An Approach to Automatic Adaptation of DAiSI Component Interfaces

Yong Wang and Andreas Rausch
Department of Informatics
Technical University Clausthal
Clausthal-Zellerfeld, Germany
e-mail: yong.wang, andreas.rausch@tu-clausthal.de

Abstract— **The Dynamic Adaptive System Infrastructure (DAiSI) is a platform which supports dynamic adaptive system. DAiSI can change its behavior at runtime. Behavioral changes can be caused by user's needs, or based on context information if the system environment changes. It is a run-time infrastructure that operates on components that comply with a DAiSI-specific component model. The run-time infrastructure can integrate components into the system that were not known at design-time. Communication between components in DAiSI is supported by services. Services of components are described by domain interfaces, which have to be specified by the component developer. Components can use services of other components, if the respective required and provided domain interfaces of components are compatible. However, sometimes services that have been developed by different developers can do the same thing, e.g., provide the same data or operations, but they are represented by different syntactic. Therefore, in a previous article, we present an approach which enables the use of syntactically incompatible service by using an ontology-based adapter that connects services, which provide the same data in different format. In this paper we use an existing ontology to semantically describe interfaces of components and present an improved algorithm using SPARQL and reasoning to discover interfaces in triplestore. In addition, we propose to use the historical data to predict the best suitable interface.**

Keywords—component models; self-adaptation; dynamic adaptive systems; ontology.

I. INTRODUCTION

An increasing interest in dynamic adaptive systems could be observed in the last two decades. A platform for such systems has been developed in our research group for more than ten years. It is called Dynamic Adaptive System Infrastructure (DAiSI). DAiSI is a component based platform that can be used in self-managed systems. Components can be integrated into or removed from a dynamic adaptive system at run-time without causing a complete application to fail. To meet this requirement, each component can react to changes in its environment and adapt its behavior accordingly.

Components are developed with a specific use-case in mind. Thus, the domain interfaces describing the provided and required services tend to be customized to very specific requirements of an application. This effect limits the re-use of existing components in new applications. The re-use of existing components is one key aspect in software engineering for minimize re-developing existing components. One measure to aspect is to increase reusability. However, re-using components in other application contexts than they have been originally developed for is still a big challenge. This challenge gets even bigger, if such components should be integrated into dynamic adaptive systems at run-time.

A valid approach to tackle this challenge is adaptation. Because of the nature of DAiSI platform, in DAiSI applications, DAiSI components are considered as black boxes. Capabilities and behavior of DAiSI components are specified by interfaces that describe required and provided services. In this approach, we suggest a solution to couple provided and required services that are syntactically incompatible but semantically compatible. To be able to utilize specific provided services that offer the needed data or operations on a semantical level, we suggest constructing an adapter that enables interaction between services that are only compatible on some semantical level [1].

The goal of an adapter is to enable communication between two formerly incompatible components. In order to achieve a common understanding between components, a common knowledge-base is needed. In this work we use ontology as the common knowledge-base to represent services and the schema of data. Ontology and run-time information represented by an ontology language are stored in triplestore. Required interfaces can discover/map the representation of provided interfaces in the database by using a Query Engine. To illustrate that this approach is suitable for adaptive systems, we extend our DAiSI infrastructure by an ontology-based adapter engine for service adaptation.

To strengthen the dynamic adaptive nature of the DAiSI, we generate these adapters at run-time. We argue that these adapters cannot be generated at compile time, because the different components that should interact with each other are not known at compile time, but only at integration time, which is the same requirement just like dynamic adaptive systems.

In this work, we improve the algorithm for discovery of the provided interfaces with using semantic query language and reasoning. Programming interfaces with semantic notation are translated firstly into triplestore readable semantic format and then stored into the triplestore. Required interfaces can discover the required interfaces with help of their semantic description and relation of used ontologies. As opposed to by discovery one-to-one relation of entities of function, input- and output parameters between provided and required interfaces in OWL file, discovery of provide interfaces is supported by using SPARQL, which can represent the entire required interface based on the graph pattern including filter function. Especially, in this work, we use the historical data of output parameters of interface to predicate and filter for discovered provided interfaces.

To illustrate this approach, we use a parking space example to show how to create an adapter to enable interaction between semantical identical components that have been developed by different developers or for different applications.

The rest of this paper is structured as follows: In Section II, we describe the already sketched problem in more detail. Section III gives an overview of relevant related work. In Section IV, we give a short overview of the DAiSI component model and a few hints for further reading. Section V explains structure of the adapter engine and adaptation processes. In Section VI, we show Interface description with using ontology layer. Section VII explains the discovery process of provided services based on query engine and triplestore, before the paper is wrapped up by a short conclusion in Section VIII.

II. Problem Description

Specifications of interfaces between the components in a dynamic adaptive application are mostly the early stage of developing process. Specified interfaces could not be changed, whenever a dynamic adaptive application is developed. They are very domain specific and their definition is driven by the use cases of the future application in mind. To ensure many applications run in a shared context with other applications from different domains, all specified interfaces are centrally managed in a library that is so called interface pool.

It is a tedious task to harmonize one large interface pool among different developers from different vendors that operate in different domains. It often causes results in a slow standardization process. This slows the development process down and, especially in dynamic adaptive systems, diminishes the chances for the development of new applications. Developers will in those cases often start their own interface pool. This, on the other hand, reduces the chances to re-use existing components from other domains.

Additionally, the management of one central interface pool in a distributed system does not scale well. One way to mitigate this issue would be a de-centralization of management of interfaces. To tackle these challenges, we propose to keep the domain interfaces in de-centralization and allow the domain interfaces between different domains unharmonized.

To be able to harmonize services across domains, every interface pool is required to use ontology. By either merging these ontologies later, or by using distributed ontologies we ensure that interfaces from different interface pools base on a common knowledge. Based on common knowledge, on-the-fly generated adapters enable to interaction syntactically incompatible services across domains.

Services of components can be provided by implementing domain interfaces, so called provided services. Desired services of components can be specified by other domain interfaces, so called required services. In this case, required and provided interfaces could not be the same domain interface. In order to build communication between provided and required services, they must stand in relation to each other, mapping between provided and required services are necessary. In the graphical notation of DAiSI components, provided services are marked as filled circles, required services are noted as semi-circles (similar to the UML lollipop notation [2]) and the relation between those two services are depicted by interfaces notations in domain area and across the DAiSI components linking interface for provided and required services (cf. Figure 1).

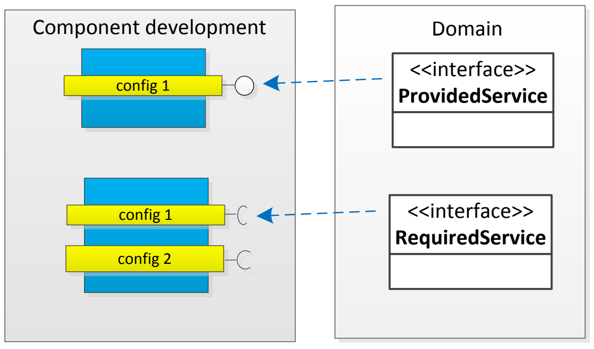


Figure 1. DAiSI components and domain specific interface definitions.

We propose that services that are semantically compatible, but lack compatibility on a syntactical level, should be usable. For example, an application wants to use parking spaces information, which is supported by different system providers. Each provider has its own service, they are mostly not compatible. The lack of compatibility can be covered by the following three types of incompatibility: Different Naming, Different Data Structure, and Different Control Structure. Adapters between the different services can be generated. We believe that we can connect all semantically compatible but syntactically incompatible services using adaptation based on these three types. We illustrate the three types of incompatibility below with parking use case.

A. Different Naming

By "Different Naming" we denote cases in which the names of interfaces describing services or names of functions do not match. While they are syntactically different, their names share the same semantics and could be used synonymous.

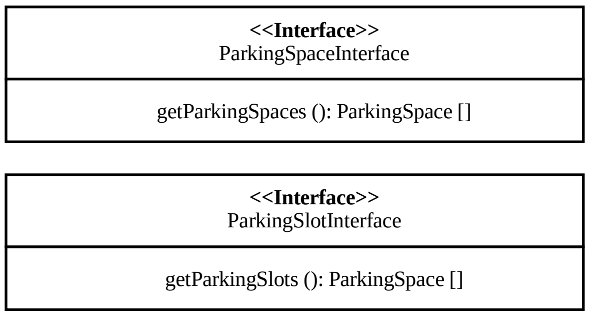
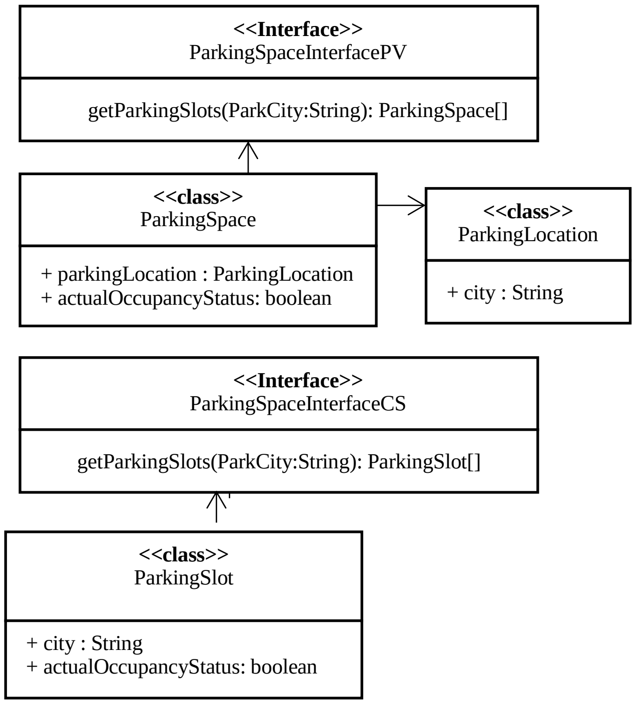

<<Interface>>
ParkingSpaceInterface
getParkingSpaces (): ParkingSpace []
<<Interface>>
ParkingSlotInterface
getParkingSlots (): ParkingSpace []

Figure 2. Example of two interfaces with Different Naming.

The first example, depicted in Figure 2, shows two interfaces: `ParkingSlotInterface` and `ParkingSpaceInterface`. Each of them defines one of the following methods: `getParkingSlots`, and `getParkingSpaces` respectively. The names of their input- and output parameter of the methods are identical. They are named differently, but offer the same service.

B. Different Data Structure

In this type of incompatibility, the names of the interfaces and their functions are the same. However, the parameters differ in their data types. Moreover, the encapsulated data is similar and the data structures can be mapped to each other. In Figure 3, in the Different Data Structure example an interface `ParkingSpaceInterfacePV` is depicted. It contains a function `getParkingSlots` which returns a parameter of the type `ParkingSpace`. In the interface `ParkingSpaceInterfaceCS`, there is a function get-`ParkinSlots`, with the same name but different output parameter `ParkingSlot`**.**

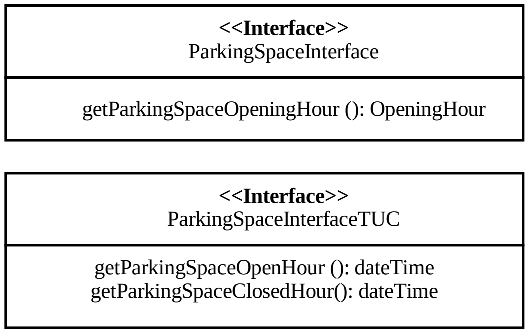


<<Interface>>
ParkingSpaceInterfacePV
getParkingSlots(ParkCity:String): ParkingSpace[]
<<class>>
ParkingSpace
+ parkingLocation : ParkingLocation
+ actualOccupancyStatus: boolean
<<class>>
ParkingLocation
+ city : String
<<Interface>>
ParkingSpaceInterfaceCS
getParkingSlots(ParkCity:String): ParkingSlot[]
<<class>>
ParkingSlot
+ city : String
+ actualOccupancyStatus: boolean

Figure 3. Example of two interfaces with a Different Data Structure.

C. Different Control Structure

In this case, the functions between provided interface and required interface have not one to one relation. One function could be mapped to many functions. To obtain valid results, the control structure has to be modified. In the example in Figure 4, two interfaces `ParkingSpaceInterface` and `ParkingSpaceInterfaceTUC` are given. By definition, an opening hour should be composed of the start– and the end time name of a parking space. As such, the two functions `getParkingSpaceOpenHour` and `getParkingSpaceClosedHour` from the `ParkingSpaceInterfaceTUC` interface in comparison provide the same information as `getParkingSpaceOpeningHour` from the `ParkingSpaceInterface` interface. Therefore, workflow of functions as a composite process is needed. A composite process specifies control structure of functions involved in the composition, in this example, a sequence control workflow is need for `getParkingSpaceOpenHour` and `getParkingSpaceClosedHour` to provide an integrated result to `getParkingSpaceOpeningHour.`



<<Interface>>
ParkingSpaceInterface
getParkingSpaceOpeningHour (): OpeningHour
<<Interface>>
ParkingSpaceInterfaceTUC
getParkingSpaceOpenHour (): dateTime
getParkingSpaceClosedHour(): dateTime

Figure 4. Example of two interface requiring Different Control Structures.

To enable the mapping between interfaces, a common knowledge-base is needed. Because of the issues stated earlier, it should not be mandatory that both sets of interface definitions are of the same domain. A common knowledge base defined by ontologies can be generated using merging or other integration mechanisms on classical ontology languages or by using a distributed ontology language. Both interfaces do not need to contain information on how to interpret the data of each other. That means that interfaces can be developed independently, without knowing anything about a possible re-use in another system.

III. Related Work

A dynamic adaptive system is a system that adjusts its structure and behavior according to the user's needs or to a change in its system context at run-time. The DAiSI is one example of an infrastructure for dynamic adaptive systems [3][4][5][6]. It has been developed over more than a decade by a number of researchers. This work is based on DAiSI and extends the current run-time infrastructure.

According to a publication of M. Yellin and R. Storm, challenges regarding behavioral differences of components have been tackled by many researchers [7]. The behavior of the interface of a component can be described by a protocol with the help of state-machines. The states of two involved components are stored and managed by an adapter. In further steps of this method, ontology is used as a language library to describe a component's behavior. To automate the adaptation of services, a semi-automated method has been developed to generate adapters with the analysis of a possible behavioral mismatch [8][9].

Another solution for the connection of semantically incompatible services is presented in [8]. They use buffers for the asynchronous communication between services and translate the contents of those buffers to match the syntactical representations of the involved services. The behavioral protocols of services can automatically be generated with a tool that is based on synthesis– and testing techniques [10]. Ontologies are used in their method to describe the behavior of components and to create a tool for automated adaptation [11]. Mapping-driven synthesis focuses on mapping of actions of the interfaces of services. Interfaces are identified by correspondence between actions of the interface of component based on the ontology and reasoning [12]. The data mapping is still not considered in this approach. All Approaches mentioned above are based on state-machine. However, some components require a very complex state-machine; the development of which can easily become very expensive. Thus, in this work, we present another way that does not rely on the consideration of dependencies within the behavior or the involved interfaces.

The method of transformation of an ontology into interfaces is already integrated into Corba Ontolingua [13]. With this tool an ontology can be transformed into the interface definition language (IDL). A. Kalyanpur [14] has developed a method which allows automatic mapping from Web Ontology Language (OWL) to Java. The Object Management Group (OMG) [15] has defined how to transform the Unified Markup Language (UML) into ontology. With their method, UML classes are first converted into a helper class and then transformed into ontology [16]. G. Söldner [17] has shown how to transform the UML itself into ontology. A downside of the above methods: The interface and the ontology have a strong relation. If a developer changes the ontology, all interfaces which are linked to this ontology have to be modified. In this work, we decouple this strong relation. Alternating a part of the ontology now only affects the interfaces directly linked to the specific part.

Another approach for semantically described Web service is pressed in [18][19]. Web Service Modeling Ontology (WSMO) is ontology that can be used to describe various aspect related to semantic Web Services. Web Ontology Language for Services (OWL-S) is an ontology for describing Web services. It consists of three elements, ServiceProfile, ServiceModel, and ServiceGrounding. Because of the similar structure of Web service and program interface, we consider OWL-S useful for semantic representation of programming interface of DAiSI components.

Matching and merging existing ontologies is still a big challenge regarding speed and accuracy. To simplify this, many application interfaces (APIs) have been developed, e.g., Agreement Maker [20] and Blooms [21]. Most of them follow a survey approach [22], or use data available on the Internet [23]. Many methods are used to match entities to determine an alignment, like testing, heuristics, etc. To improve accuracy, many of them use third-party vocabularies such as WordNet or Wikipedia. However, we simply use ontology merging in our approach and we did not conduct further research on the challenges mentioned.

IV. The DAiSI Component Model

The DAiSI component model can best be explained with a sketch of a DAiSI component. Figure 5 shows a DAiSI component. The blue rectangle in the background represents the component itself. The provided and required services are depicted with full– and semi circles, as stated earlier. The dependencies between these two kinds of services are depicted by the yellow bars. They are called component configurations. At run-time, only one component configuration can be active. Being active means that all connected, required services are present and consumed (the dependencies could be resolved), and the provided services are being produced. To avoid conflicts, the component configurations are sorted by quality with the best component configuration noted at the top (Conf1 in Figure 5) and the least favorable one noted at the bottom (Conf2 in our example). The following paragraphs explain the DAiSI component model, depicted in Figure 5.

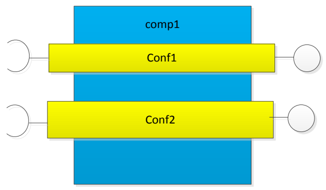


Figure 5. A DAiSI component.

The component model is the core of DAiSI and has been covered in much more detail in [24]. The component configurations (yellow bars) are represented by class with the same name. It is associated to a `RequiredServiceReferenceSet`, which is called a set to account for cardinalities in required services. The provided services are represented by the `ProvidedService` class. Interface roles, represented by `InterfaceRole`, allow the specification of additional constraints for the compatibility of interfaces that use run-time information, bound services and the internal state of a component, and are covered.

To be able to narrow the structure of a dynamic adaptive system down, blueprints of possible system configurations can be specified. The classes `Application`, `Template`, `RequiredTemplateInterface`, and `ProvidedTemplateInterface` are the building blocks in the component model that are used to realize application architecture conformance. One `Application` contains a number of `Templates`, each specifying a part of the possible application. A `Template` defines (needs and offers) `RequiredTemplateInterfaces` and `ProvidedTemplateInterfaces` which refer to `DomainInterfaces` and thus form a structure which can be filled with actual services and components by the infrastructure. More details about templates and application architecture conformance in the DAiSI can be found in [24].

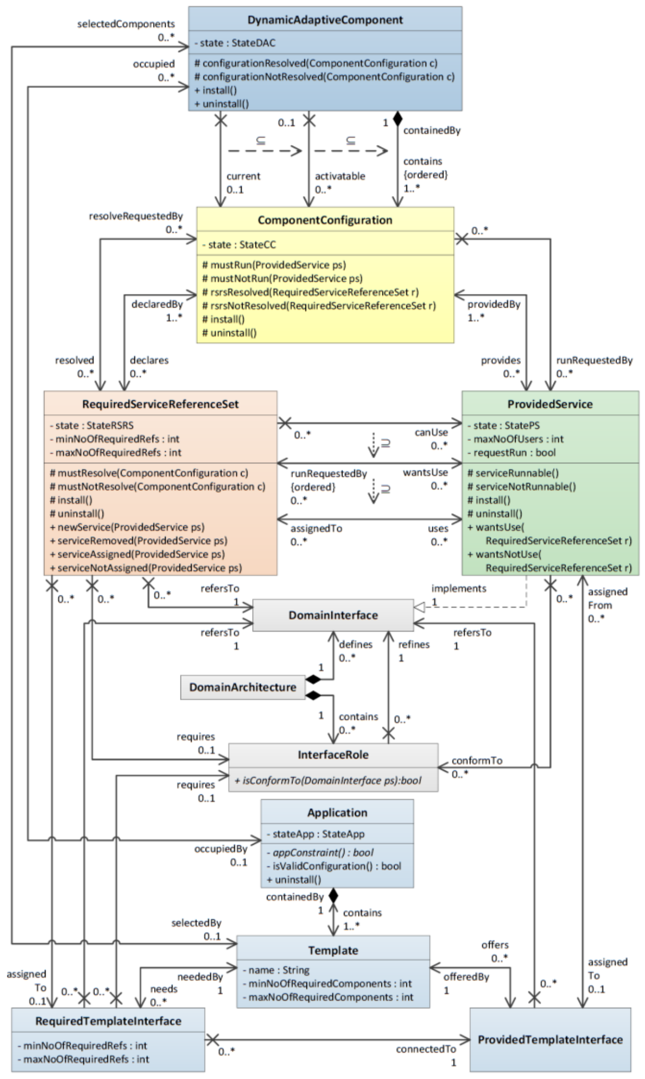


Figure 6. The DAiSI component model.

The DAiSI infrastructure is composed of the DAiSI component model, a registration service, which works like a directory for running DAiSI components, and a configuration service which manages how provided– and required services are connected to each other and what component configurations are marked as active. The configuration service constantly checks (either periodically, or event-driven), if the current system configuration (active component configurations, component bindings, etc.)) can be improved.

For the adaptation of syntactical incompatible services, we added a new infrastructure service: The adapter engine. The adapter engine keeps track of all provided and required services in the system. Whenever a new DAiSI component enters the system, the adapter engine analyzes its provided services and generates adapter components (which are DAiSI components themselves) to all syntactically incompatible, but semantically compatible services. We will describe this process in the following in more detail.

V. DAiSI Komponent for Adaptation

In this section, we show the basic concept of adapter generation based on Java programming language and structure of the adapter engine. In the end we present process for adaptation in DAiSI platform.

A. *Basic principle of the adapter*

```
//Interface of provided service
public interface ParkingSpaceInterfacePV {
 ArrayList<ParkingSpace> getParkingSpaces(String
parkCity);
}
//Implement of provider interface
public class ParkingSpacePV
        implements ParkingSpaceInterfacePV{
  public ArrayList<ParkingSpace>
             getParkingSlots(String parkCity) {
        }

//Interface of Required Service
public interface ParkingSpaceInterfaceCS {
  ArrayList<ParkingSlot> getParkingSlots(String park-
City);
}
```

Figure 7 Interfaces of service and their implement.

```
// generated adapter
public class generatedAdapter
                implements ParkingSpaceInterfaceCS {
public ArrayList<ParkingSlot>
        getParkingSlots(String parkCity)
     {
        ParkingSpaceInterfacePV ps;
        ArrayList<ParkingSpace> arrParkingSpace;
        ArrayList<ParkingSlot> arrParkingSlots;
                arrParkingSpace =
        ps.getParkingSlots(parkCity);
        arrParkingSlots =
        adapterEngine.mapping(arrParkingSpace);
        return arrParkingSlots;
      }
}
```

Figure 8. Basic principle of the adapter between two interfaces.

```
//Usage of generated adapter of required component
public class requiredComp {
        generatedAdapter adapter;
        ParkingSpaceInterfaceCS cs = adapter;
}
```

Figure 9. Provided interface and connection of adapter.

Adapter is a DAiSI component, which uses provided interface of provided component and implements required interface for required component to manage communications between provided interface and required interface. Required interface of required component uses provided interface of adapter to access provided interface of provided component. Figure 8 shows an example adapter component. The provided service of the adapter component class `generatedAdapter` implements the required interface `ParkingSpaceInterfaceCS` that is shown in Figure 7. The implementation of function `getParkingSlots` of provied interface in the adapter calls the function `getParkingSlots`, which provided by provided interface of provided component. The return of function `getParkingSlots` of required interface are mapped to function of required interface the through the function adapter Engine mapping. **Fehler! Verweisquelle konnte nicht gefunden werden.** shows connection of the adapter and the component with required interface.

B. *Structure of the adapter engine*

Figure 10 shows the structure of the adapter engine. The task of adapter engine is generating adapter based on semantic description of provided and required interfaces at runtime.

The information collector aggregates the information from provided– and required services (e.g., interface, methods, parameters, and return types) and then translates into knowledge representation language such as Resource Description Framework (RDF) and Web Ontology Language (OWL). In this approach, semantic descriptions of interfaces and related ontology are managed by a central triplestore which is a database for storage RDF triples.

The component Service Discovery discovers provided interfaces for a required interface in triplestore through SPARQL queries and screen out best candidates.

The component Mapper compares the discovered required and provided interface based on sematic descriptions of both interfaces and exports an assignment list, which maps the information from provided services to required services. Mapping of many-to-many relationship of functions is not supported by this work.

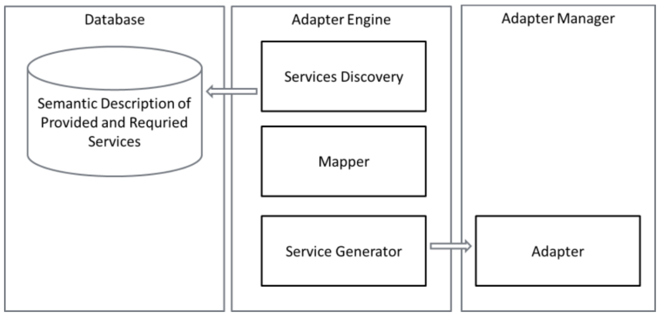


Figure 10. Structure of the adapter engine.

The Service Generator receives the assignment lists from the Mapper and generates a new DAiSI component, which can use the provided interface and implements the required interface. The Adapter Manager is a DAiSI component. It keeps track of the lifecycle of every DAiSI component adapter. Whenever a DAiSI component (provided or required) leaves the system, the adapter Manager destroys all generated adapters related to the DAiSI component and thereby removes them from the system.

C. *Adaptation of DAiSi component*

Figure 11 shows the process for involved DAiSI component. The component `comp.b` contains a configuration config1, which requires an interface A. The configuration `config1` of component `comp.b` runs at state NOT_RESOLVED as long as it cannot find an interface with same syntax or semantically compatible interface in the system. When the component `comp.a` enters the system, semantic descriptions of provided services are registered into the knowledge-base, this means, the description of service B can be found in the knowledge-base. After this, the service B is provided in the system. Adapter engine can now discover description of service B in triplestore. If service B and service A is syntactically and/or semantically compatible, the adapter engine could check another candidate, which wants to use service `B`, if number of components which service B can only serve are limit. As long as a free place is provided by service B, thus, it generates an adapter – a DAiSI component called adapter, which requires the service `B` and provides service `A`. The DAiSI configuration service connects `comp.b` to adapter and adapter to `comp.a`. The dependency of `comp.b` could be resolved.

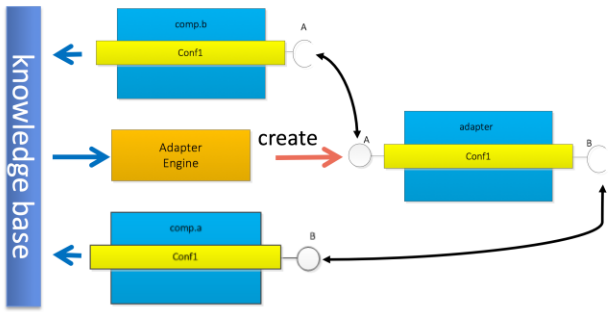


Figure 11. Adaptation process with adapter engine.

VI. Semantic Description of DAiSI Components Interface

Description of a machine-readable interface with all relevant semantical information is a key aspect of our concept. In this section we show how to describe a interface semantically.

There are two abstract semantic levels in our system, software level, which contains programming code and instance of parameter (data), semantic level, which provides semantic representation of programming code and data. Required Interface can discover provided interface in triplestore based on his semantic representations. Figure 12 shows interface GetParkingSlot noted with Parking Ontology is represented by ParkingSpace Interface in T-Box. Instances of parameters of interface GetParkingSlot is descripted in A-Box with TUCParkSSE, which link to ParkingSpaceInterface in T-Box.

For our system, we use a four-layer ontolgy structure for the construction of the knowledge-base. The four layers are part of two groups A-Box and T-Box. T-Box describes the concepts of domain in terms of controlled vocabularies, e.g., classes, properties and relationship of classes. A-Box contains the knowledge of instance of domain concepts, e.g., instance of classes. The basic knowledge is defined in the T-Box group. Such knowledge can be divided in different upper ontologies. Corresponding of ontologies can be merged, if different dynamic adaptive systems are being associated. Merging ontologies is a different research area on which we do not focus, however, the available results are sufficient for our work.

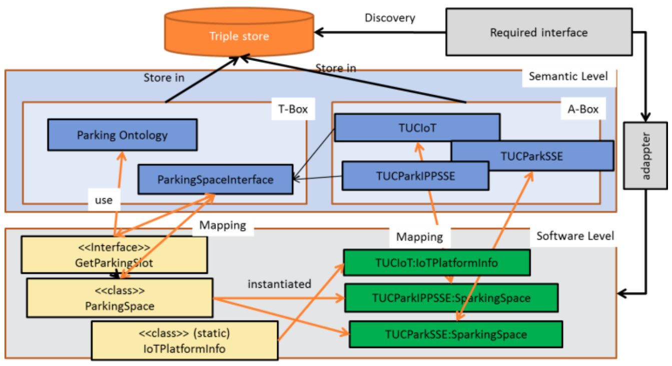


Figure 12. Overview of relation of semantic level and software level.

Figure 13 shows the four-layer ontology structure. The Basic Thing and Domain independent Ontologie, called Upperontolgoy layer, define the basic knowledge, which can be widely used in different domains and has already got agreement by many committees, such as OWL-S, schema.org, Dbpedia, etc. Domain ontology describes the domain related vocabularies. Every application or domain can define its own domain ontology. In the Domain Ontology, which is the second, or middle layer, all necessary definitions can be found that are relevant for an application. In our approach, we develop a parking ontolgoy for our application to extentent the vocabularies for the parking relevant services. The layer of the Application dependent ontology is the lowest level in T-Box. It represents the code of domain interfaces, more precisely their names, methods, parameters and return types. The domain dependent and independant ontology could be used in the application ontology in order to enriche the represention of services. This three-layer ontology in T-Box has the main advantage that every part can be developed separately. Every fragment of a layer can be merged with other fragments using ontolgy-merging and ontolgy-mapping. In addition, data, e.g., instance of java class or vaule of parameter, can be semantically represented in the A-Box.

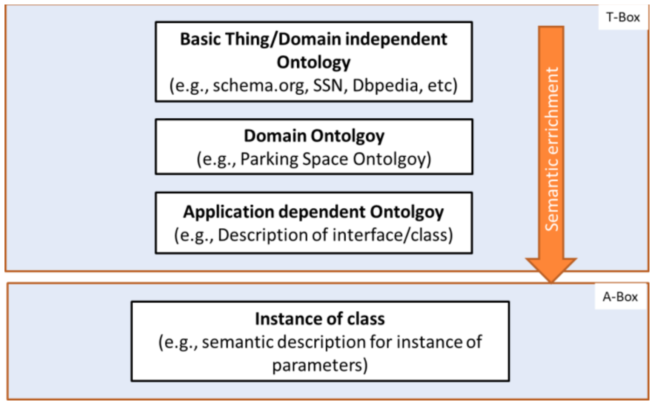

T-Box
Basic Thing/Domain independent Ontology
(e.g., schema.org, SSN, Dbpedia, etc)
Domain Ontolgoy
(e.g., Parking Space Ontolgoy)
Application dependent Ontolgoy
(e.g., Description of interface/class)
Semantic errichment
A-Box
Instance of class
(e.g., semantic description for instance of parameters)

Figure 13. Layers ontology structure.

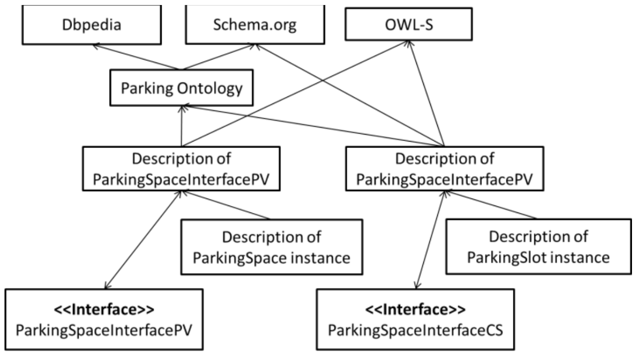

Dbpedia
Schema.org
OWL-S
Parking Ontology
Description of ParkingSpaceInterfacePV
Description of ParkingSpaceInterfacePV
Description of ParkingSpace instance
Description of ParkingSlot instance
<<Interface>> ParkingSpaceInterfacePV
<<Interface>> ParkingSpaceInterfaceCS

Figure 14. Data structure of the example.

Figure 14 shows the layout of the ontologies for the parking use case. We used upmost ontologies, OWL-S, Dbpedia and Schema.org. The Parking ontolgoy is delevoped based on the Dbpedia and schema.org. To describe interfaces, OWL-S is needed to define relations for representation of the interfaces, like methods, relation between parameters and methods. Paramter and return tpyes are created as new ontolgoy and directly mapped to the existing ontolgoy like Parking Ontolgoy, Dbpedia, etc. Represention of interface has no limit to usage of ontologies. Every information in any of the ontologies can be used in any interface. Semantic description of interface and java Interface can be bi-directional translated. Each description of interface could have an ontology item, in order to facilitate management of the ontology data in datastore such as updata, remove, modification. Run-time information such as instance of input and output parameters, so called datatyp object, could be descripted with owl instance of owl class, which can be found in T-Box. In our approach, we use historical datatyp objects to increase accuracy of services discovery. The following examples show how the Ontology is defined.

A. Domain independent ontology

Domain independent ontologies could be used for different domains. They are independent on domains and commonly are upper ontology to define basic vocabularies and data schema, e.g., schema.org, which is schemas for structured data on the Internet, Semantic Sensor Network Ontology (SSN), which can be used to describe sensor, observations and related concepts. In our approach, Domain independent ontology can be used to directly describe interface and to link to the domain dependent ontology. Especially, OWL-S is an important part for description of interfaces.

OWL-S is one ontology for describing Web services. The essence of semantic Web Servic description is to realize automatic web Service discovery, which shares the same goals of automatic adaptation of DAiSI components in our approach. Three elements of OWL-S, ServiceProfile, ProcessModel and ServiceGrounding, can be well mapped to DAiSI services. ServiceProfile uses for publishing and discovering services. ProcessModel describes in detail for the operation of a service. ServiceGrounding provides details of how the message service interoperability. In particular, because of similarity of ServiceProfile and interface, ServiceProfile can be used to describe programming interface.

Figure 15 shows relation of OWL-S and programming interface. Interface name can be descripted with Ontology Serviceprofile, which are depicted by anelliptical shape. Function name corresponds to process and input and output parameter corresponds to process:input and process:ouput.

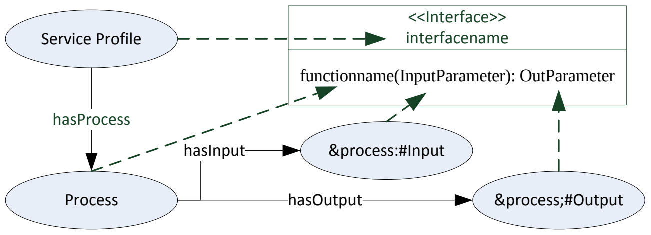


Figure 15. Relationship of programming interface and process in OWL-S.

B. Application Ontology Parking Ontolgoy

Unfortunately, domain independent ontology cannot cover requirements of vocabularies in every individual domain. We need to develop domain ontologies to meet the demand. Domain ontology can use the independent domain ontology to increase its reusability.

In our approach, we define a Parking Ontology. Parking Ontology is an ontology, which describes Parking space relevant issues, such as location, usage, ticket, opening hours, etc. Figure 16 shows a fragment of Parking Ontology, which contains relevant parts for the example in this paper. ParkingSpace is the main part, it relates to ParkingLocation with OWL objectproperty parkingLocation and StatusOfParkingSpot with statusOfParkingSport. ParkingLocation is static information.Static means, the information is not modified in run-time, mostly the value is stored in database or local in device. StatusOfParkingSpot is run-time information, which's value can be changed at run-time.

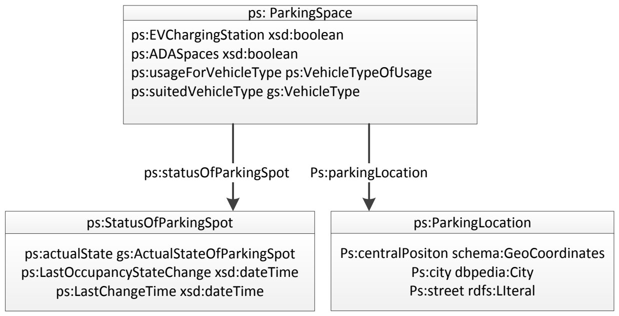

ps: ParkingSpace
ps:EVChargingStation xsd:boolean
ps:ADASpaces xsd:boolean
ps:usageForVehicleType ps:VehicleTypeOfUsage
ps:suitedVehicleType gs:VehicleType
ps:statusOfParkingSpot
Ps:parkingLocation
ps:StatusOfParkingSpot
ps:actualState gs:ActualStateOfParkingSpot
ps:LastOccupancyStateChange xsd:dateTime
ps:LastChangeTime xsd:dateTime
ps:ParkingLocation
Ps:centralPositon schema:GeoCoordinates
Ps:city dbpedia:City
Ps:street rdfs:LIteral

Figure 16. Relationship of programming interface and process in OWL-S.

C. Descrption of Interface

To semantically understand programming interfaces, they should be translated into semantic technology supported data format. In fact, the chosen sematic language has big influence for interface discovery. On the one hand, translation between semantic data formats is supported mostly only in one direction, e.g., JSON-LD can be translated into RDF/XML, but it is difficult to translate it back to its original format. On the other hand, format of results of the translations from different language of same programming interface are usually not identical. Therefore, in our approach, we use only semantic Language RDF/XML, so that the system is kept more harmonious.

In this work, we use OWL-S (implemented in RDF/XML) to descript an interface. The descriptions are separated into 3 parts, description interface, description function, description parameters.

1) Interface description

Figure 17 shows an interface description. An interface contains many methods, which are described by processes. In this example, an interface is described by the process `GetParkingProcess` with Service:describedBy.

```
<service:Service rdf:ID="ParkingSlotService">
   <service:presents
   rdf:resource="&Parking_profile;#Profile_Parkin
   gSlot_Service"/>
   <service:describedBy
   rdf:resource="&Parking_process;#GetParkingProc
   ess"/>
</service:Service>
```

Figure 17. interface description.

2) Function description

Figure 18 shows the structure of a method description. A method is described by AtomicProcess and constants of input and output parameters. The parameter can use existing ontology, e.g., for input `ParkingCity` use Ontology from Ddpeida `db;#City`, or use self-defining parameter `THIS;#ParkingSlot`, which we will show below.

```
<process:AtomicProcess rdf:ID=getParkingProcess">
   <process:hasInput>
        <process:Input rdf:ID="ParkingCity">
       <process:parameterType
       rdf:datatype="&xsd;#string">&db;#City</pro
       cess:parameterType>
       </process:Input>
   </process:hasInput>
   <process:hasOutput>
       <process:Output
       rdf:ID="ExpressParkingSpace">
       <process:parameterType
       rdf:datatype="&xsd;#anyURI">&THIS;#Parking
       Slot</process:parameterType>
       </process:Output>
   </process:hasOutput>
</process:AtomicProcess>
```

Figure 18. Function description.

3) Input and Output Parameter

Figure 19 shows the self-defining parameter description. Self-defining datatype is described by owl-class and relate to the developed Parking Ontology. Linking to existing Ontologies ensures the relations between the elements of different parameters can be also discovered. Each relations for this class should be described by owl:objectproperty or owl:dataproperty and relate to existing ontology on demand.

```
<owl:Class rdf:ID="ParkingSlot">
   <owl:equivalentClass
   rdf:resource="&ps;#ParkingSpace"/>
</owl:Class>

<owl:ObjectProperty rdf:about="parkingLocation">
   <owl:equivalentProperty
   rdf:resource="&ps;#parkingLocation"/>
   <rdfs:range rdf:resource="#ParkingLocation"/>
   <rdfs:domain rdf:resource="#ParkingSlot"/>
</owl:ObjectProperty>

<owl:ObjectProperty
rdf:about="actualOccupancyStatus">
   <owl:equivalentProperty
rdf:resource="&ps;#ActualState"/>
   <rdfs:range
rdf:resource="#ActualStatueOfParkingSpot"/>
   <rdfs:domain rdf:resource="#ParkingSlot"/>
</owl:ObjectProperty>
```

Figure 19. Two example interfaces with annotations.

D. *Java-Annotations for the Interface and Class Description*

Programming interface and semantic interface description should be bilateral transformable. In this approach, Java is used to implement DAiSI framework. We use an aspect oriented method – annotations in Java as a link between the ontology and the actual implementation.

In an interface, every element has at least one label that links it to the ontology. Every label has an attribute `has-Name` to reference the ontology. Ontology names can be found in the application layer. Interface names, for example, need only one label: `@Interfacename`. Functions have three types of labels: `@Activity`, `@OutputParam` and `@InputParameter`. The label for input or output is used only if a function has input– or output parameters. With the help of annotations, the definition of elements of an interface is decoupled from the actual ontology. This measure was taken to ease the changes of either an interface or the ontology, without the necessity to alter both. The code-snippets in Figure 20 present two Java interfaces as examples.

```
@Interfacename(hasName = " ParkingSlotService")
public interface ParkingSpaceInterfacePV{
   @Activity(hasName = "GetParkingProcess")
   @OutputParam(hasName= "ParkingSpace")
   public ParkingSpace []getParkingSlots (
      @Inputparam(hasName = "&db;#City")
         ParkCity:String);
   }
```

Figure 20. Example interface with annotations.

VII. Discovery and Mapping

Discovery and Mapping play an important role in adapter. In this approach, discovery process bases on database, which stores semantic information of services. Mapper uses results from discovery process to create the alignment between required and provided DAiSI services. In this section, we present the details below.

A. *Storage of Semantic Information.*

Semantic descriptions of all interfaces should storage in dynamic adaptive systems. Management of huge information in memory is a big challenge for the device. Therefore, we store semantic information in permanent storage to tackle this issue. Triplestore is a kind of database for storing RDF triples. It can build on relational database or non-relational such as Graph-base databases. Querying of semantic information in these databases is partly supported by the SPARQL.

The ontology layers, which mentioned above, are not forced to be stored in one triplestore. They could be distributed in different databases and expose their ontology through a Web service end-point, typically urls in an ontology, so that increase the reusability of the ontology. SPARQL engine supports partly discovery in using such external urls. In order to reduce management difficulty, we save domain ontology, application ontology and corresponding instance of parameters in a database.

Input and output parameters contain two kinds of information, static value and dynamic value. Static value is value of parameter, which usually save in local database and do not change in the run-time. Accordingly, dynamic value changes at run-time. However, because of huge amount of data, it is difficout even impossible to store all historical datat in database. Therefore, in our appraoch, we store last few historical data.

B. *Discovery*

SPARQL is a set of specifications that define a query language for RDF data, concerning explicit object and property relations, along with protocols for querying, and serialization formats for the results. Reasoner can infer new triples by applying rules on the data, e.g., RDFS reasoner, OWL reasoner, transitive reasoner and general purpose rule engine. By using reasoner more required information can be found, e.g., equivalent classes, classes with parents relation, etc. SPARQL engine can use reasoner in forward chaining, which proceed to add any inferred triples to data set in data store, and backward chaining, which reasoning between a SPARQL endpoint and the data store. Backward chaining is used when ontologies are constantly updated. DAiSI is an adaptive system, components frequently enter and discharge a system, this issue causes regularly addition of new ontologies for service of components in data store. Hereby, change backward chaining is most suitable for DAiSI.

Discovery has two steps, first step is discovery with definition of interface's information, that means only with interface name, input and output parameter name; second step is using static instance information of class to filter results. E.g., application wants to look for services, which could provide parking space in Clausthal in Germany. In the first step, all semantic compatible interfaces, which could provide parking spaces in different locations, are found. Locations of parking spaces are static information which saved mostly in database. Such location information can be used to filter the mount of discovered interfaces to find interfaces which can provide parking spaces in Clausthal. Using static information avoids accessing each interface, so that it avoids the side effect, -component state changed with calling function.

```
select ?interface
where {
?interface <process#hasInput> ?var
?var <process#paramterTyp> "dbpedia#City"

?interface <process#hasOutput> ?var2
?var2 <process#paramterTyp> "this#ParkingSlotApp"
}
```

Figure 21. Example SPARQL query.

Figure 21 shows the SPARQL query example. To find interface we need description required input and output parameters in query. Query could be created directly from semantic noted programming interface.

C. *Mapping*

Discovery result is the interface name of component. In order to create an adapter we need create details relation between required and provide interface. Mapping of each parameter in input and output parameter can be restructured with help of his sematic annotation. According to the results of mapping, an adapter (new DAiSI component) will be created.

VIII. Conclusion

This paper is an extended version of the work published in [1]. In first approach, we presented the enhancement to the DAiSI: A new infrastructure service. Syntactically incompatible services can be connected with the help of generated adapters, which are created by the adapter engine. The adapter engine is prototypically implemented with Java. Re-use of component across different domains is enabled with this approach. In this paper, we extend our previous work by detail of the layered structure of ontologies, an improved discovery process based on SPARQL and triplestore. The new layer structure supports description of instance of parameters and it increases the re-use of ontology. By using triplestore and SPARQL, it facilitates discovery service in a huge number of components. Semantic description hat still strength influence on discovery results. In further steps, we will reduce the closed related relation between semantic description and discovery.

IX. Acknowledgement

This work is supported by BIG IoT (Bridging the Interoperability Gap of the Internet of Things) project funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 688038

References

[1] Y. Wang, D. Herrling, P. Stroganov, and A. Rausch, "Ontology-based Automatic Adaptation of Component Interfaces in Dynamic Adaptive Systems," in Proceeding of ADAPTIVE 2016, The Eighth International conference on Adaptive and Self-Adaptive Systems and Application, 2016, pp. 51-59.

[2] OMG, OMG Unified Modeling Language (OMG UML) Superstructure, Version 2.4.1, Object Management Group Std., August 2011, http://www.omg.org/spec/UML/2.4.1, [Online], retrieved: 06.2015.

[3] H. Klus and A. Rausch, "DAiSI–A Component Model and Decentralized Configuration Mechanism for Dynamic Adaptive Systems," in Proceedings of ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications,Venice, Italy, 2014, pp. 595–608.

[4] H. Klus, "Anwendungsarchitektur-konforme Konfiguration selbstorganisierender Softwaresysteme," (Application architecture conform configuration of self-organizing software-systems), Clausthal-Zellerfeld, Technische Universität Clausthal, Department of Informatics, Dissertation, 2013.

[5] D. Niebuhr, "Dependable Dynamic Adaptive Systems: Approach, Model, and Infrastructure," Clausthal-Zellerfeld, Technische Universität Clausthal, Department of Informatics, Dissertation, 2010.

[6] D. Niebuhr and A. Rausch, "Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems based on Run-time Testing," in Proceedings of the 4th Workshop on Services Integration in Pervasive Environments (SIPE 09) at the International Conference on Pervasive Services 2009, (ICSP 2009), 2009, pp. 7–12.

[7] D. M. Yellin and R. E. Strom, "Protocol Specifications and Component Adaptors," ACM Transactions on Programming Languages and Systems, vol. 19, 1997, pp. 292–333.

[8] C. Canal and G. Salaün, "Adaptation of Asynchronously Communicating Software," in Lecture Notes in Computer Science, vol. 8831, 2014, pp. 437–444.

[9] J. Camara, C. Canal, J. Cubo, and J. Murillo, "An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution," Electronic Notes in Theoretical Computer Science, vol. 189, 2007, pp. 21–34.

[10] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic Synthesis of Behavior Protocols for Composable Web-Services," Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, 2009, pp. 141–150.

[11] A. Bennaceur, C. Chilton, M. Isberner, and B. Jonsson, "Automated Mediator Synthesis: Combining Behavioural and Ontological Reasoning," Software Engineering and Formal Methods, SEFM – 11th International Conference on Software Engineering and Formal Methods, 2013, Madrid, Spain, pp. 274–288.

[12] A. Bennaceur, L. Cavallaro, P. Inverardi, V. Issarny, R. Spalazzese, D. Sykes and M. Tivoli, "Dynamic connector synthesis: revised prototype implementation,", 2012.

[13] OMG, "CORBA Middleware Specifications," Version 3.3, Object Management Group Std., November 2012, http://www.omg.org/spec/#MW, [Online], retrieved: 02.2016.

[14] A. Kalyanpur, D. Jimenez, S. Battle, and J. Padget, "Automatic Mapping of OWL Ontologies into Java," in F. Maurer and G. Ruhe, Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, SEKE'2004, 2004, pp. 98–103.

[15] OMG, OMG Unified Modeling Language (OMG UML) Superstructure, Version 2.4.1, Object Management Group Std., August 2011, http://www.omg.org/spec/UML/2.4.1, [Online], retrieved: 06.2015.

[16] J. Camara, C. Canal, J. Cubo, and J. Murillo, "An Aspect-Oriented Adaptation Framework for Dynamic Component Evolution," Electronic Notes in Theoretical Computer Science, vol. 189, 2007, pp. 21–34.

[17] G. Söldner, "Semantische Adaption von Komponenten," (semantic adaption of components), Dissertation, Friedrich-Alexander-Universität Erlangen-Nürberg, 2012.

[18] D. Martin, M. Bursten, J.Hobbs, et al., "OWL-S: Semantic markup for web services," W3C member submission, 22, 2007-04.

[19] D. Martin, M. Bursten, J.Hobbs, et al., "OWL-S: Semantic markup for web services," W3C member submission, 22, 2007-04.

[20] D. Faria, C. Pesquita, E. Santos, M. Palmonari, F. Cruz, and M. F. Couto, The AgreementMakerLight ontology matching system, in On the Move to Meaningful Internet Systems: OTM 2013 Conferences, Springer Berlin Heidelberg, pp. 527–541.

[21] P. Jain, P. Z. Yeh, K. Verma, R. G. Vasquez, M. Damova, P. Hitzler, and A. P. Sheth, "Contextual ontology alignment of lod with an upper ontology: A case study with proton," in The Semantic Web: Research and Applications, Springer Berlin Heidelberg, 2011, pp. 80–92.

[22] P. Shvaiko and J. Euzenat, "Ontology matching: state of the art and future challenges," IEEE Transactions on Knowledge and Data Engineering, vol. 25(1), 2013, pp. 158–176.

[23] M. K. Bergmann, "50 Ontology Mapping and Alignment Tools," in Adaptive Information, Adaptive Innovation, Adaptive Infrastructure, http://www.mkbergman.com/1769/50-ontology-mapping-and-alignment-tools/, July 2014, [Online], retrieved: 02.2016.

[24] H. Klus, A. Rausch, and D. Herrling, "Component Templates and Service Applications Specifications to Control Dynamic Adaptive System Configuration," in Proceedings of AMBIENT 2015, The Fifth International Conference on Ambient Computing, Applications, Services and Technologies, Nice, France, 2015, pp. 42–51.