source code.

EasyApp is designed for software developers not for standard users. It outputs source code. Its success mainly depends on the number and variety of goal ontologies.

2.6.9. Sense2Web [7]

It provides a platform to annotate sensor data semantically with Semantic Sensor Network ontology and link to existing data. Sensor devices are connected to Sense2Web with a gateway. Annotated raw data is stored as RDF in SDB [32] and served by Apache Joseki Server [33]. For example, location of a temperature sensor is defined with Sense2Web. Then, sensors in a range can be queried from application.

2.6.10. SensorMasher [9]

SensorMasher uses linked data principals to make sensor data available on the web. Sensor data is annotated semantically and published as Web resources. Sensor data published in this platform can be accessed through SPARQL endpoints and RESTful services. Users can access data in JSON, XML, and RDF formats.

The main disadvantage of SensorMasher platform is sensor devices are directly connected to the platform. Semantic annotation process is done inside the platform automatically with no user intervention. This is a problem in Web of Things. Users cannot search and compose services outside the platform since no distributed infrastructure exists.

2.6.11. Smart Home [12]

Article is about defining Smart Home device data with ontologies since many types of devices exists. Semantically annotated data can be stored in a public store that applications are aware about.

It is stated that an ontology should be defined for every type of device such as medical devices, home electronics, mobile devices etc. This is not possible since so many devices exist and new ones are coming always.

2.6.12. A Goal-Based Service Framework [3]

A framework for dynamic service discovery and composition based on user goal is presented in the article. Client informs system about a goal and platform searches services according to client's context information. If required, services are composed according to domain ontologies and service composition is returned for the goal. At the end, user gets service or service composition able to fulfill the goal.

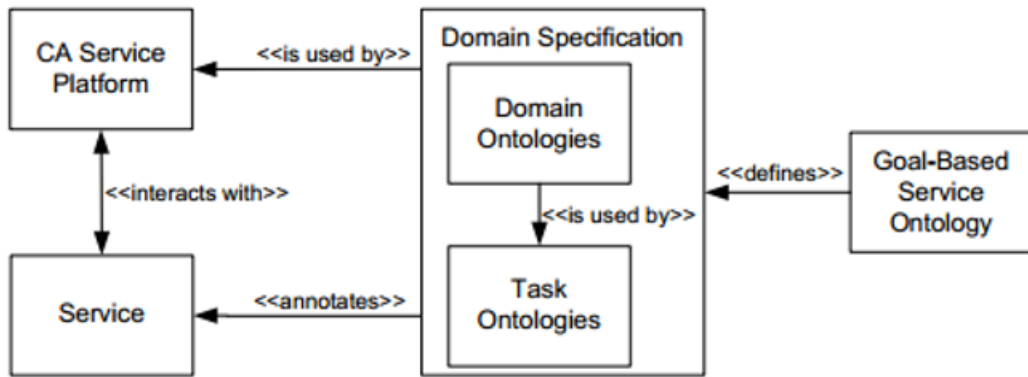


Figure 6: Main components of the Goal-Based Framework

No prototype exists for the framework.

CHAPTER 3

PROPOSED FRAMEWORK

With the penetration of mobile device usage in real life and cheaper and more powerful microcontrollers having access to the internet, abstraction of the raw data from sensors or devices is required to infer high-level, understandable knowledge. This inference is mainly based on the composition of services. In this chapter, conceptual design of the proposed solution named EasyLife is discussed in detail.

The major challenge in Web of Things is that large number of service clients and service providers are present in the environment. In such an environment, it becomes difficult for a client to manually match services with requests. A service platform is needed to enable service discovery, selection and composition activities.

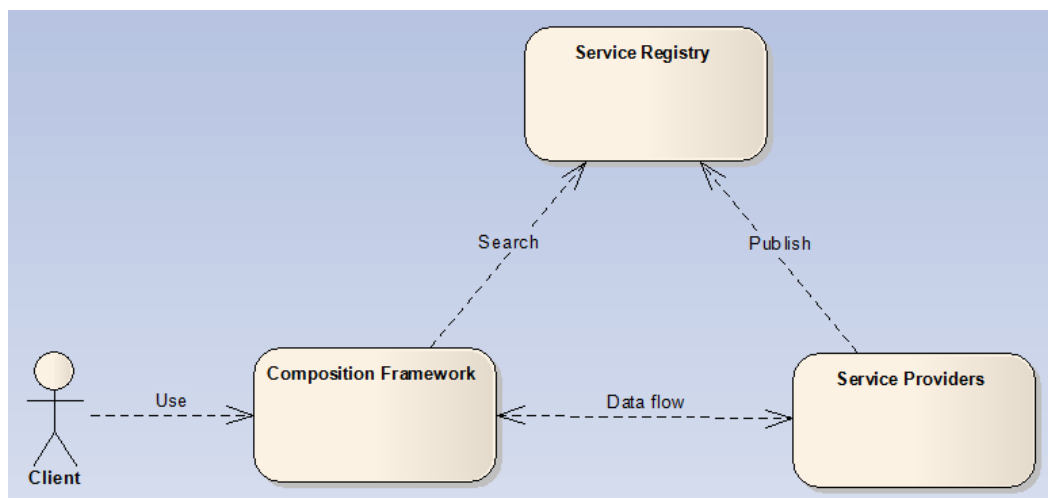


Figure 7: Service platform architecture

Figure 7 depicts the architecture of EasyLife platform. Since the platform uses services as backend, Service Oriented Architecture is adopted. Composition framework is the service consumer in architecture. Service providers define services and publish to a known service registry to be searched by the consumers.

In our framework, there are some features that define a service as “smart”. Figure 8 reveals these features and relations between participants of EasyLife.

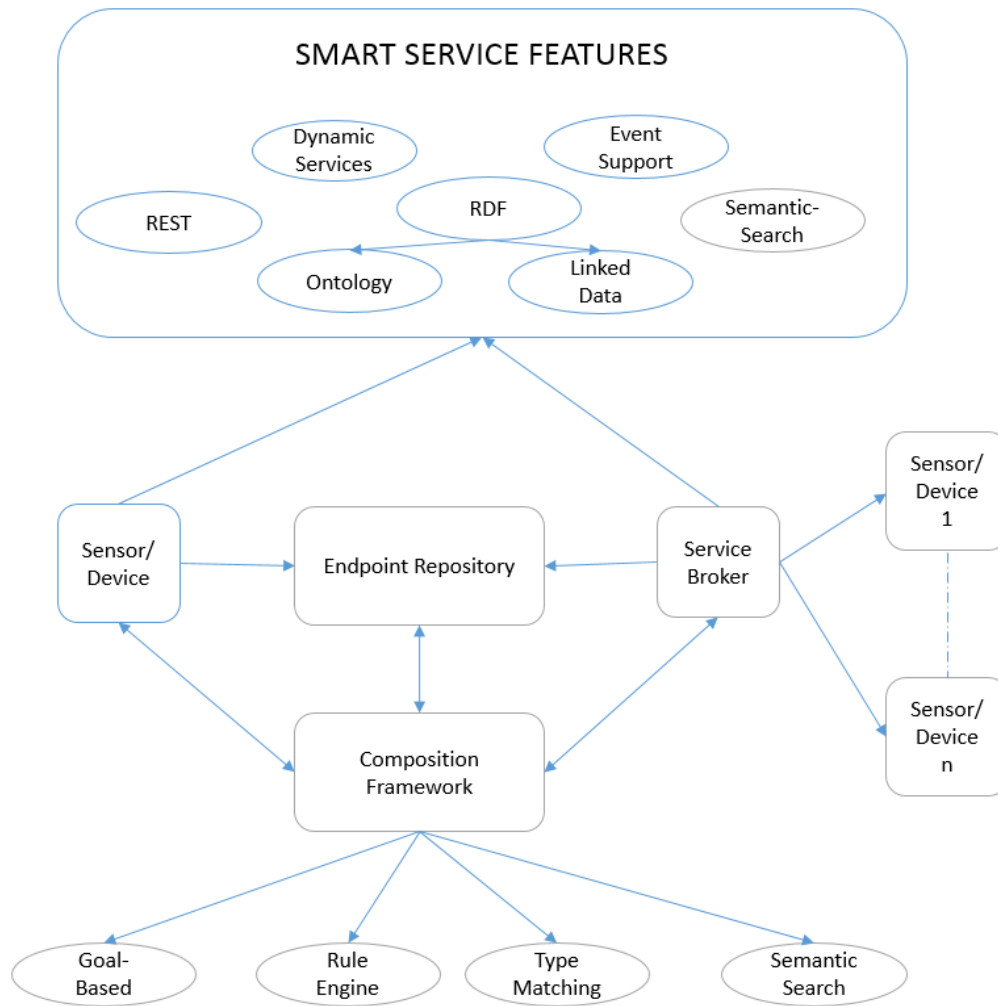


Figure 8: Architecture of EasyLife

At the center of the framework, the composition component resides. This component has a graphical user interface that the user interacts with. Endpoint repository is a simple structure that only contains endpoint URLs of the service providers. The composition framework also has communication with the endpoint repository to search services. The endpoint repository allows semantic search of services the composition framework needs while building compositions. When a composition is created, the framework uses participating “smart services” to carry out required operations. For example, a method can be called or a continuous data providing event can be bound to infer valuable knowledge in real time. A smart service either belongs to the sensor/device itself or to a broker. If a device has the ability of communicating with web standards and supports semantic operations, it can be directly used by the composition framework. In the latter case, the broker communicates with sensors using specific protocols and provides smart services and semantic data to the composition framework. Service consumer does not need to know about the smart service

infrastructure. If service is published using the same ontology and structure, it can get involved in the EasyLife system.

The rest of this chapter gives details about service definition, endpoint repository and composition framework.

3.1. Service Definition

Service definition is the first requirement for composition. Addition of semantics to service descriptions and messages exchanged between service clients, service providers and the platform enables complex reasoning tasks [36]. Assuming that a conceptual model is accepted through participants, semantic interoperability becomes possible. In our framework, we created a generic ontology (EasyLife Ontology) [APPENDIX A] as the conceptual model. Service providers define published services with this lightweight ontology. Our ontology consists of the minimum set of RDF classes to define a service. But owner of the service can link to different ontologies or linked data and embed into service definition document. These descriptions are used to guide composer about service and promote search operation.

OWL-S [17] is an ontology used to describe Semantic Web Services. It is constructed with Web Ontology Language (OWL) [14] which is a semantic markup language for publishing and sharing ontologies on the web. EasyLife Ontology is designed to be a small subset of OWL-S. Main difference between them is the complexity. OWL-S enables user to define more complex services. User can put restrictions, conditions on definitions. Atomic or simple processes can be described with OWL-S ontology. Sophisticated logical statements can also be embedded in service descriptions. As stated in the introduction and related work chapters, penetration of smart service usage and composition depend on the simplicity. Thus, our ontology omits complex parts and accepts fundamental features. Functionality definition (hasInput, hasOutput, hasParameter) in our ontology comes from OWL-S and is integrated into “Sensing Device” class. A major extension is that EasyLife Ontology provides event definition support. Service owners can define events for services and composers can use these events in their compositions.

A service is defined with the following EasyLife Ontology items:

- Sensing Device
 - Name
 - Device or service name
 - Events (if exists)
 - URL of the event
 - Comment
 - Output type
 - Methods (if exists)
 - URL of the method
 - Comment
 - Input (if exists)
 - Comment
 - Type
 - Order
 - Output type

Lightweight ontology enables the framework and service providers to be more flexible considering the followings:

- Adding new devices
- Adding new services
- No change required for infrastructure

Most defined ontologies are domain specific and restrictive. We can compare the EasyLife Ontology with domain specific ones based on a sample service definition.

Part of a sample service definition with domain specific ontology is as follows:

```
<TemperatureSensor>
  <GetCurrentValue>
    <returnType>integer</returnType>
  </GetCurrentValue>
</TemperatureSensor>
```

Part of a sample service definition with EasyLife Ontology is as follows:

```
<TemperatureSensor>
  <hasGetMethod name="GetCurrentValue">
    <hasOutput>
      <parameterType>integer</parameterType>
    </hasOutput>
  </hasGetMethod>
</TemperatureSensor>
```

Let's assume that first definition is used in composition. If simple changes like method name, parameter type are to be applied, composition infrastructure and existing compositions using the service should be updated. Composition framework is unable to use new definition and needs to be updated accordingly. If composition infrastructure is provided by a company and users create compositions using it, users will be dependent on coming updates by the company. Besides, users only will be able to use services compatible with the ontologies that company supports.

If we assume that the flexible definition is used in composition, no update is required in composition framework. Because framework does not depend on the specific name "GetCurrentValue". It only knows that service has a method named "GetCurrentValue" from "hasGetMethod" tag. So the name of method does not matter for composition framework. Users can define and use their services easily.

One of the most important challenges in composition frameworks is supporting new generation devices. With domain specific ontologies, newly produced devices can adapt to composition ecosystem just by conforming to that ones. If it does not have method named "GetCurrentValue", composition infrastructure will be unable to use it in compositions. Lightweight ontology enables new devices to be embedded to ecosystem with a generic definition.

It is not possible to add new services to existing devices with a restrictive ontology. However, flexible ontology supports interface independence, so "GetMeanValueForDay"

method can be easily added to an existing “TemperatureSensor” device. This does not require any change in composition framework.

Advantages of a lightweight ontology can be understood with a scenario:

- A patient uses Device A to measure blood pressure.
- Measured value is queried with “GetMeasuredValue” method by the composition framework and sent to patient’s doctor.
- Device A crashes.
- The patient cannot find another Device A and instead buys Device B.
- Since the producer of Device B is different from that of Device A, interface of measured value service and return type is also different.
- In this case, while restrictive ontology causes Device B to be useless, generic ontology enables to define a new composition and use it.
- Doctor requests to measure blood sugar of the patient.
- Device B is capable of measuring blood sugar, there is no need to buy a new one.
- Since flexible ontology enables to add new services to existing devices, a new service “GetCurrentDiabeticValue” can be defined and used in a composition.

Considering the resource constrained environments in WoT, it is a requirement for an ontology to be simple and lightweight. Because of this argument, EasyLife ontology is not complex. Existing ontologies (such as OWL-S and SSN) are mostly complex for a non-expert user to use them. This complexity hinders the widespread adoption of comprehensive semantic models in IoT.

EasyLife Ontology does not prevent user from using other ontologies in service definitions. For instance, W3C’s Geolocation Ontology [34] can be used to describe the location of a sensor.

Following XML part shows an instance of embedding other ontologies to service definition:

```
<geo:Point>
  <geo:lat>55.701</geo:lat>
  <geo:long>12.552</geo:long>
</geo:Point>
```

It can also be useful to link a property of the service to an existing linked data on the cloud. To state that a sensor is inside METU campus, definition may be as follows:

```
<Location> http://dbpedia.org/page/Middle_East_Technical_University </Location>
```

This information boosts semantic service search. Querying services with a location based query, a composer can find services in METU campus easily.

In WoT, services communicate over web standards. Resources described in service definitions are accessible over internet. So, we adopted RESTful method declaration in our framework. This method has advantages over adhoc protocols since HTTP is an accepted protocol all over the world and no security issues appear such as reserving a specific port and adjusting firewalls for access.

```

<hasMethod>
  <Method rdf:about="http://192.168.1.18/opticalProximity">
    <parameterType rdf:datatype="&xsd:string">type_integer</parameterType>
    <rdfs:comment>Gets proximity sensor value</rdfs:comment>
    <hasInput>
      <Input rdf:about="#VOID">
        <rdfs:comment>void</rdfs:comment>
        <parameterType rdf:datatype="&xsd:string">type_void</parameterType>
        <parameterOrder>1</parameterOrder>
      </Input>
    </hasInput>
  </Method>
</hasMethod>

```

Figure 9: RESTful method declaration

The figure above shows a sample method declaration. When user opens resource link with a web browser, result will appear on the screen.

Event support is the most important property of a composition framework. Situation awareness is only possible with processing notifications and inferring high-level, useful knowledge. EasyLife enables user with describing events and consuming in composition framework.

```

<hasEvent>
  <Event rdf:about="tcp://192.168.1.3:1883/sensors/opticalProximity/EventCurrentValueChanged">
    <rdfs:comment>Fired when optical proximity sensor current value changed</rdfs:comment>
    <parameterType rdf:datatype="&xsd:string">type_integer</parameterType>
  </Event>
</hasEvent>

```

Figure 10: Event declaration

The figure above shows a sample event declaration using EasyLife ontology.

Since services in WoT should be accessible with web protocols, either device should have web support or a smart gateway should present services of devices. In case of smart gateways, communication between gateway and other devices can be over specific protocols. Existing WSDL-based services can also be included into EasyLife platform with smart gateways. To achieve this integration, inputs and outputs of existing services should be converted to provide RDF-based semantic data. Conversion from semantically annotated data to raw data and from data to RDF data is defined as “lifting and lowering” in [35]. “Lifting and lowering” processes are done in smart gateways.

3.2. Endpoint Repository

In our framework, “endpoint” is a service provider that provides one or more functionalities to consumers. An endpoint declares its services in a file created using EasyLife ontology. Service definition contains both semantic information about services and technical details about how to use them. In this way, composition framework enables users to search services

and use them in compositions. A sample service definition file is given in APPENDIX B. It contains multiple services, event and method declarations. Endpoints should have a “service definition URL” serving service definition file to clients. In EasyLife system, composition framework is the main consumer of services. Thus, it should be aware of existing endpoints to search and use services in compositions. Our solution for this problem is “Endpoint Repositories”. An Endpoint Repository holds “service definition URLs” of endpoints. It enables endpoint owners to register “service definition URLs” to repository via a web site. Endpoint Repository is a SPARQL server and supports SPARQL queries. When a composition framework user searches services supplying a keyword, framework constructs a SPARQL query and sends to Endpoint Repository. Endpoint Repository runs query over available service definitions and merges results into a single service definition file. Query result is returned to composition framework and listed to user.

In WoT, there may be billions of devices and services available on the network. Hence, scalability is a major problem that should be overcome. If we send search queries to each endpoint instead of Endpoint Repository, resource-constrained devices can be out of power or bandwidth. In fact, every endpoint may not be able to support SPARQL queries. To deal with this problem, we designed Endpoint Repository such that it retrieves service definition files from endpoints and caches them until expiration. Service definition files also contain a field that indicates expire date. When a file is expired, Endpoint Repository reloads it from its endpoint to local cache. If an endpoint’s service definition is subject to change frequently, expire date should be close to publish date to reflect changes to repository in a short time.

An endpoint repository is similar to a UDDI registry. In UDDI, providers publish service definitions, quality assurances and technical details of services to registry. But in our solution, endpoint repository only requires “service definition URLs”. When needed, definitions are retrieved from endpoints. By this way, service definitions are distributed and management is done locally by service providers. Change in service definitions does not need to be reflected to repository. This is important for non-expert service providers. UDDI forces a centralized structure because it holds all service definitions. Administration and maintenance of this centric registry is highly difficult and costly. As in the case of IBM, Microsoft and SAP, most of the public UDDI registries were closed. Considering a huge number of services in WoT, a centralized structure similar to UDDI is not realistic and practical. Our Endpoint Repository has also semantic search capability when compared to UDDI. This is a powerful feature to find desired services.

It is worth mentioning that EasyLife system allows more than one Endpoint Repository. This is reasonable when we want to load-balance repositories over search queries. Composition framework can also be configured to work with specific repositories. For example, a university may have its own university repository containing endpoints that reside in campus. A government can also have a repository that is only available to governmental composition frameworks. In this manner, we distribute repositories and handle scalability issues.

3.3. Composition Framework

EasyLife contains a semi-automatic composition framework that has the following features:

- Contains web based Graphical User Interface (GUI).
- Semantically search services from endpoints.
- Supports history.
- Supports Linked-Data.
- Aware of the context of user.
- Supports method composition.
- Supports event composition.
- Process over input and output parameters.
- Supports creating rules.
- Supports template definition and dynamic composition.
- Supports goal-based event composition. User can search through predefined goals and compose services according to this schema.
- Contains type matcher providing basic and advanced type matching.
- Supports many more instances.

To address many users including non-experts in WoT, usability of the system is important. System should have a simple GUI and if required should be installed easily. EasyLife composition framework has a web-based GUI. Users have membership to create and manage their compositions. GUI contains everything to create a composition. As depicted in Figure 11, in the simplest case, composer firstly searches services. Framework converts search keys to SPARQL queries and sends semantic queries to Endpoint Repository. Endpoint Repository searches from registered service definitions and return matching services as RDF data. Composition framework lists services to the composer. Then, composer drags and drops methods or events to the composition area and binds to each other to achieve desired goal. After that, composer saves composition and runs whenever s/he wants.

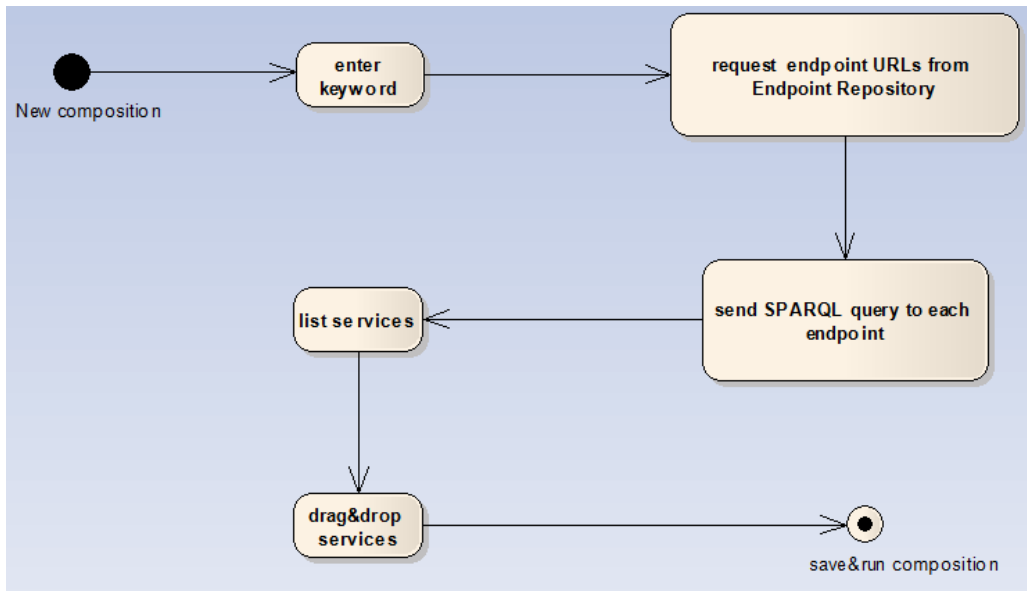


Figure 11: Basic composition scenario

Considering billions of devices and services, it is vital to search and find required services easily. Successful search can be boosted with semantics if processed carefully. EasyLife enables semantic search since services are defined with ontologies. Besides, we have mentioned that ontologies other than EasyLife ontology also can be embedded into service definitions. Advantages of semantic search can be explained with a simple scenario:

- Service type is defined using an ontology as following.

- `<type> http://linkedgedata.org/page/ontology/Temperature</type>`
- In the referenced ontology, the term “Temperature” has meanings in many cultures (i.e “Sıcaklık(tr)”, “Temperature(en)”).
- When a user searches as “Sıcaklık”, endpoints can traverse type field in service definition from referenced ontology and can search from keyword in different cultures.
- Eventually, searches in different cultures yield same service results.

Linked Data is one way to publish, share and connect data via URIs on the web. It focuses on interconnecting data and resources on the web by defining relations between ontologies, schemas or directly linking published data to other existing resources on the web [7]. Using publicly available Linked Data in service descriptions also enhances search process. Because it enables related data and relevant information to be discovered. Relating service attributes such as location, type to other resources on the web improves findability. We can also explain this case with a scenario too:

- Service type is defined using an ontology as following.

- `<location> http://dbpedia.org/page/Middle_East_Technical_University </location >`
- In the referenced dataset, “city” property of the resource is “Ankara”.
- When a user searches services in “Ankara”, composition framework queries over dataset. Machine can understand that “Ankara” contains “Midde East Technical University” and returns defined service to the user.

The power of semantics depends on how much ontology and Linked Data are used in defining services. If service definition quality is higher, search result quality will be better.

Search activity can be optimized with user context and keeping and interpreting search histories. With the usage of contextual information as inputs, client interaction will be reduced. User context may contain location, age, educational status, gender, etc. Dynamic context attributes like location can be obtained from internet service provider or from device (if supports). Static attributes are acquired from user’s membership information. Composition framework analyses search history and presents services used in previously created compositions. It can also reorder search results according to user’s location information. Services closer to user location can be introduced at first. User can also sort results according to some specific attributes such as provider, location, name, and quality.

Result of the search phase provides service descriptions that contain methods and events of each service. At the composition phase, user can chain methods and events according to the goal to be achieved. Input of a method can be another method or output of an event can be input of a method. The framework traverses composition from output and invokes required methods accordingly.

Processing over method's or event's input/output is also available in framework. For example, if a light sensor provides values in the range [1, 1023], and the method that will use this value requires range [1, 255], user should be able to modify the output by calculations. Otherwise, composition will produce wrong result.

In sensor domain, value of data depends on real time processing which may trigger some other actions. Mostly, a sensing device sends data to a system to create situation awareness. This depends on the use of event mechanism. A composition becomes much more powerful with an event since it runs silently and alerts when required. Without events, compositions would be lightweight. Either user would run them many times to check if they succeed or framework would run periodically. In the case of user, it would be time consuming since user should wait in front of screen and run composition many times. In the latter case, participants of the composition may have performance issues because of many unnecessary requests. Moreover, desired result may not be achieved. This situation is mostly possible since it is hard to send "get data request" to a device at correct time. For example, considering a motion detector having 20Hz frequency, it is almost impossible with a feasible request frequency to inform user if a motion is detected. Sensor's data frequency may also be too low (once in a day). In this case, resource-constrained devices would be overloaded with many requests to catch the correct time.

EasyLife framework enables events in sensor domain. They are the first class citizens of composition. Composer can use events, bind provided data to output or to another method. Methods in EasyLife can be internally provided (such as GeoLocation service) or can be part of external services. Considering internally provided services, if composition framework has required abilities, then user can directly achieve desired goal. Otherwise, s/he can search through services and use them. For instance, if user has a specific pattern recognition service, then s/he can bind sensor data to the service and act according to output of that external service.

Interpreting event data and producing high-level knowledge depends on combining with two or more events and defining rules. EasyLife composition framework supports creating rules over events. User can use Event Composer component to select events. Then, s/he can create rules as the following samples:

- If Event A is fired 10 seconds after Event B
- If Event A has value greater than 378
- If average of last 10 value of Event A is less than 55

Rule support is provided with Drools Rule Engine [11] which is an enterprise framework. It has many powerful features such as processing stream of events, temporal reasoning and reasoning over absence of events. So, variety of rules provided by EasyLife can be increased easily with the needs.

EasyLife framework enables saving composition definition as a template and run later. Templates can be useful when user creates a composition to achieve a goal and starts with changing services in different contexts. Composition is saved with semantic descriptions of services. When user fires composition, available services are analyzed. Dynamic service selection is carried out semi-automatically with user. For each required service in composition, framework presents available services to user and s/he selects one of them.

Power of template and dynamic service selection can be explained with a Smart Home scenario:

- User creates a composition to manage Smart Home light and temperature devices so that temperature should be 26 °C at winter and 22 °C at summer. Brightness of lights should be 75%.
- When user goes home and fires composition, framework finds light handler and air conditioner devices.
- User approves the usage of devices in composition and ambience adapts to user needs.
- User goes to a hotel and wants to apply ambience settings to room too.
- User selects template composition and runs.
- Framework finds appropriate devices and presents, user selects and approves.
- User's goal achieved in hotel room and home successfully with single composition.

Template composition is not a new idea. In [37], ontology (OWL-S) based template workflow definition is discussed in detail. A matching and ranking algorithm is provided for selecting services to fulfill template workflow. Author in [38] proposes a solution for template business process modeling so that users can search and match a template according to requirements. Concrete services will be given with a configuration file at design time by user. Template construction and matching process details are given in article. In [39], task based dynamic service composition is discussed. Main motivation for article is reducing complexity of service composition by dynamically selecting suitable services according to user context. A contribution to these works in our study is the history support for optimizing service selection. Since our framework is semi-automatic, it interacts with user when one or more matching services are available. User can select "remember" option and framework does not ask later for this type of services if the selected service is available in the context. By this way, after a period of time, user interaction will be minimized to run a template composition.

Another feature of the EasyLife framework is goal-based service composition. This feature requires that the goals are defined with ontologies. These goal ontologies should be also publicly available. A goal can be defined as follows in the ontology:

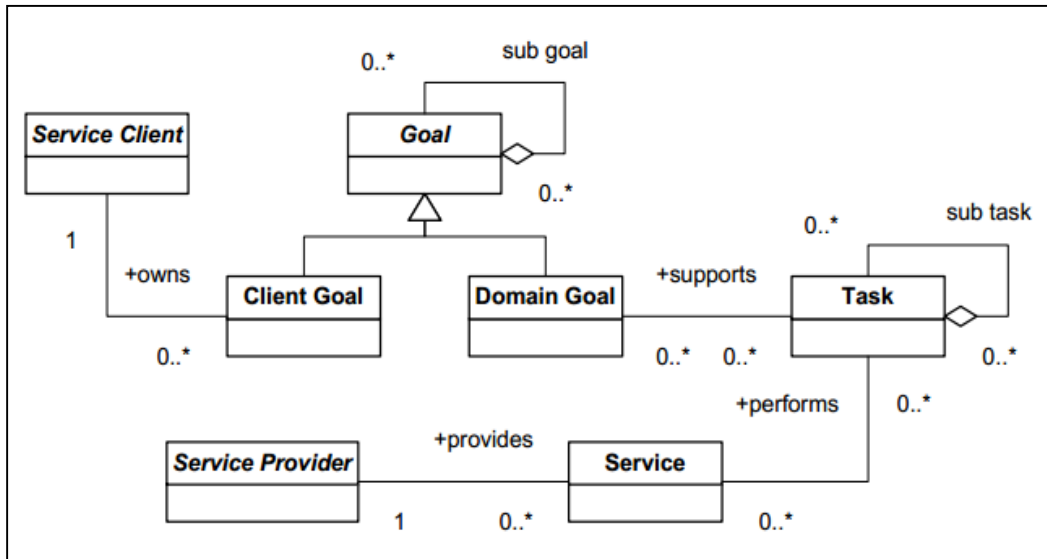


Figure 12: Definition of goal, task and service [3]

EasyLife adopts goal and task definition in [3]. A goal is the high-level concept that user wants to achieve. A task is defined as the means to fulfill a goal. Goals and related tasks are presented in goal ontologies. A service can perform one or more tasks.

When user informs EasyLife platform about the goal to be fulfilled, platform’s matching algorithm searches in goal ontologies that conforms to user’s goal. If a matching goal is found, related tasks are retrieved. Tasks contain semantic tags that will be used in search queries. At the next step, EasyLife constructs and sends query to Endpoint Repositories to search services supporting tasks and service definitions. For each task in goal, user selects a service. From this point, flow is same as Template Compositions. Whenever user wants to run the composition, suitable services are presented to user and s/he selects and approves.

An example goal can be as follows:

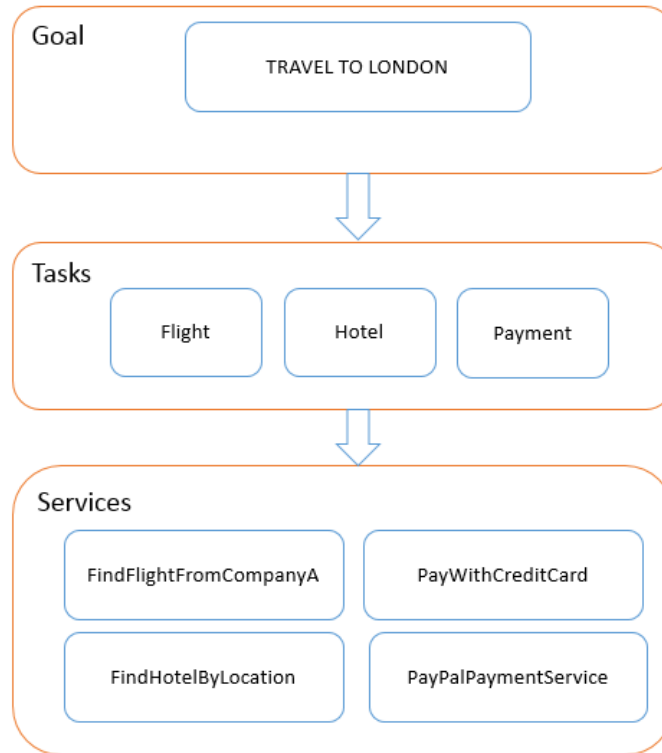


Figure 13: Sample goal, task and service definition

Goal ontologies help user to create a composition with high-level terms. Non-expert users can take advantage of this by just typing goal description and running the result. If there exists a large number of goal ontologies defined, composition will be easier for the user.

Type Matcher is a component of the framework that decides whether a type is convertible to another type. If composer tries to bind methods or events, it analyses inputs and outputs and allows chaining if possible. Decision for simple types is straightforward but not for complex types. Semantic inference should be done over types to check inheritance and compatibility.

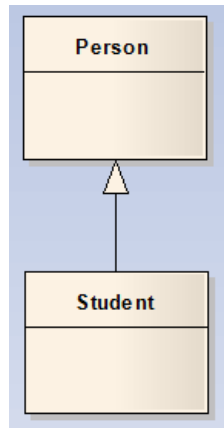


Figure 14: Sample class inheritance

Let's assume that Student class inherits from Person class as shown in Figure 14. Method A requires parameter of type Person and Method B produces Student. If user wants to bind output of Method B to Method A, Type Matcher checks suitability and infers that Student is a Person. Then, it allows binding process.

CHAPTER 4

PROTOTYPE IMPLEMENTATION

One problem with proposed solutions for service definition, discovery and composition framework is that they are usually untested. To support the solution proposed in this thesis, a prototype is developed. Details of the implementation are discussed in detail.

4.1. Scope of Prototype

We focused on the fundamental features of EasyLife platform in our prototype and implemented the following parts:

- Create EasyLife ontology
- Define service definitions with this ontology
- Support methods and events in definitions
- A SPARQL-enabled endpoint repository
- Service composition framework

Template and goal based composition, linked data, service search, history and context management are not included in this prototype. Discussion about the way of implementing these features based on Related Work (Chapter 2) is given below.

An ontology based template definition and service selection algorithm using a matching and ranking algorithm is provided in [37]. Author in [38] proposes a solution for template business process modeling so that users can search and match a template according to requirements. Dynamic service composition according to user context is discussed in [39]. In [31], executing task according to dynamically changing context is discussed. In EasyLife, when user creates a composition and wants to save it as template and run later, Composition Framework can save semantic definition of participant services. Then, at runtime, matching services can be found with an algorithm as defined in [37] and user context can be used to filter services as discussed in [39]. User can be prompted to select from alternatives and composition will be constructed accordingly.

Goal or task based compositions are discussed in [3] and [4]. In [3], ontology based task definition, dynamic service discovery and composition issues are stated. In [4], goal is analyzed, decomposed in sub-goals, relevant services are searched and a template source code is generated for developers. Assuming that goal and task ontologies are available, goal based composition feature can be implemented with the same way stated in these articles.

In EasyLife, linked data integration is used to optimize search operations and give more information about services by using known linked datasets. When linked data is used in service definitions, composition framework and endpoint repository should be able to process them. Apache JENA [13] can be used to query and infer knowledge from linked data.

Searching services can be done with service definition, keywords or linked data. Searching through service name, comment and URL fields is straightforward. But when it comes to information provided with linked data tags, Apache JENA can be used.

Implementing history feature is not a challenging issue. When user selects a service, system can keep keyword, selected service and user context. Then, when user searches same or similar keywords, system searches services and provides previously selected services to user at top order. Different selections based on same keywords can be optimized with user context such as location.

Components of prototype implementation are shown in Figure 15.

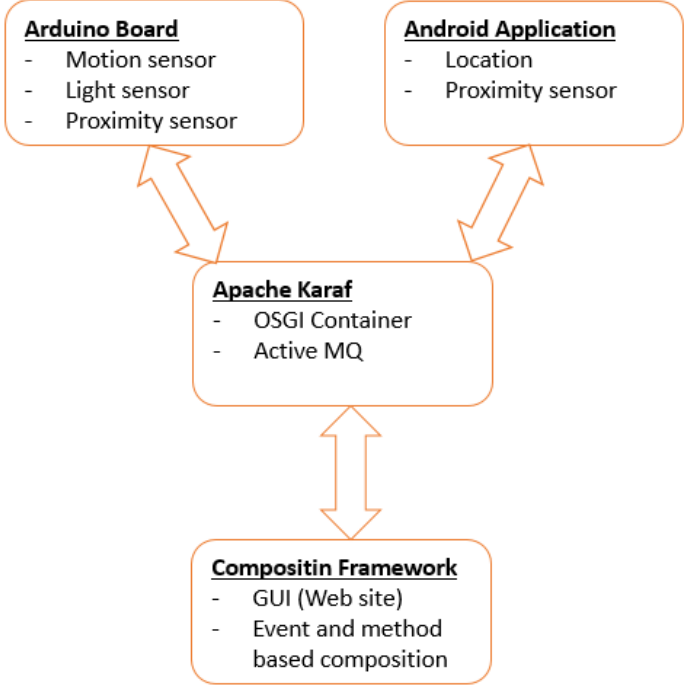


Figure 15: Components of prototype implementation

Apache Karaf is an OSGI container and contains an MQTT server named Apache Active MQ. This server enables event messaging. We also deployed a web project to Karaf to act as endpoint repository. It returns URLs in the repository when requested. Composition framework enables user to compose services based on a goal to be fulfilled. Framework runs as a web site which is implemented with Java and JSP. Arduino is a simple board which has a microcontroller. It supports plugging many sensors and Ethernet adapter. We integrated motion, light and proximity sensors on the board and connected it to Apache

Karaf with an Ethernet adapter. It provides sensor values with RESTful methods and event mechanisms. Android application runs on an Android smart phone. It provides location and proximity sensor values of device. Apache Karaf and composition framework run on a computer. All of the components are connected to a modem to communicate with each other.

4.2. Composition Framework

EasyLife composition framework is a web application implemented with Java in Eclipse development environment. Figure 16 depicts the architecture of framework.

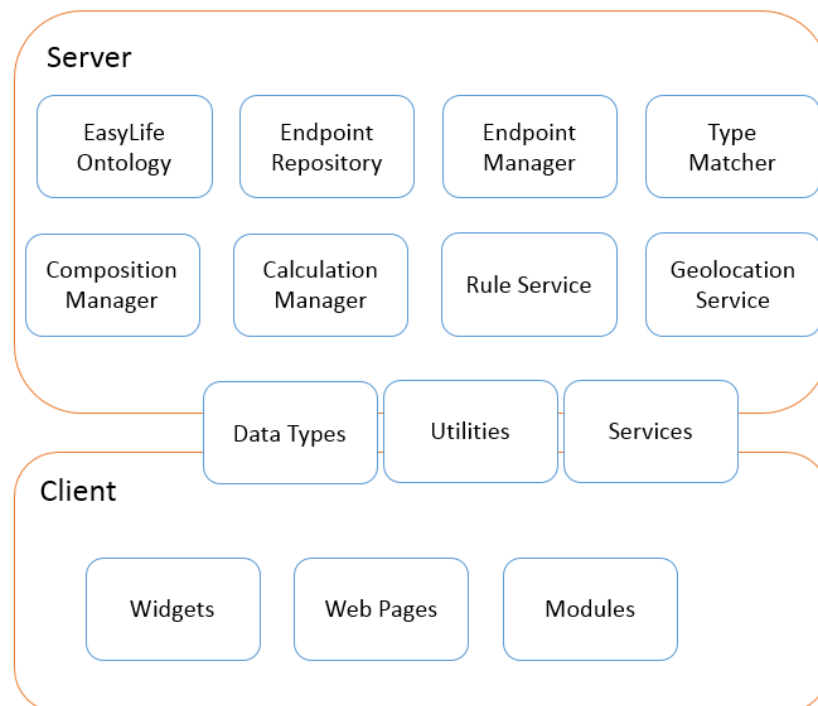


Figure 16: Server and client architecture of EasyLife

Framework project contains server and client sides. Server part runs the business logic and client part manages the interaction with user. There exists also a shared part that contains common code modules to be used in both server and client side.

4.2.1. Implementation of Server Side

Server side contains ontology definition, interacts with endpoint repository and endpoints, saves and runs compositions. Main modules of server are as follows:

- **EasyLife Ontology:** Ontology class contains the generic ontology definition as classes. Definition of ontology is presented in APPENDIX A. Apache JENA [13] is a free and open source Java framework for building Semantic Web and Linked

Data applications. It contains many components and tools. In prototype implementation, we used JENA library to convert ontology definition to Java source code. It yields java classes and attributes according to EasyLife Ontology. Since details of ontology are mentioned in Proposed Framework, we skip here.

- Endpoint Repository: This module is a web project which holds the URLs of service providers. When composition framework searches services, it finds and returns matching service definitions. In prototype, it only contains one endpoint URL.
- Endpoint Manager: This module provides two main functionalities. First one is it fetches service definitions from endpoints and caches in local storage. Multiple service definitions are parsed from RDF data and converted to internal data types. If service definition does not conform to EasyLife ontology, it is ignored. Second one is it constructs SPARQL queries to search services and sends request to endpoint managers.
- Type Matcher: This part contains the logic to decide whether one data type is convertible to another data type and does the conversion. It is used by Composition Manager to handle output-input relation in composition chain.
- Composition Manager: It handles business logic to create, run and delete compositions. Method based and event based compositions have different complexities. If a composition does not contain any event, it runs only once at a time. But, in the case of an event, composition is triggered every time when participated event produces data. Output may or may not be generated since it depends on the rules over composition. If rules succeed, then next item in the composition chain is executed. Otherwise, composition waits for next data from event source.
- A composition contains exactly one output node in the chain. This is required to find entry point of the composition. Output node is automatically put on the composition area when a new composition is created with GUI.

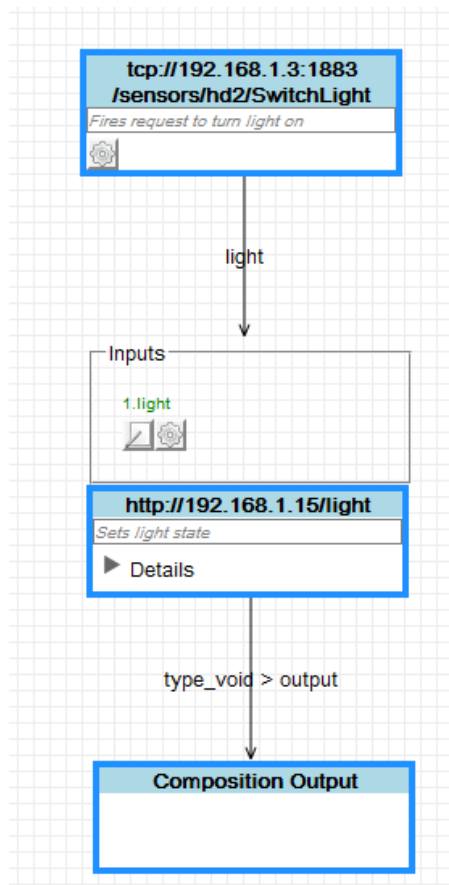


Figure 17: Method based composition

- Figure 17 shows a simple method based composition. Composition Manager finds output node and gets method “http://192.168.1.15/light” which is bound to the output. Then it continues traversing and gets the first method. Since no other method exists, it starts invoking and passes output of method to next one in the chain.

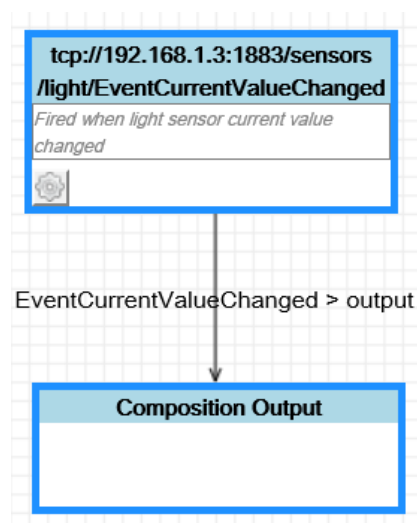


Figure 18: Event based composition

- An event based composition is shown in Figure 18. To run this composition, Composition Framework subscribes to event and waits for notification. When an indication is received from event source, event data is processed and sent to the output.
- Calculation Manager: It processes user defined calculations over input and output parameters of methods and events. For example, user defines a calculation that multiplies output of a method by 255 and divides by 1024. Calculation manager carries out these calculations while composition is running.
- Rule Service: While creating a composition with events, user can define rules over events. This component sets up rules of composition when it runs. Then it waits for event notifications to check rules over event data. If rule succeeds, Rule Service notifies Composition Manager to continue processing the composition. Otherwise, it discards notification and waits for next event indications. Drools Rule Engine [11] is used in this part which is a professional enterprise framework for the construction, maintenance, and enforcement of business policies in an organization, application, or service.
- Geolocation Service: It is responsible to provide a service that enables user to process coordinate and location operations such as calculating distance between two coordinates.

4.2.2. Implementation of Client Side

Client side is responsible for enabling user to manage compositions. It is a web project implemented with Java, JSP, HTML and Google Web Toolkit (GWT) [10] library. GWT is a development toolkit for building and optimizing complex browser-based applications. It runs on Eclipse environment with a plugin. Client side of the prototype implementation contains the following parts:

- Widgets: Contains all web parts to be used in composition GUI.

- Web Pages: There are some web pages that user interacts with the composition framework. These pages are mostly composed of widgets and managed by client modules.
- Modules: Handles client side logical operations.

4.2.3. Implementation of Shared Part

Shared part is used by both server and client modules. Parts of this module are listed below.

- Data Types: Contains all required data types to be used in EasyLife Composition Framework.
- Utilities: Contains useful utility classes.
- Services: Contains GWT classes to manage compositions, search, and services.

4.2.4. Graphical User Interface (GUI) of Composition Framework

In this part of the chapter, GUI of the composition framework is explained in detail. Screenshots with sample data are provided in explanations for better understanding.

When user enters the system, following composition page is displayed.

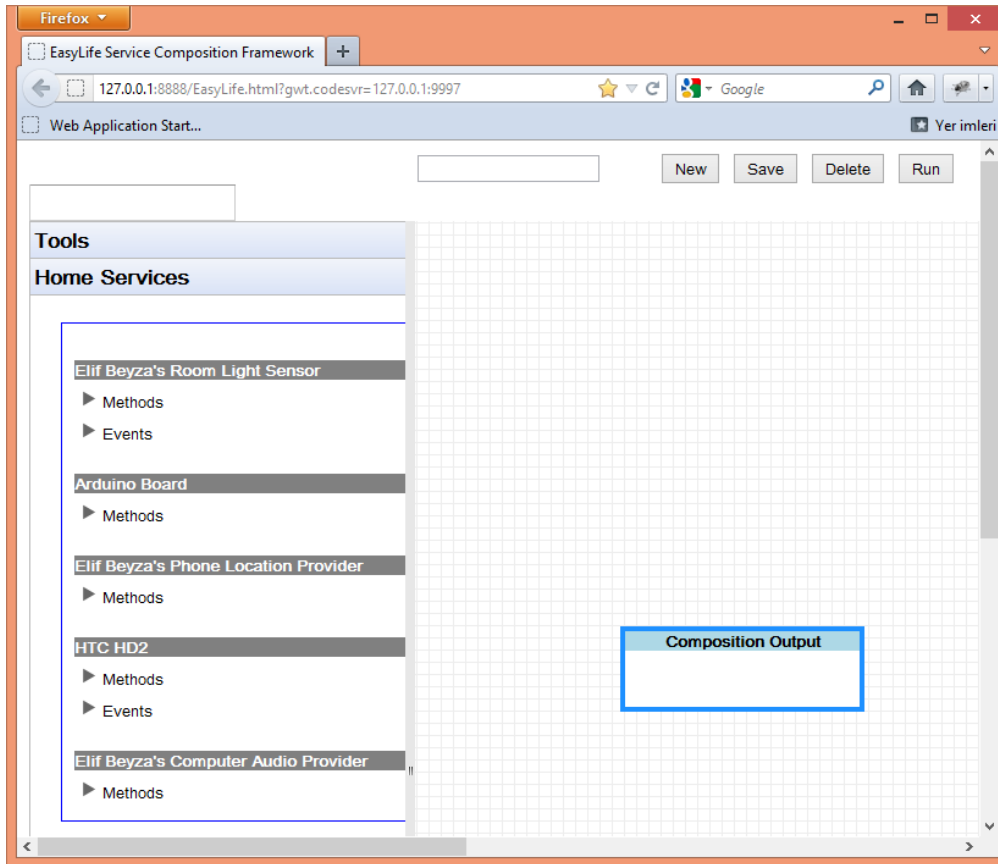


Figure 19: EasyLife composition framework GUI

If we look at the layout of the page, tools and services from endpoints are provided on the left. Services are searched from endpoints and according to the responses; service tree is filled with methods and events. Action buttons about current composition are on the top. Composition area fills the rest of the page. User drags and drops methods and events from services to this area and binds to each other according to the goal that will be achieved. Composition Output is required in every composition to know the start point.



Figure 20: Tools pane in composition framework

Tools pane contains helper functionalities. At the moment, only Event Composer tool exists. This tool is used to combine two events and force rules over them.

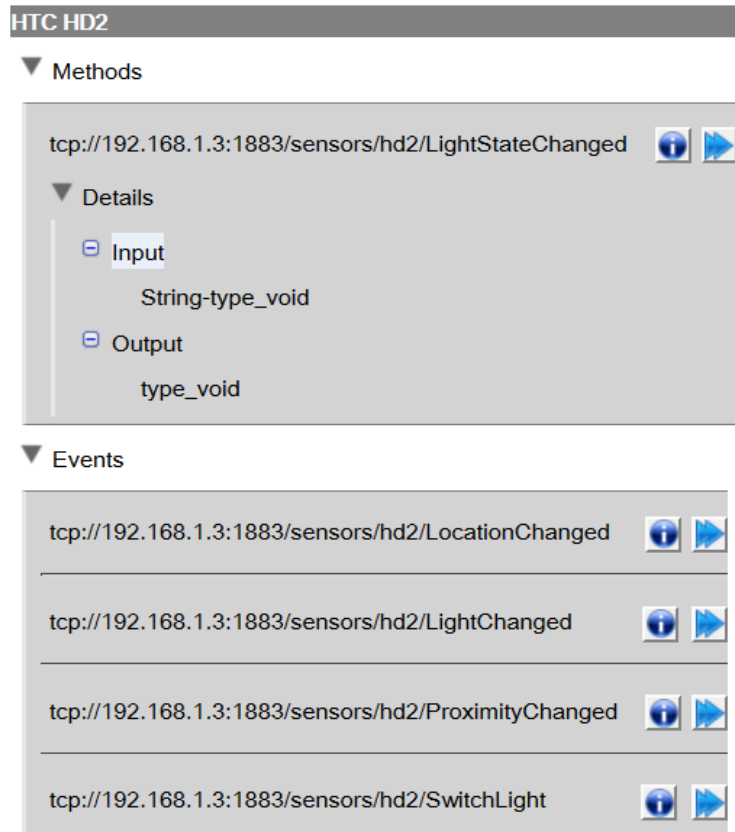




Figure 21: Endpoint services in composition framework

Endpoints may provide many devices and these devices may include many methods and events. For example, “HTC HD2” is a device provided by “HOME SERVICES” endpoint (Figure 21). This device has one method and three events. Button with picture  provides information about methods and events. Button having image  adds an instance of method/event to the composition pane on the right. Details of a method are also available to user as input and output.

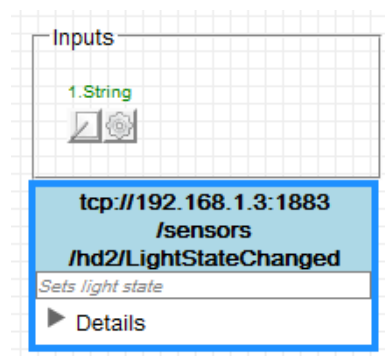



Figure 22: Event in composition

Figure above shows an event that is added to the composition pane. Input parameters and available actions are provided on the top. By clicking  button, user can enter a manual value from screen below:

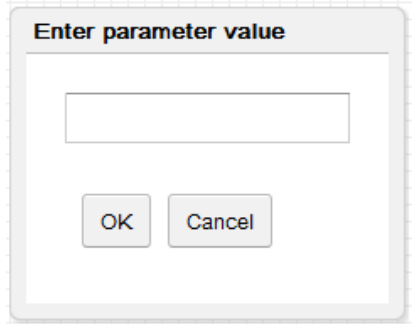



Figure 23: Defining parameter

By clicking  button, user can process an input parameter from below screen:

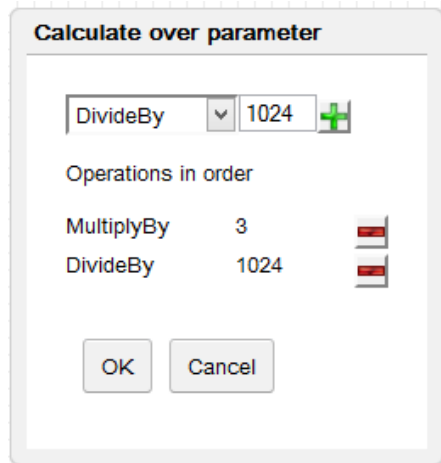


Figure 24: Processing over parameter

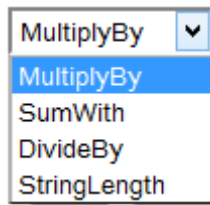


Figure 25: Process types over parameters

These parameter actions can be extended if needed. Operations are applied in the order defined by the user.

A basic composition is illustrated below. When composition is started, current light value is fetched from light sensor and printed to the output.

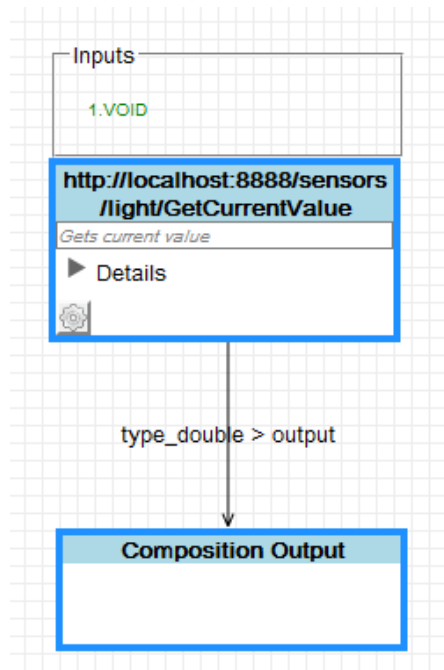


Figure 26: Sample composition

If there exists at least one event, it should be on the “top” which means the trigger component of the composition. When event is fired, execution stack starts and next action in the chain is executed.

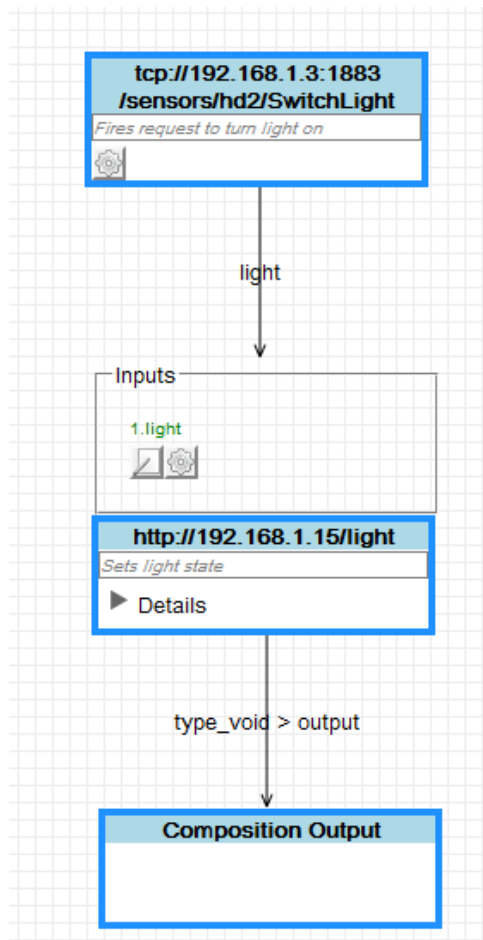


Figure 27: Sample composition

In the composition shown above, user tries to control the light of Arduino device from mobile device. When a “switch” event is fired from mobile device, composition framework invokes Arduino device’s light method with the parameter taken from event.

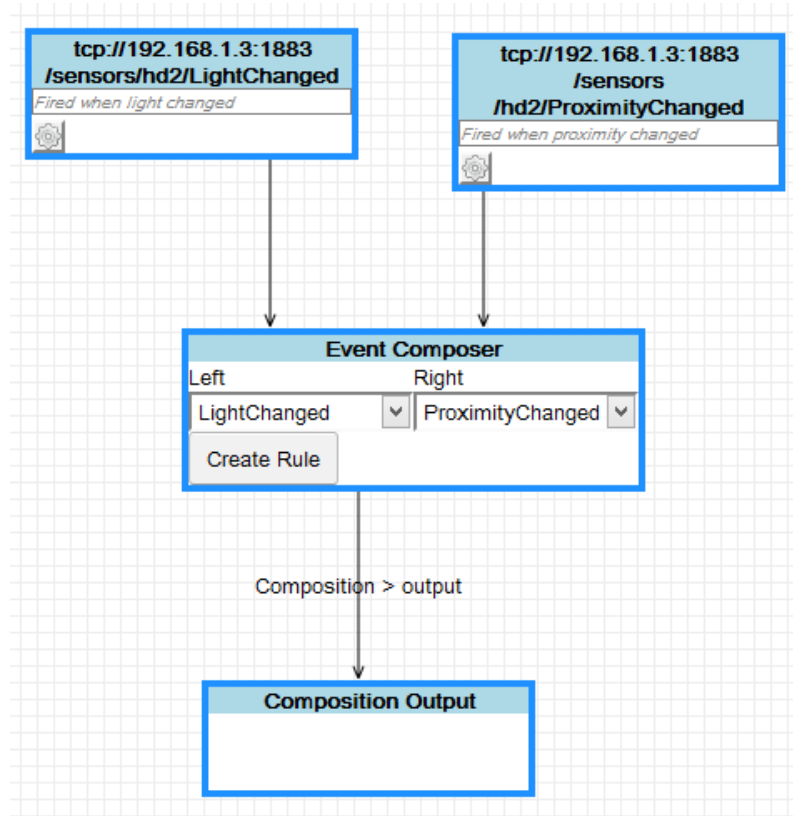


Figure 28: Sample composition

Event composer tool is used to combine two events, mostly with rules. When user binds events to tool, lists on the left and on the right are filled with event names. Then, user chooses the left and the right events. Afterwards, user can create rules from the screen below:

The 'Create Rule' dialog box contains the following options under the 'Fire event' section:

- If the **Average** of last data is **EqualTo**
- If the **Average** of data in last **Millisecond** data is **EqualTo**
- If the Right event data received after **Millisecond** than Left event data received
- If the Right event data received between - **Millisecond** after Left event data received

At the bottom of the dialog are 'OK' and 'Cancel' buttons.

Figure 29: Rule creation

These rules are implemented just as prototype and can be extended if necessary. Defined rules are converted to Drools rule definitions in server side before composition runs.

4.3. Android Application

In prototype implementation, we developed an Android Application using Eclipse and Android library. This application acts as a sensor device and provides events and methods that can be used in composition.

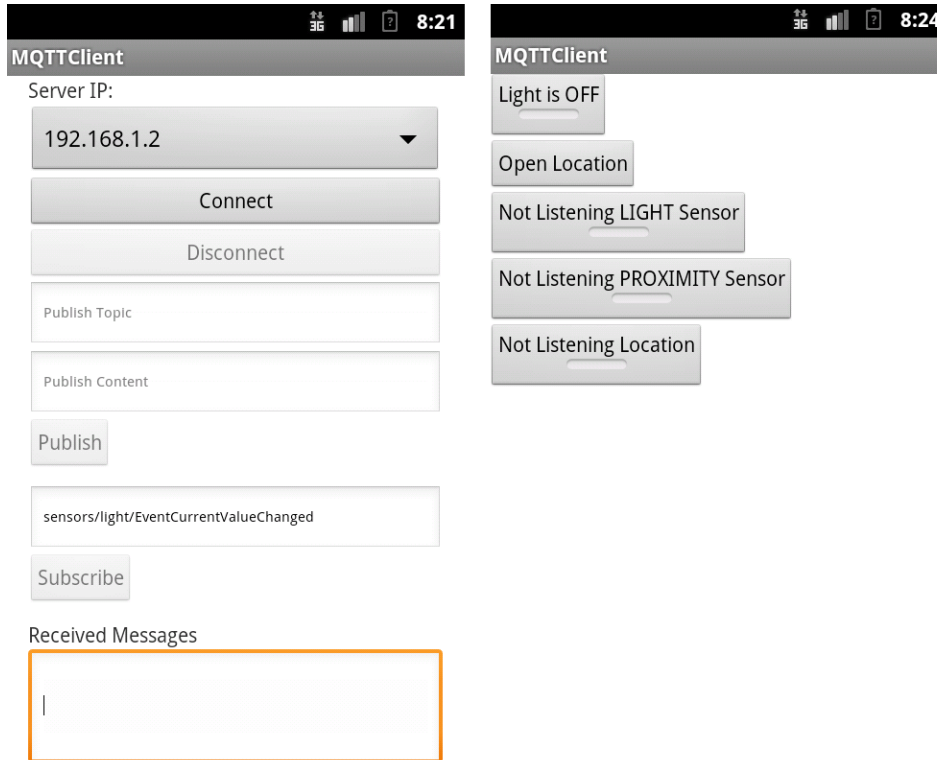


Figure 30: Android application

At first page, user selects MQTT server IP and connects to it. Second page contains buttons to send events such as light sensor value, proximity sensor value, location and a method to on/off the light.

Service definition of this application is provided by an endpoint (in our case it is “HOME SERVICES”). Then, from composition GUI, user can use these services in compositions.

4.4. Arduino Application

We also developed an application for Arduino device. As seen in Figure 31, Arduino setup includes the following sensors on the board:

- Light sensor: Provides light value of the environment.
- Motion sensor: Fires event when a motion is detected.
- Proximity sensor: Fires event when an object is close to it. It can be used to detect whether doors/windows are open or close.

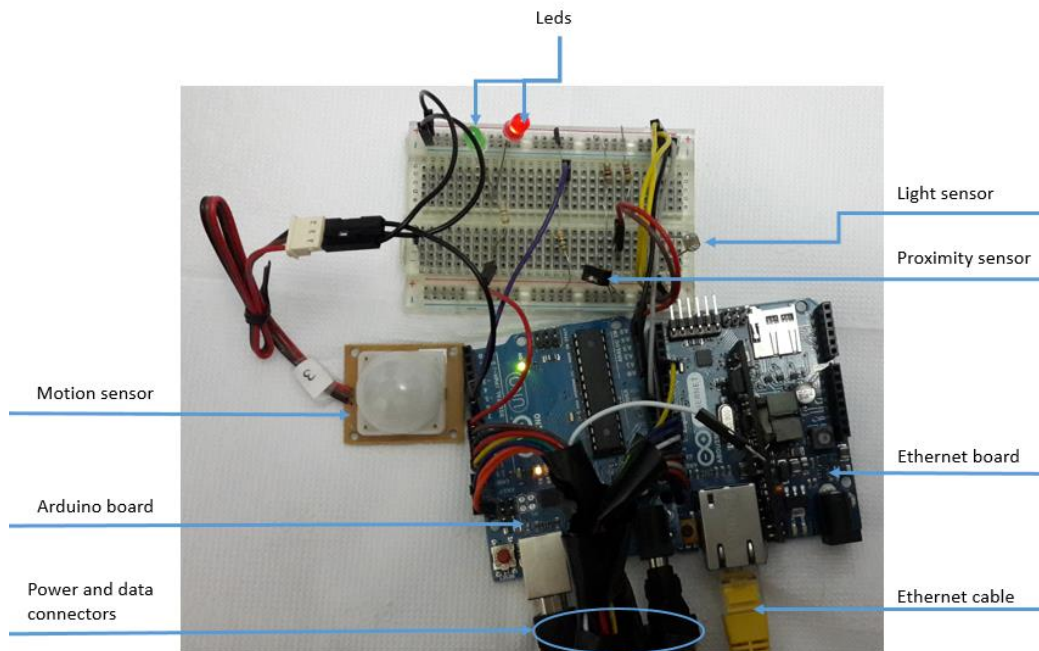


Figure 31: Arduino board setup

Our setup also contains an Arduino board and an attached Ethernet board which enables device to communicate over network. Simple leds on breadboard are used to run simple scenarios such as “alert if a motion is detected”. Some resistors exist for sensor integration.

As in Android application, service definitions of this application are also provided by “HOME SERVICES” endpoint. Application supports MQTT, thus can publish events to an MQTT server. Arduino application also has the ability of processing RESTful web requests. For example, from browser value of light sensor can be retrieved.

CHAPTER 5

EVALUATION AND RESULTS

In this section, we compare EasyLife with existing works. Some of these works are active, but some are discontinued. Comparison will be based on framework features and scenario-based capabilities.

5.1. Feature-Based Comparison

<i>Features / Frameworks</i>	<i>Implementation</i>	<i>Composition Framework</i>	<i>Semantic Service Definition</i>	<i>Semantic Service Search</i>	<i>External Service Definition</i>	<i>Event Support</i>	<i>Event Rule Definition</i>	<i>Context aware</i>	<i>History support</i>	<i>Linked data support</i>	<i>Template definition/ Dynamic composition</i>	<i>Goal search / Goal-based composition</i>
iServe	Yes	No	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Yahoo Pipes	Yes	Yes	No	No	No	No	No	No	No	No	No	No
Microsoft Popfly	Yes	Yes	No	No	No	No	No	No	No	No	No	No
ClickScript	Yes	Yes	No	No	No	No	No	No	No	No	No	No
SOA4All	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No
Amigo	Yes	No	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
EasyApp	Yes	Yes	Yes	Yes	Yes	No	No	No	No	No	No	Yes
SensorMasher	Yes	No	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
EasyLife	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes*	Yes*	Yes*	Yes*	Yes*

NA: Not Applicable

*: Not implemented in prototype

Figure 32: Framework comparison based on features

In the table above, features that are common to semantic service definition and composition are listed as columns and frameworks are shown in rows. Implementation column is about if any coding process is carried out for proposed solutions. All features after second column is related with composition framework. Thus, if no composition framework exists, these features are not applicable.

Traditional web services without semantic definition are out of the scope of this thesis. So they are not listed in this comparison table.

iServe does not provide any service composition infrastructure. It is about semantic description of services with annotations. It enables wrapping traditional WS-* services with semantic definitions. Yahoo Pipes and Microsoft Popfly provide service mashup platforms that users can aggregate, manipulate and mashup content around the web. However, they do not enable users to describe their own services and use them in compositions. Users can only use predefined tools and services in mashups. Semantics is not in the scope of these frameworks. ClickScript can just access RESTful services and enables users to create mashups with these services. It provides conditional operators. But it cannot process WS-* style web services. Event handling, rule definition and other features are missing in this composition tool. SensorMasher project aims to publish sensor data with semantic annotations. WoT is not the main concern of this project since all sensor devices are directly connected to SensorMasher. It does not provide a distributed platform that gets data over the internet. Sensor data is annotated by SensorMasher middleware. No service definition exists in this platform. Raw data is streamed and processed directly. SensorMasher also does not present any service composition infrastructure. Amigo uses OWL-S extended ontologies. EasyLife ontology is also a subset of OWL-S with MQTT support. Amigo is specific to smart home domain, but EasyLife's vision is WoT. Another disadvantage of Amigo is the usage of strict ontologies which prevents different abilities and new devices to be integrated. EasyLife's semantic search capability does not exist in Amigo. Amigo considers zero configuration for devices and does not interact with user. Existing devices communicate with each other based on a predefined workflow. These workflows can be developed as plugins and deployed to Amigo platform. EasyLife is based on service composition and requires user interaction. EasyApp provides a service flow generator infrastructure for developers. When it gets a goal from developer, it analyses and decomposes the goal into sub-goals. Then, it generates template source code to achieve the goal. Composition framework in EasyApp does not support running compositions as EasyLife does. SOA4All has a process modeling tool that user can make compositions. Services can be annotated and searched. It has event and rule support but no ability of goal-based service composition and linked data support. Template definition and dynamic service composition is not in the scope of this project. Composition framework is highly complex and is difficult for the use of non-expert users.

5.2. Scenario-Based Comparison

Scenario 1: Alert my mobile phone, if door is opened when I am away from home.

To implement this scenario following requirements should be satisfied:

- *Mobile Phone Alert Service* should be defined to alert user based on composition result.
- *Mobile Phone Location Service* should be defined to find out user location from GPS. It should respond to composition framework's "get location request" and also publish change in location to composition framework.
- *Door Status Service* should be defined to notify door status. It should respond to composition framework's "get status request" and also publish change in door status to composition framework. This service may contain a proximity sensor to detect status.

- *Composition framework* is required to combine services to achieve this goal. It should manage events and create rules to determine if alert is required.

The scenario can run as follows:

User creates and runs composition with semantically defined services. When user location changes significantly, *Mobile Phone Location Service* sends location data to *Composition Framework*. *Composition Framework* compares home and user’s location to find out if user is away from home. If comparison succeeds, it gets door status by sending “get status request” to *Door Status Service*. If door is open, then *Mobile Phone Alert Service* is invoked and user is alerted.

Same scenario can also be triggered by *Door Status Service*. When door status is changed, *Door Status Service* notifies *Composition Framework*. Then, *Composition Framework* gets user location and compares with home location. Remaining process is the same with the previous case.

This scenario requires external service description, event capability and service composition. Only SOA4All and EasyLife can handle this scenario. All other frameworks have a missing feature as seen in the following table:

Features / Frameworks	Composition Framework	External Service Definition	Event Support	Event Rule Definition
iServe	No	NA	NA	NA
Yahoo Pipes	Yes	No	No	No
Microsoft Popfly	Yes	No	No	No
ClickScript	Yes	No	No	No
SOA4All	Yes	Yes	Yes	Yes
Amigo	No	NA	NA	NA
EasyApp	Yes	Yes	No	No
SensorMasher	No	NA	NA	NA
EasyLife	Yes	Yes	Yes	Yes

Figure 33: Required features and comparisons for Scenario 1

Scenario 2: Search through smart home goals, define a template composition to adjust home light, temperature and music volume. Then run this template composition in different rooms.

To implement this scenario the following requirements should be satisfied:

- Light, temperature and music volume controller services should be defined to adjust device settings.

- Searching goals should be provided. When a goal is found, it should be processed by *Composition Framework* to create template compositions.
- *Composition Framework* is required to combine services to achieve this goal. It should enable user to create template composition and search services, and then dynamically use them in composition.

The scenario can run as follows:

User searches through different goal ontologies by providing tags such as “smart home” or “room customization”. User creates a template composition to achieve the goal. In template composition, *Composition Framework* only saves service requirements based on ontologies. User goes to a hotel and runs the created template composition. *Composition Framework* searches required services based on user location and service definitions in template composition. User is asked to select one of them if required. Then, if all participants are found, composition is performed to achieve the template goal.

This scenario needs the following features:

Features / Frameworks	<i>Composition Framework</i>	<i>Semantic Service Definition</i>	<i>Semantic Service Search</i>	<i>External Service Definition</i>	<i>Event Support</i>	<i>Template definition / Dynamic composition</i>	<i>Goal search / Goal-based composition</i>
iServe	No	NA	NA	NA	NA	NA	NA
Yahoo Pipes	Yes	No	No	No	No	No	No
Microsoft Popfly	Yes	No	No	No	No	No	No
ClickScript	Yes	No	No	No	No	No	No
SOA4All	Yes	Yes	Yes	Yes	Yes	No	No
Amigo	No	NA	NA	NA	NA	NA	NA
EasyApp	Yes	Yes	Yes	Yes	No	No	Yes
SensorMasher	No	NA	NA	NA	NA	NA	NA
EasyLife	Yes	Yes	Yes	Yes	Yes	Yes*	Yes*

NA: Not Applicable *: Not implemented in prototype

Figure 34: Required features and comparisons for Scenario 2

As seen in the table, all other frameworks except EasyLife do not support providing template definition, dynamic composition, goal search and goal-based composition features. EasyLife supports all of these smart features and enables user to run this scenario.

CHAPTER 6

CONCLUSION

In this thesis, a semantically enabled flexible service composition framework is proposed for the Web of Things domain. A prototype containing some features of the framework is implemented. Advantages and disadvantages are discussed by comparing with related works in this field. To do a composition, services should be defined in a structured way and they should be searched and accessed. Finally a user can compose them to achieve a goal. A simple and flexible ontology is created to describe services. This ontology does not pose any restriction over service definition but only forces high level abstractions such as parameter, input, output and event. Our ontology also does not inhibit service owners from using other ontologies in service definitions. For service registry, Endpoint Repository is proposed. It just holds links of endpoints and fetches service definitions from endpoints when required based on expire date. Endpoint Repository has semantic search support over service definitions. Composition framework sends SPARQL queries to it to search services. Composition framework is implemented as a web project so that it is accessible via a browser. User can use events, tools and other provided services to achieve the desired goal.

A prototype containing the following modules (or applications) is implemented:

- A lightweight ontology to semantically describe services
- An Endpoint Repository to store and access endpoints.
- A web-based composition framework
- An Android application (as endpoint) providing services such as location, proximity light status
- An Arduino application (as endpoint) providing motion, light and proximity sensor values and switch of/on on-board lights
- Configuration of an MQTT server (Apache ActiveMQ on Apache Karaf OSGI container) to handle event communication between clients
- Many simple and complex scenarios are succeeded on prototype implementation.

The prototype contains major features that prove the feasibility and abilities of proposed solution. Thus, following parts of proposed solution are not in the scope of prototype implementation:

- A powerful composition visual editor
- Semantic service search
- Template composition
- Goal-based composition
- Using user context and history
- Linked data integration
- Advanced type matcher

Prototype provides a visual editor implemented with GWT, but can be improved using the infrastructure of ClickScript project. Benefits of service search is discussed in detail but not

implemented. In future, semantic service search functionality can be integrated to EasyLife. Creating template composition and running it with dynamic service discovery is also studied in thesis without coding. This issue can lead to discovery of other interesting topics. Searching goals and goal-based composition is also another topic which can help user to create compositions easily. Considering user context and history can make framework smarter. Type matcher in composition module is simple and can be replaced with an advanced one.

Most important key point in this framework design is the hybrid structure of billions of devices. While some of them may be resource-constrained, some may be very powerful. Therefore, ontology definition is kept simple and a lightweight event protocol (MQTT) is chosen. However, not more than 3 clients are used in sample compositions. Thus, the main problem is testing framework against much more clients and complex compositions containing a lot of participants. Performance issues can emerge and should be solved with this future work. Variety of services and usage of framework to manage real life scenarios can give rise to undiscussed problems. With the increasing number of providers, services and compositions, scalability problem emerges. Search process will take more time, services will be unable to serve more requests, and even composition framework will have difficulty in creating and running more compositions. Thus, a thorough study should be carried out to find out and solve scalability problems. Security issue also remains a challenging problem. Service providers should be able to set security policies over services and only the authenticated users should be able to use those services in compositions.

REFERENCES

- [1] Evans, D. (2011). The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper*.
- [2] Barnaghi, P., Wang, W., Henson, C., & Taylor, K. (2012). Semantics for the Internet of Things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 8(1), 1-21.
- [3] da Silva Santos, L. O. B., da Silva, E. G., Pires, L. F., & van Sinderen, M. (2009, April). Towards a goal-based service framework for dynamic service discovery and composition. In *Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on* (pp. 302-307). IEEE.
- [4] Park, Y. M., Jung, Y., Yoo, H., Bae, H., & Kim, H. S. (2011). EasyApp: goal-driven service flow generator with semantic web service technologies. In *The Semantic Web: Research and Applications* (pp. 446-450). Springer Berlin Heidelberg.
- [5] soa4all. (n.d.). . Retrieved May 31, 2014, from <http://www.soa4all.eu>
- [6] Home. (n.d.). *ClickScript-Server*. Retrieved June 14, 2014, from <http://www.clickscript.ch/>
- [7] Barnaghia, P., Ganza, F., Abangara, H., Presserb, M., & Moessnera, K. Sense2Web: A Linked Data Platform for Semantic Sensor Networks.
- [8] Barnaghi, P., Wang, W., Henson, C., & Taylor, K. (2012). Semantics for the Internet of Things: early progress and back to the future. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 8(1), 1-21.
- [9] SensorMasher - Digital Enterprise Research Institute. (n.d.). . Retrieved June 14, 2014, from <http://sensormasher.deri.org/>
- [10] GWT. (n.d.). *Project*. Retrieved June 14, 2014, from <http://www.gwtproject.org/>
- [11] Drools - JBoss Community. (n.d.). *Drools - JBoss Community*. Retrieved June 14, 2014, from <http://drools.jboss.org>
- [12] Chen, L., Nugent, C., Mulvenna, M., Finlay, D., & Hong, X. (2009). Semantic smart homes: towards knowledge rich assisted living environments. *Intelligent Patient Management* (pp. 279-296). Springer Berlin Heidelberg.

- [13] Apache Jena -. (n.d.). *Apache Jena* -. Retrieved June 14, 2014, from <http://jena.apache.org/>
- [14] OWL Web Ontology Language Reference. (n.d.). *OWL Web Ontology Language Reference*. Retrieved June 14, 2014, from <http://www.w3.org/TR/owl-ref>
- [15] Guinard, D., Trifa, V., Karnouskos, S., Spiess, P., & Savio, D. (2010). Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3), 223-235.
- [16] Fernández, J. D., Martínez-Prieto, M. A., Gutiérrez, C., Polleres, A., & Arias, M. (2013). Binary RDF representation for publication and exchange (HDT). *Web Semantics: Science, Services and Agents on the World Wide Web*, 19, 22-41.
- [17] OWL-S: Semantic Markup for Web Services. (n.d.). *OWL-S: Semantic Markup for Web Services*. Retrieved June 14, 2014, from <http://www.w3.org/Submission/OWL-S/>
- [18] Web Service Modeling Ontology (WSMO). (n.d.). *Web Service Modeling Ontology (WSMO)*. Retrieved June 14, 2014, from <http://www.w3.org/Submission/WSMO/>
- [19] Domingue, J., Pedrinaci, C., Maleshkova, M., Norton, B., & Krummenacher, R. (2011). Fostering a relationship between linked data and the internet of services. In *The future internet* (pp. 351-364). Springer Berlin Heidelberg.
- [20] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures* (Doctoral dissertation, University of California).
- [21] Verborgh, R., Steiner, T., Van Deursen, D., De Roo, J., Van de Walle, R., & Vallés, J. G. (2013). Capturing the functionality of Web services with functional descriptions. *Multimedia tools and applications*, 64(2), 365-387.
- [22] Nelson's Weblog: tech / bad / whySoapSucks. (n.d.). *Nelson's Weblog: tech / bad / whySoapSucks*. Retrieved June 14, 2014, from <http://www.somebits.com/weblog/tech/bad/whySoapSucks.html>
- [23] Gonzalez, J. L., & Marcelin-Jimenez, R. (2011, May). Phoenix: A Fault-Tolerant Distributed Web Storage Based on URLs. In *Parallel and Distributed Processing with Applications (ISPA), 2011 IEEE 9th International Symposium on* (pp. 282-287). IEEE.
- [24] Linked Data. (n.d.). - *Design Issues*. Retrieved June 14, 2014, from <http://www.w3.org/DesignIssues/LinkedData.html>
- [25] Bauer, F., & Kaltenböck, M. (2011). Linked Open Data: The Essentials. *Edition mono/monochrom, Vienna*.
- [26] The Linking Open Data cloud diagram. (n.d.). . Retrieved June 14, 2014, from <http://lod-cloud.net/>

- [27] iServe - Open University. (n.d.). Retrieved June 14, 2014, from <http://iserve.kmi.open.ac.uk/>
- [28] Pipes: Rewire the web. (n.d.). *Pipes Blog RSS*. Retrieved June 14, 2014, from <http://pipes.yahoo.com/pipes/>
- [29] Microsoft Popfly. (2014, June 14). *Wikipedia*. Retrieved June 14, 2014, from http://en.wikipedia.org/wiki/Microsoft_Popfly
- [30] Deliverables. (n.d.). *Deliverables*. Retrieved June 14, 2014, from <http://www.hitech-projects.com/euprojects/amigo/deliverables.htm>
- [31] Vallée, M., Ramparany, F., & Vercoouter, L. (2005). Flexible Composition of Smart Device Services. *PSC*, 5, 165-171.
- [32] Apache JENA-SDB (n.d.). Retrieved June 14, 2014, from <http://jena.apache.org/documentation/sdb/>
- [33] Joseki: The Jena RDF Server. (n.d.). *SourceForge*. Retrieved June 14, 2014, from <http://sourceforge.net/projects/joseki/>
- [34] W3C Geospatial Ontologies. (n.d.). *W3C Geospatial Ontologies*. Retrieved June 14, 2014, from <http://www.w3.org/2005/Incubator/geo/XGR-geo-ont-20071023>
- [35] Krummenacher, R., Norton, B., & Marte, A. (2010). Towards linked open services and processes. In *Future Internet-FIS 2010* (pp. 68-77). Springer Berlin Heidelberg.
- [36] Sycara, K., Paolucci, M., Soudry, J., & Srinivasan, N. (2004). Dynamic discovery and coordination of agent-based semantic web services. *Internet Computing, IEEE*, 8(3), 66-73.
- [37] Sirin, E., Parsia, B., & Hendler, J. (2005, November). Template-based composition of semantic web services. In *Aaai fall symposium on agents and the semantic web* (pp. 85-92).
- [38] Wen, J., Xu, B., Bu, F., & Cai, H. (2010, December). A Service Composition Model Based on Business Process Template. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on* (Vol. 2, pp. 1029-1033). IEEE.
- [39] Molina, A. J., Koo, H. M., & Ko, I. Y. (2007). A template-based mechanism for dynamic service composition based on context prediction in ubicomp applications. In *Proceedings of the International Workshop on Intelligent Web Based Tools (IWBT'2007)*.

APPENDICES

APPENDIX A: EASYLIFE ONTOLOGY

```
<!DOCTYPE rdf:RDF [  
  <!ENTITY rdf    'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>  
  <!ENTITY rdfs   'http://www.w3.org/2000/01/rdf-schema#'>  
  <!ENTITY xsd    'http://www.w3.org/2001/XMLSchema#'>  
  <!ENTITY owl  'http://www.w3.org/2002/07/owl#'>  
  <!ENTITY sensor 'http://www.easylife.com/sensor#'>  
>  
  
<rdf:RDF  
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
  xmlns:rdfs="&rdfs;"  
  xmlns:xsd="&xsd;"  
  xmlns:owl="&owl;"  
  xmlns="http://www.easylife.com/sensor#"  
  xml:base="&sensor;">  
  
  <owl:Ontology rdf:about="">  
    <rdfs:comment>Sensor Ontology</rdfs:comment>  
  </owl:Ontology>  
  
  <!-- ONTOLOGY DEFINITION -->  
  
  <owl:Class rdf:ID="SensingDevice">  
  
    <rdfs:subClassOf>  
      <owl:Restriction>  
        <owl:onProperty rdf:resource="#name"/>  
        <owl:allValuesFrom rdf:resource="&xsd:string"/>  
      </owl:Restriction>  
    </rdfs:subClassOf>  
  
    <rdfs:subClassOf>  
      <owl:Restriction>  
        <owl:onProperty rdf:resource="#observes"/>  
        <owl:allValuesFrom rdf:resource="#Observable"/>  
      </owl:Restriction>  
    </rdfs:subClassOf>  
  
</owl:Class>  
</rdf:RDF>
```

```

        </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasEvent"/>
            <owl:allValuesFrom rdf:resource="#Event"/>
    </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Observable">

    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#observableType"/>
            <owl:allValuesFrom rdf:resource="#ObservableType"/>
        </owl:Restriction>
    </rdfs:subClassOf>

    <rdfs:subClassOf>
    <owl:Restriction>
        <owl:onProperty rdf:resource="#hasMethod"/>
            <owl:allValuesFrom rdf:resource="#Method"/>
    </owl:Restriction>
    </rdfs:subClassOf>
</owl:Class>

    <owl:Class rdf:ID="ObservableType">
        <owl:oneOf rdf:parseType="Collection">
            <owl:Thing rdf:about="#Temperature" />
            <owl:Thing rdf:about="#Light"/>
            <owl:Thing rdf:about="#Motion"/>
            <owl:Thing rdf:about="#Location"/>
        </owl:oneOf>
    </owl:Class>

    <owl:Class rdf:ID="Method">

        <rdfs:subClassOf>
            <owl:Restriction>
                <owl:onProperty rdf:resource="#parameterType" />
                <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>

```



```

                <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
                </owl:Restriction>
            </rdfs:subClassOf>

            <rdfs:subClassOf>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#hasInput"/>
                    <owl:allValuesFrom rdf:resource="#Input"/>
                </owl:Restriction>
            </rdfs:subClassOf>

        </owl:Class>

        <owl:Class rdf:ID="Input">

            <rdfs:subClassOf>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#parameterType"/>
                    <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
                    <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
                </owl:Restriction>
            </rdfs:subClassOf>

            <rdfs:subClassOf>

                <owl:Restriction>
                    <owl:onProperty rdf:resource="#parameterOrder"/>
                    <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
                    <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
                </owl:Restriction>
            </rdfs:subClassOf>

        </owl:Class>

        <owl:Class rdf:ID="Event">

            <rdfs:subClassOf>
                <owl:Restriction>
                    <owl:onProperty rdf:resource="#parameterType" />
                    <owl:maxCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:maxCardinality>
                    <owl:minCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:minCardinality>
                </owl:Restriction>
            </rdfs:subClassOf>

```

```
        </owl:Restriction>
    </rdfs:subClassOf>
```

```
</owl:Class>
```

```
<owl:DatatypeProperty rdf:ID="hasEvent"/>
  <owl:ObjectProperty rdf:ID="hasMethod"/>
  <owl:ObjectProperty rdf:ID="hasInput"/>
  <owl:DatatypeProperty rdf:ID="parameterOrder"/>
  <owl:DatatypeProperty rdf:ID="parameterType"/>
  <owl:ObjectProperty rdf:ID="observableType"/>
  <owl:DatatypeProperty rdf:ID="name"/>
  <owl:ObjectProperty rdf:ID="observes"/>
```

```
<!-- END OF ONTOLOGY DEFINITION -->
```

```
</rdf:RDF>
```

APPENDIX B: SAMPLE SERVICE DEFINITION FILE

```
<!DOCTYPE rdf:RDF [
  <!ENTITY rdf    'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <!ENTITY rdfs  'http://www.w3.org/2000/01/rdf-schema#'>
  <!ENTITY xsd   'http://www.w3.org/2001/XMLSchema#'>
  <!ENTITY owl 'http://www.w3.org/2002/07/owl#'>
  <!ENTITY sensor 'http://www.easylife.com/sensor'>
]>

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="&rdfs;"
  xmlns:xsd="&xsd;"
  xmlns:owl="&owl;"
  xmlns="http://www.easylife.com/sensor#"
  xml:base="&sensor;">

  <SensingDevice rdf:ID="Arduino">
    <name rdf:datatype="&xsd:string">Arduino Board</name>

    <hasEvent>
      <Event
        rdf:about="tcp://192.168.1.3:1883/sensors/light/EventCurrentValueChanged">
          <rdfs:comment>Fired when light sensor current value
            changed</rdfs:comment>
          <parameterType
            rdf:datatype="&xsd:string">type_integer</parameterType>
          </Event>
        </hasEvent>

      <hasEvent>
        <Event
          rdf:about="tcp://192.168.1.3:1883/sensors/opticalProximity/EventCurrentValueChanged"
          >
          <rdfs:comment>Fired when optical proximity sensor
            current value changed</rdfs:comment>
          <parameterType
            rdf:datatype="&xsd:string">type_integer</parameterType>
          </Event>
        </hasEvent>

    <observes>
      <Observable rdf:about="#ArduinoObservable">
        <observableType rdf:resource="#Light"/>

      <hasMethod>
        <Method rdf:about="http://192.168.1.18/light/ON">
```

```

        <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
        <rdfs:comment>Sets          light          state
ON</rdfs:comment>
        <hasInput>
            <Input    rdf:about="#ON/OFF    Status
String">
                <rdfs:comment>ON          or
OFF</rdfs:comment>
                <parameterType
rdf:datatype="&xsd;void">type_void</parameterType>
                <parameterOrder>1</parameterOrder>
                </Input>
            </hasInput>
            </Method>
        </hasMethod>
        <hasMethod>
            <Method rdf:about="http://192.168.1.18/light">
                <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
                <rdfs:comment>Sets light state</rdfs:comment>
                <hasInput>
                    <Input    rdf:about="#ON/OFF    Status
String">
                        <rdfs:comment>ON          or
OFF</rdfs:comment>
                    <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
                    <parameterOrder>1</parameterOrder>
                    </Input>
                </hasInput>
                </Method>
            </hasMethod>
            <Method
rdf:about="http://192.168.1.18/opticalProximity">
                <parameterType
rdf:datatype="&xsd:string">type_integer</parameterType>
                <rdfs:comment>Gets          proximity    sensor
value</rdfs:comment>
                <hasInput>
                    <Input rdf:about="#VOID">
                        <rdfs:comment>void</rdfs:comment>
                    <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>

```

```

        <parameterOrder>1</parameterOrder>
        </Input>
        </hasInput>
        </Method>
        </hasMethod>

        </Observable>
    </observes>

</SensingDevice>

<SensingDevice rdf:ID="HTC Android Device">
    <name rdf:datatype="&xsd:string">HTC HD2</name>

    <hasEvent>
        <Event
rdf:about="tcp://192.168.1.3:1883/sensors/hd2/LocationChanged">
            <rdfs:comment>Fired when location changed</rdfs:comment>
            <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
            </Event>
        </hasEvent>

        <hasEvent>
            <Event rdf:about="tcp://192.168.1.3:1883/sensors/hd2/LightChanged">
                <rdfs:comment>Fired when light changed</rdfs:comment>
                <parameterType
rdf:datatype="&xsd:string">type_float</parameterType>
                </Event>
            </hasEvent>

            <hasEvent>
                <Event
rdf:about="tcp://192.168.1.3:1883/sensors/hd2/ProximityChanged">
                    <rdfs:comment>Fired when proximity changed</rdfs:comment>
                    <parameterType
rdf:datatype="&xsd:string">type_float</parameterType>
                    </Event>
                </hasEvent>

                <hasEvent>
                    <Event rdf:about="tcp://192.168.1.3:1883/sensors/hd2/SwitchLight">
                        <rdfs:comment>Fires request to turn light on</rdfs:comment>
                        <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
                        </Event>
                    </hasEvent>

                <observes>

```

```

    <Observable rdf:about="#HD2PhoneObservable">
      <observableType rdf:resource="#Light"/>

      <hasMethod>
        <Method
rdf:about="tcp://192.168.1.3:1883/sensors/hd2/LightStateChanged">
          <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
          <rdfs:comment>Sets light state</rdfs:comment>
          <hasInput>
            <Input rdf:about="#ON/OFF Status
String">
              <rdfs:comment>ON or
OFF</rdfs:comment>
            </Input>
          </hasInput>
          <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
          <parameterOrder>1</parameterOrder>
        </Method>
      </hasMethod>

    </Observable>
  </observes>

</SensingDevice>

<SensingDevice rdf:ID="DeviceLightSensor">
  <name rdf:datatype="&xsd:string">Elif Beyza's Room Light Sensor</name>
  <observes>
    <Observable rdf:about="#ExampleLightObservable">
      <observableType rdf:resource="#Light"/>

      <hasMethod>
        <Method
rdf:about="http://localhost:8888/sensors/light/GetCurrentValue">
          <parameterType
rdf:datatype="&xsd:string">type_double</parameterType>
          <rdfs:comment>Gets current
value</rdfs:comment>
          <hasInput>
            <Input rdf:about="#VOID">
              <rdfs:comment>void</rdfs:comment>
            </Input>
          </hasInput>
          <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
          <parameterOrder>1</parameterOrder>
        </Method>
      </hasMethod>
    </Observable>
  </observes>
</SensingDevice>

```

```

        </hasInput>
      </Method>
    </hasMethod>

    <hasMethod>
      <Method
rdf:about="http://localhost:8888/sensors/light/GetMeanValueForDay">
        <parameterType
rdf:datatype="&xsd:string">type_double</parameterType>
        <rdfs:comment>Gets mean value for the
day</rdfs:comment>
        <hasInput>
          <Input rdf:about="#DAY">
            <rdfs:comment>Day of the
month</rdfs:comment>
          <parameterType
rdf:datatype="&xsd:string">type_integer</parameterType>
            <parameterOrder>1</parameterOrder>
          </Input>
        </hasInput>
      </Method>
    </hasMethod>

  </Observable>
</observes>

  <hasEvent>
    <Event
rdf:about="tcp://192.168.1.3:1883/sensors/light/EventCurrentValueChanged">
      <rdfs:comment>Fires event if light changes at least by 1
%</rdfs:comment>
      <parameterType
rdf:datatype="&xsd:string">type_integer</parameterType>
    </Event>
  </hasEvent>

  <hasEvent>
    <Event
rdf:about="tcp://192.168.1.3:1883/sensors/light/EventHighNoise">
      <rdfs:comment>Fires event when the sensor noise is
high</rdfs:comment>
      <parameterType
rdf:datatype="&xsd:string">type_integer</parameterType>
    </Event>
  </hasEvent>

</SensingDevice>

```

```

<SensingDevice rdf:ID="DeviceLocationProvider">
  <name rdf:datatype="&xsd:string">Elif Beyza's Phone Location Provider</name>
  <observes>
    <Observable rdf:about="#ExampleLocationProvider">
      <observableType rdf:resource="#Location"/>

      <hasMethod>
        <Method
rdf:about="http://localhost:8888/sensors/location/GetCurrentLocation">
          <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
          <rdfs:comment>Gets current
value</rdfs:comment>
          <hasInput>
            <Input rdf:about="#CoordinateSystem">
              <rdfs:comment>Coordinate
system</rdfs:comment>
              <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>

              <parameterOrder>1</parameterOrder>
              </Input>
            </hasInput>
            <hasInput>
              <Input rdf:about="#Datum">
                <rdfs:comment>Datum</rdfs:comment>
                <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>

                <parameterOrder>2</parameterOrder>
                </Input>
              </hasInput>
            </Method>
          </hasMethod>

        </Observable>
      </observes>
    </SensingDevice>

  <SensingDevice rdf:ID="DeviceAudioProvider">
    <name rdf:datatype="&xsd:string">Elif Beyza's Computer Audio
Provider</name>
    <observes>
      <Observable rdf:about="#ExampleAudioProvider">
        <observableType rdf:resource="#Location"/>

        <hasMethod>
          <Method
rdf:about="http://localhost:8888/sensors/audio/GetAudioStream">

```



```

                                <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
                                <rdfs:comment>Gets          last          audio
stream</rdfs:comment>
                                <hasInput>
                                    <Input rdf:about="#VOID">

                                <rdfs:comment>void</rdfs:comment>
                                                                <parameterType
rdf:datatype="&xsd:string">type_void</parameterType>
                                <parameterOrder>1</parameterOrder>
                                                                </Input>
                                                                </hasInput>
                                                                </Method>
                                                                </hasMethod>

                                                                </Observable>
                                                                </observes>
                                                                </SensingDevice>

<SensingDevice rdf:ID="GeoLocationService">
    <name rdf:datatype="&xsd:string">GeoLocation Service</name>
    <observes>
        <Observable rdf:about="#GeoLocationService">
            <observableType rdf:resource="#Location"/>

            <hasMethod>
                <Method
rdf:about="http://localhost:8888/sensors/geolocation/CalculateDistance">
                    <parameterType
rdf:datatype="&xsd:string">type_double</parameterType>
                    <rdfs:comment>Calculates distance between two
locations and returns distance in meters</rdfs:comment>
                    <hasInput>
                        <Input rdf:about="#Location1">
                            <rdfs:comment>First
Location</rdfs:comment>
                                <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>
                                <parameterOrder>1</parameterOrder>
                                                                </Input>
                                                                </hasInput>
                                                                <hasInput>
                                                                    <Input rdf:about="#Location2">
                                                                        <rdfs:comment>Second
Location</rdfs:comment>
                                                                <parameterType
rdf:datatype="&xsd:string">type_string</parameterType>

```

```
<parameterOrder>2</parameterOrder>
      </Input>
    </hasInput>
  </Method>
</hasMethod>

</Observable>
</observes>
</SensingDevice>

</rdf:RDF>
```


TEZ FOTOKOPİ İZİN FORMU

ENSTİTÜ

Fen Bilimleri Enstitüsü

Sosyal Bilimler Enstitüsü

Uygulamalı Matematik Enstitüsü

Enformatik Enstitüsü

Deniz Bilimleri Enstitüsü

YAZARIN

Soyadı : Özpınar
Adı : Mustafa
Bölümü : Bilişim Sistemleri

TEZİN ADI (İngilizce) : A flexible semantic service composition framework
for pervasive computing environments

TEZİN TÜRÜ : Yüksek Lisans Doktora

1. Tezimin tamamı dünya çapında erişime açılsın ve kaynak gösterilmek şartıyla tezimin bir kısmı veya tamamının fotokopisi alınsın.
2. Tezimin tamamı yalnızca Orta Doğu Teknik Üniversitesi kullanıcılarının erişimine açılsın. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.)
3. Tezim bir (1) yıl süreyle erişime kapalı olsun. (Bu seçenekle tezinizin fotokopisi ya da elektronik kopyası Kütüphane aracılığı ile ODTÜ dışına dağıtılmayacaktır.)

Yazarın imzası

Tarih