Ontology-based Automatic Adaptation of Component Interfaces in Dynamic Adaptive Systems

Yong Wang, Dirk Herrling, Peter Stroganov, and Andreas Rausch
Department of Informatics
Technical University Clausthal
Clausthal-Zellerfeld, Germany
e-mail: yong.wang, dirk.herrling, peter.stroganov, andreas.rausch@tu-clausthal.de

***Abstract*—Dynamic adaptive systems are systems that change their behavior at runtime. Behavioral changes can be caused by user's needs, or based on context information if the system environment changes. The Dynamic Adaptive System Infrastructure (DAiSI) has been developed as a platform for such systems. It is a run-time infrastructure that operates on components that comply to a DAiSI-specific component model. DAiSI-based systems are "open" by design. The run-time infrastructure can integrate components into the system that were not known at design-time. To control the system configuration of such an open and self-organizing system, a configuration service has been developed that can make use of application blueprints to ensure application architecture conformance. Components in a DAiSI system communicate with each other through services. Services are described by domain interfaces, which have to be specified by the component developer. Components can utilize services, provided by other components as long as the respective required and provided interfaces are compatible. However, sometimes services seem to be doing the same thing, e.g., provide the same data or operations, but differ on a syntactical level. Therefore, in this article, we present an approach which enables the use of syntactically incompatible services. We developed an ontology-based method for the generation of an adapter that connects services, which provide the right data in the wrong format. In this paper we present a method to describe interfaces of components and an algorithm to automatically generate adapters between them.**

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 tailored to a very specific application. This effect limits the re-use of existing components in new applications. One measure to minimize redeveloping existing components is to increase reusability. The reuse of existing components is one key aspect in software engineering. 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 dynamic adaptive nature of DAiSI applications, DAiSI components are considered as black boxes. Their capabilities and behavior 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. On a semantical level, the provided service does offer the needed data or operations. To be able to utilize a specific provided service, we suggest to construct an adapter that enables interoperability between services that are only compatible on some semantical level.

The goal of an adapter is to enable communication between two formerly incompatible components. In order to translate different representations of data, a common knowledge-base is needed. In this work we use a central ontology as the common knowledge-base. 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, as the different components that should interact with each other are not known at compile time, but only at integration time, which is the same as run-time in dynamic adaptive systems.

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, how the adaptation of services with the help of an ontology works. In Section VI, the algorithm for the adapter generation is shown in more detail, before the paper is wrapped up by a short conclusion in Section VII.

 ISBN: 978-1-61208-463-3

II. PROBLEM DESCRIPTION

Whenever a dynamic adaptive application is developed, the interfaces between the components are specified at an early stage. They are very domain specific and their definition is driven by the use cases of the future application in mind. On the other hand, many applications run in a shared context with other applications from different domains.

Harmonizing one large interface pool among different developers from different vendors that operate in different domains is a tedious task, which often 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. To tackle these challenges, we propose to keep the domain interfaces un-harmonized. To be able to use a service across domains, we propose to adapt syntactically incompatible services by on-the-fly generated adapters. To be able to do so, we require every interface pool to use an ontology. By either merging these ontologies later on, or by using distributed ontologies we ensure that interfaces from different interface pools share a common semantic.

Components offer provided services. To be able to do so, they may require others and thus, specify required services. Provided and required services stand in relation to each other, mapping which required services are necessary to produce which provided services. In the graphical notation for DAiSI components provided services are marked as filled circles, required services are noted as semi-circles (similar to the UML lollipop notation [13]) and the relation between those two are marked as bars across the component, linking provided and required services (cf. Figure 1).

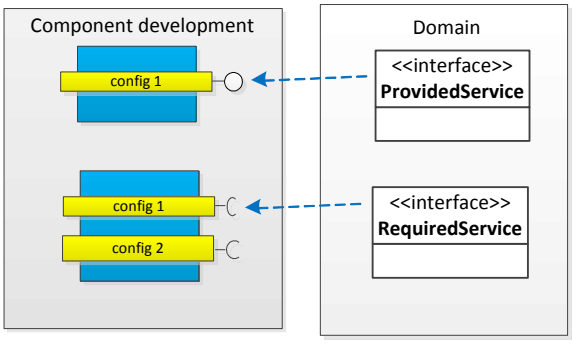

Figure 1. DAiSI components and domain-specific interface definitions.

We propose that services that are semantically compatible, but lack compatibility on a syntactical level, should be usable. We suggest to generate adapters between the different services and define three different adaptation scenarios to face the following three types of incompatibility: Different Naming, Different Data Structure, and Different Control Structure. We believe that we can connect all semantically compatible but syntactically different services using these three types of adapters.

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. The first example, depicted in Figure 2, shows two interfaces: `PowerInfo` and `PowerQuality`. They are named differently, but offer the same functionality. Each of them defines one of the following methods: `update`, and `save` respectively. The names of their parameters are identical and so are the return types.

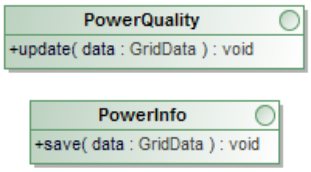

Figure 2. Example of two interfaces with Different Naming.

B. Different Data Structure

In this type of conflict, the names of the interfaces and their functions are the same. However, the parameters differ in their data types. The encapsulated data however is similar and the data structures can be mapped to each other. In Figure 3 in the Different Data Structure example an interface `PowerQuality1` is depicted. It contains a function `saveGridInfo` which processes a parameter of the type `GridData`. In the interface `PowerInfo1`, there are two other functions with the same name but with two different input parameters.

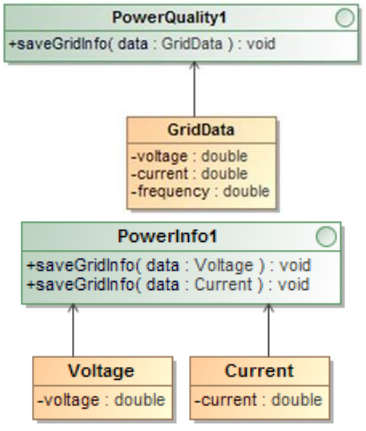

Figure 3. Example of two interfaces with a Different Data Structure.

C. Different Control Structure

In this case, the functions of two different interfaces can

not be mapped directly to each other. To obtain valid results, the control structure has to be modified. In the example in Figure 4, two interfaces `UserInterface` and `UserManager` are given. By definition a username should be composed of the first– and the last name of a person. As such, the two functions `getFirstName` and `getLastName` from the `UserManager` interface in comparison provide the same information as `getUsername` from the `UserInterface` interface.

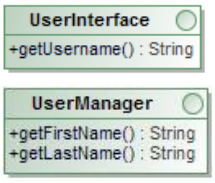

Figure 4. Example of two interface requiring Different Control Structures.

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 [4][7][15][17]. 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 [22]. 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, an 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 [5][20].

Another solution for the connection of semantically incompatible services is presented in [5]. They used buffers for the asynchronous communication between services and translated 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 [18]. Ontologies are used in their method to describe the behavior of components and to create a tool for automated adaptation [8]. 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 [11]. With this tool an ontology can be transformed into the interface definition language (IDL). A. Kalyanpur [21] has developed a method which allows automatic mapping from Web Ontotolgy Language (OWL) to Java. The Object Management Group (OMG) [13] has defined how to transform the Unified Markup Language (UML) into an ontology. With their method, UML classes are first converted into a helper class and then transformed into an ontology [19]. G. Söldner [12] has shown how to transform the UML itself into an 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 decoupled this strong relation. Alternating a part of the ontology now only affects the interfaces directly linked to it.

Matching and merging existing ontologies is still a big challenge regarding its speed and accuracy. To simplify this, many application interfaces (APIs) have been developed, e.g., Agreement Maker [9] and Blooms [14]. Most of them follow a survey approach [10], or use data available on the Internet [6]. 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, ontology merging is simply used in our approach and we did not conduct further research on the challenges mentioned.

IV. The DAiSI Component Model and Infrastructure Services

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 6. The component model is the core of DAiSI and has been covered in much more detail in [2]. The component configurations (yellow bars) are represented by

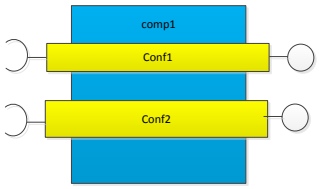

Figure 5. A DAiSI component.

the class of 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 in more detail in [1].

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 [2].

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.

Figure 7 shows the structure of the adapter engine. It computes on the basis of descriptions of services (provided and required) and generates adapter components. The information collector aggregates the information of provided– and candidates for required services (e.g., methods, parameters, and return types). The mapper component compares the gathered information and computes an assignment list, which maps the information from provided services to candidates for required services, the provided service could satisfy. The generator takes the assignment lists from the mapper and spawns new DAiSI components which each could map one provided service to a semantically compatible required service. The adapter engine keeps track of the lifecycle of every DAiSI component. Whenever a DAiSI component leaves the system, the adapter engine destroys all generated adapters and thereby removes them from the system.

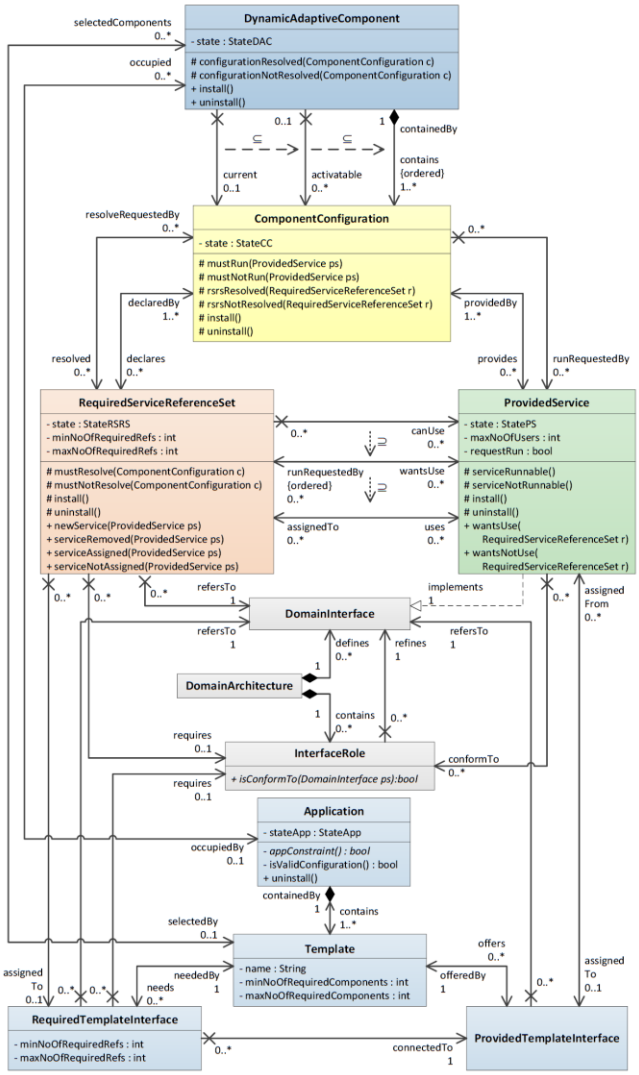

Figure 6. The DAiSI component model.

Figure 8 shows the process and the involved DAiSI components. The component `comp.a` enters the system and provides the service `B`. The adapter engine analyzes the service `B` and, together with its previously built knowledgebase, comes to the conclusion that `comp.b` could use service `B`, but they are syntactically incompatible. The adapter engine can not find another candidate to use service `B`. Thus, it generates only one adapter – a DAiSI component called adapter. It requires the service `B` and provides service `A`. The

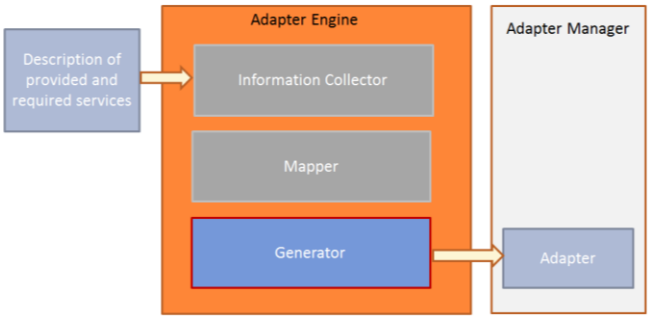

Figure 7. Structure of the adapter engine.

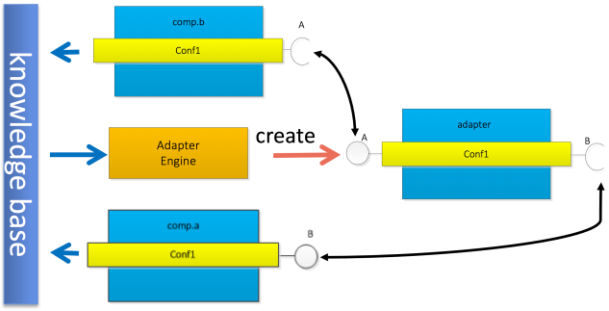

Figure 8. Adaptation process with adapter engine.

DAiSI configuration service binds `comp.b` to adapter and adapter to `comp.a`. The dependency of `comp.b` could be resolved.

V. Interface Description of DAiSI Components with Ontologies

A machine-readable interface description that includes the important semantical information is a key aspect of our concept. Fortunately, making semantic information machine readable, is a well researched and understood field of computer science. For our system, we use a three-layer ontology structure for the construction of the knowledge-base. The upper layer is called UpperOntology layer. In this layer, basic knowledge is defined. Such knowledge can be divided in different upper ontologies. If need be, e.g., if two dynamic adaptive system instances are being merged, the corresponding upper ontologies can be merged. Merging ontologies is a different research area on which we do not focus, however the available results are sufficient for our work.

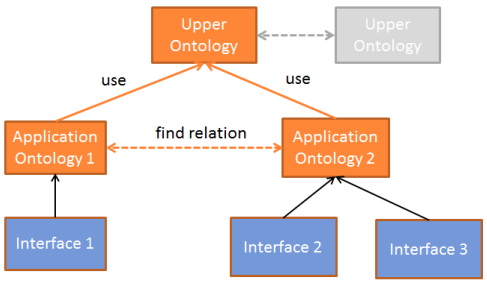

Figure 9. Three-layer ontology structure.

Figure 9 shows the three-layer ontology structure, we used. Every application or domain defines its own upper ontolgy. In the application layer of the ontology, which is the second, or middle layer, all necessary definitions can be found that are relevant for an application. The interface layer of the ontology is the lowest level. It represents the domain interfaces, more precisely their names, methods, parameters and return types. The code of the domain interfaces is directly connected to the ontology. This structure of a three-layer ontology 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.

Figure 10 shows the layout of the ontology for the application example presented in the beginning of this paper. We used two upmost ontologies – Upper Ontology and UML Schema Ontolgy. All definitions and relations for the interfaces, like methods, parameters, or return types can be found in these two ontologies. In the application layer, the ontology data is split by topics. Every information in any of the ontologies can be used in any interface. Datatype classes can also be defined directly in the upper ontology. The following examples show, how the Ontology is defined.

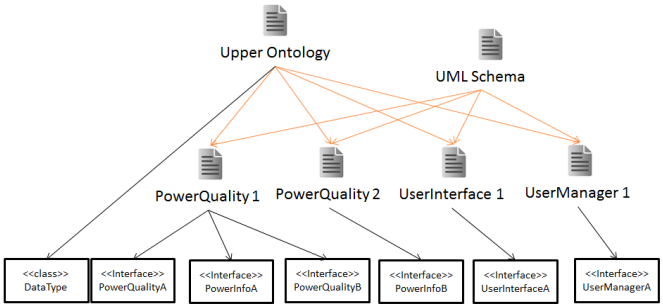

Figure 10. Data structure of the example.

A. Upper Ontology and UML Schema

Figures Figure 11 Figure 12 show graphic representations of the ontologies Upper Ontology and UML Schema.

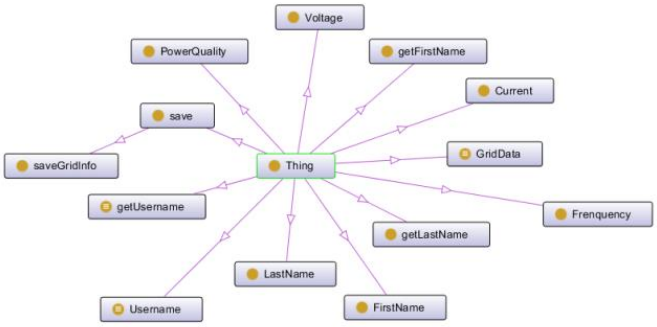

Figure 11. Upper ontology.

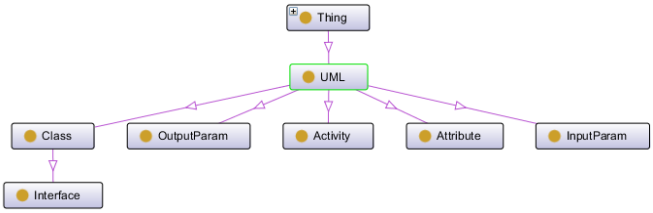

Figure 12. Ontology for UML-Schema.

B. Application Ontology

The ontology file for the interface layer describes all definitions for elements that are used in one application. Every element in an interface, e.g., the interface name, the names of methods, input– and output parameters, are defined as one individual in the ontology. Relations between the elements of different interfaces are also defined in this ontology.

1) Interface name description

Consider the interface `UserManager` which is defined in the ontology of the same name. It is an individual of the upper ontology. Its type is defined as `ont:Interface`, which again is defined in the ontology for the UML-Schema. The code-snippet in Figure 13 shows the definition of the `UserManager` in OWL.

```
<!-- …/ont.owl#UserManager -->

<owl:NamedIndividual
   rdf:about="&ont;UserManager">
   <rdf:type rdf:resource="&ont;Interface"/>
</owl:NamedIndividual>
```

Figure 13. Example of an interface name in the application ontology.

2) Function name description

A function name is also defined as an individual and is of the type `ont:Activity`. To define the relation of the function to the output parameters the individual `ont:hasOutParam` is used. The code-snippet in Figure 14 shows the function `getLastName` as an example; it defines `LastName` as an output.

```
<!-- …/ont.owl#getLastName -->

<owl:NamedIndividual
   rdf:about="&ont;getLastName">
   <rdf:type rdf:resource="&ont;Activity"/>
   <rdf:type rdf:resource="&ont;getLastName"/>
   <ont:hasOuputParam
      rdf:resource="&ont;LastName"/>
</owl:NamedIndividual>
```

Figure 14. Example for the description of a function name in ontology.

3) Input and Output Parameter

Input and output parameters provide important information for the adaptation. Parameter types have to be defined exactly like input and output parameters. The code-snippet in Figure 15 shows the definition for `FirstName`.

```
<!-- …/ont.owl#FirstName -->

<owl:NamedIndividual  rdf:about="&ont;FirstName">
   <rdf:type rdf:resource="&ont;OutputParam"/>
   <rdf:type rdf:resource="&ont;FirstName"/>
</owl:NamedIndividual>
```

Figure 15. Example definition of an output parameter.

C. Java-Annotations for the Interface Description

Our prototype is implemented in Java. 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 `hasName` 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 16 present two Java interfaces as examples.

```
@Interfacename(hasName = "PowerQuality1")
public interface PowerQuality {
   @Activity(hasName = "save")
   public void save(
      @Inputparam(hasName = "GridData")
         GridData griddata);
   }
```

```
@Interfacename(hasName = "UserInterface1")
public interface UserInterface1 {
   @Activity(hasName= "getUsername")
   @OutputParam(hasName= "username")
   public String getUsername();
}
```

Figure 16. Two example interfaces with annotations.

VI. Algorithm for Adapter Generation

In this section we describe the basic concept of the adapter generation in Java, the process for interface comparison, and the inner workings of the generated adapters. Later-on, examples of adapter actions will be shown.

A. Basic principle of the adapter

Every adapter is a DAiSI component, connecting two different interfaces. Figure 17 shows `comp.C` as an example adapter component. The implementation in Java translates a function call from one (`update`) to another (`save`). The provided service of the adapter implements the required interface. The mapping between the required and provided interfaces is implemented in functions of the required interface.

 ISBN: 978-1-61208-463-3

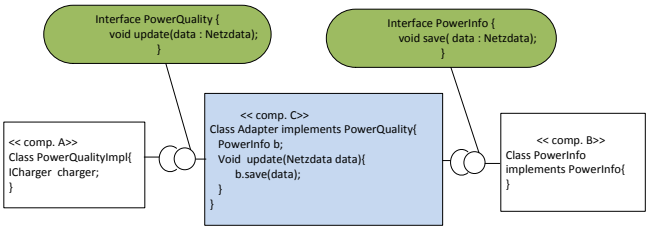

Figure 17. Basic principle of the adapter between two interfaces.

B. *Process for interface comparison*

The information collector in the adapter engine collects instances of the annotations in the Java interface definitions. The mapper uses this collected information to search all dependent instances in the ontology, which are stored in the knowledge-base. The mapper searches for all required interfaces which could possibly use the provided interface. Required services can use provided services, if they are semantically compatible. This path, from required interfaces to provided interfaces is used as a mapping to create the adapter components. Figure 18 shows schematically how the adapter generation works.

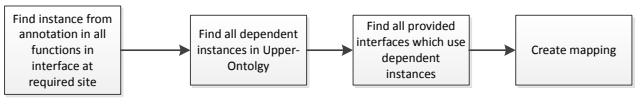

Figure 18. Process for mapping required interfaces to provided interfaces.

C. *Scenario examples*

1) *Different names*

In this case, two functions in required and provided interfaces offer the same functionality, but are named differently. The ontology is used to find the relationship between two functions. The adapter engine generates an adapter component. The required service of the adapter component calls the method of the provided service. Figure 19 shows a UML activity diagram of the behavior of the generated adapter.

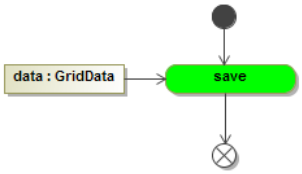

Figure 19. Activity diagram for the call of the `update` method. The call is adapted to the `save`-method.

2) *Different Data Structure*

In the second example, the data structures of parameters are different. For the mapping between parameters, a mapping scheme is searched in the ontology. The adaptation component calls the function from the provided interface using the found mapping for the parameters. Figure 20 shows an UML activity diagram of the behavior of the generated adapter.

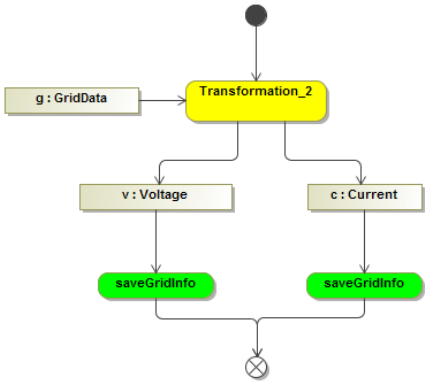

Figure 20. Activity diagram for the adaptation of different data structures

3) *Different Control Structure*

Composition is used, if one function of a required interface can be composed into two or more functions of a provided interface. In this case, the adaptation component calls all functions whose return values can be composed to the required data and composes their return values to match the return value of the adapted interface. Figure 21 shows how two functions are called to account for a difference in control structures.

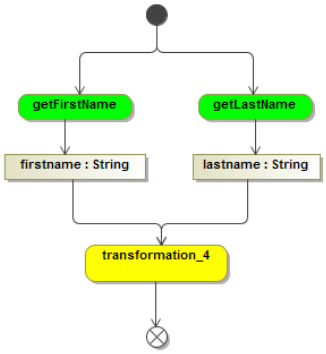

Figure 21. Activity diagram for the adaptation of different control structures

VII. CONCLUSION

In this work, we presented the newest enhancement to the DAiSI: A new infrastructure service. The adapter engine is prototypically implemented with Java and OWL API [3]. It allows the binding of syntactically incompatible services with the help of generated adapters. One of the main benefits is a possible increase of re-use of components across different domains. The layered structure of ontologies allows a collaborative, distributed development.

VIII. FUTURE WORK

In further steps, we will use a distributed ontology, so that every component can be linked directly to the ontology, describing its structure.

 ISBN: 978-1-61208-463-3

REFERENCES

[1] H. Klus, D. Herrling, and A. Rausch, "Interface Roles for Dynamic Adaptive Systems", in Proceeding of ADAPTIVE 2015, The Seventh International conference on Adaptive and Self-Adaptive Systems and Applications, 2015, pp. 80–84.

[2] 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.

[3] The OWL API, https://github.com/owlcs/owlapi/wiki, [Online], December 2015, retrieved: 02.2016.

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

[16] J. Camara, J. Martin, G. Saaün, C. Canal, and E. Pimentel, "Semi-Automatic Specification of Behavioural Service Adaptation Contracts", in Proceedings of the 7th International Workshop on Formal Engineering Approaches to Software Components and Architectures, 2010, pp. 19–34.

[17] 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.

[18] 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.

[19] 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.

[20] H. R. Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casti, "Semi-Automated Adaptation of Service Interactions", Proceedings of the 16th international Conference on World Wide Web, 2007, pp. 993–1002.

[21] 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.

[22] 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.

 ISBN: 978-1-61208-463-3