Semantic Self-Management and Mediation for Home Service Platforms

Jan Schaefer,
RheinMain University of Applied Sciences,
Design Computer Science Media Department, Distributed Systems Lab,
Unter den Eichen 5, D-65195 Wiesbaden, Germany,
http://wwwvs.cs.hs-rm.de, jan.schaefer@hs-rm.de

Abstract:
Service-oriented computing has long outgrown the enterprise. Today, personal living environments feature an increasing number of convenience-, health- and security-related applications provided by distributed interacting devices and services (often called smart homes), which do not only support users but require management tasks such as application installation, configuration and continuous maintenance. These tasks are becoming complex and error-prone. One way to escape this situation is to enable service platforms to configure and manage themselves. The approach presented here extends services with semantic descriptions to enable platform-independent autonomous service level management and mediation using model driven architecture and autonomic computing concepts. It has been implemented as a semantic autonomic manager, whose concept, prototypical implementation and evaluation are presented.

1 Introduction

The *Service-Oriented Architecture* (SOA) principle has spread from global enterprise systems to personalized online services, mobile devices and increasingly smart homes. Services in these areas are usually implemented on top of a middleware called *service platform*. Platforms tailored to the requirements of personal living environments are typically called *home service platforms*.

Constant technological advances and a multitude of vendors with diverse intentions lead to an increasing fragmentation and heterogeneity between these platforms and services severely complicating interoperability and, thus, effective IT management. The situation is even worse in personal living environments as they usually lack a dedicated tech-savvy administrator, who is able or willing to constantly ensure correct operation of smart home services. As a consequence, service platforms should expose autonomic traits such as self-management according to *Service Level Agreements* (SLAs), which they are lacking at the moment.

Services offer public interfaces to enable interaction. These interfaces define a service's functionality syntactically, but they do not describe its semantics. A human service consumer bridges this gap by relying on additional information sources such as documentation or mutual understanding to learn what functionality a service is actually providing. How-

ever, with increasing numbers of services in personal living environments, the manual decision making process has become too complex and, thus, infeasible. Here, semantic web technologies are being discussed as means to provide machine-processable, semantic descriptions of service capabilities to enable automated service mediation. Platforms supporting semantic services are called *semantic service platforms*. Some platforms already use semantics to define context models.

This paper presents a semantic self-management approach for service-based distributed applications. Here, managed services provide semantic descriptions of their functional and management capabilities. *Model Driven Architecture* (MDA) [OMG03] concepts are used to define abstraction levels for platform-independent and platform-specific semantic service and service management models as well as accompanying platform-specific code. Using these abstractions and mappings, concrete instances of formally defined semantic services can be managed according to predefined QoS requirements. In addition, the service instance's QoS properties can be used for dynamic QoS-dependent service binding.

The approach relies on an autonomic manager called *Self-Manager* implementing the MAPE-K control loop — *M*onitor, *A*nalyze, *P*lan, *E*xecute and *K*nowledge — as defined by the *Autonomic Computing Initiative* (ACI) [KC03]. In addition, the self-manager uses semantic models, rules and queries in its knowledge base to manage services on an abstract level: To achieve this, runtime service monitoring data is mapped to high-level models, which are then processed. The derived (counter)actions are executed in managed services via their management agents or interfaces. This decouples the management knowledge and process from the underlying implementation technologies in the same way that semantic service descriptions are defined independently from their respective implementations. Also part of the approach but not presented here due to limited space is a concept for self-healing services, which enables fault-tolerant service execution on distributed computing nodes. As the self-manager itself is service-based, it can be executed redundantly as well, effectively preventing a single point of failure.

The remainder of this paper is structured as follows: Section 2 reviews work related to the approach presented here. Section 3 introduces the concept of the self-management approach for semantic applications, which is followed by a description and evaluation of its prototypical implementation in section 4. The paper ends with a conclusion and a discussion of future work in section 5.

2 Related Work

The *Semantic Markup for Web Services* (OWL-S) [W3C04] was originally intended to be used with Web services as implementation technology, but thanks to its modular design other technologies can be integrated. The SLM ontologies are based on the *UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification* (QFTP) [Obj08], which defines QoS characteristics (properties), QoS constraints and fault tolerance definitions (among others), by the *Object Management Group* (OMG). Apart from the fault-tolerance concepts, the *QoS-MO Ontology for Semantic*

Modeling [TS08] already provides QFTP-derived concepts. As OWL-S and QoS-MO are described sufficiently by respective publications, however, they are not covered here in detail. This section only explains their basic purpose as well as extensions required for the work presented here.

The *OSGi Service Platform* (OSGi) [OSG12a] specification published by the OSGi Alliance provides a component model for the development of modular Java-based applications. Since its initial release, it has been tailored to the needs of embedded and enterprise applications alike. The increasing popularity of OSGi led to broadening tool support and integration into embedded environments such as car entertainment systems as well as enterprise applications.

Even before the arrival of semantic services in particular, several context modeling approaches for service platforms emerged. Gu, Pung, Zhang [G+04] propose using layered ontologies for context models in the home domain. Their work covers concepts that have become widely-acknowledged and, thus, also appear in more recent publications (e.g. activities, locations, persons, devices including custom service concepts). However, the interactions between formal service definitions and runtime systems (the actual service groundings) and how user preferences influence the platform's behavior are not discussed. Diaz Redondo et al. [R+08] propose the combination of OWL-S with an OSGi grounding called *OWL-OS*. Here, the authors use OWL-S service categories as a means to group services into aspects (e.g. lighting), which can be used by clients during service look-up.

With respect to home service platforms, especially *universAAL* [H+11] has to be considered. It combines the results of several European projects aiming to establish a platform in the market. It uses custom ontologies for modeling context and OWL-S for describing services.

Romero et al. [R+11] propose a platform based on the *Service Component Architecture* (SCA), which provides service and component abstractions similar to OSGi. The paper focuses on device integration over heterogeneous communication protocols into an event processing architecture, which is able to process data from a wide collection of devices (including sensor networks). Despite the similarities to the work presented in this paper, however, all these platforms lack SLM capabilities and support for autonomic behavior.

A-OSGi [F+10] implements the MAPE-K control loop on top of OSGi aiming at the management of the OSGi container. The logic of the analysis component, however, is programmed in Java and semantic modeling is not supported.

3 Semantic Self-Management

3.1 Abstraction Layers

Figure 1 presents the three abstraction layers (aligned horizontally) and four functional areas (aligned vertically) that form the foundation of the semantic self-management approach. The first functional area is the *Application Context*, with which semantic services interact by referencing context-provided entities. As the management model is defined

independently from context knowledge, however, the model is not covered in this paper.

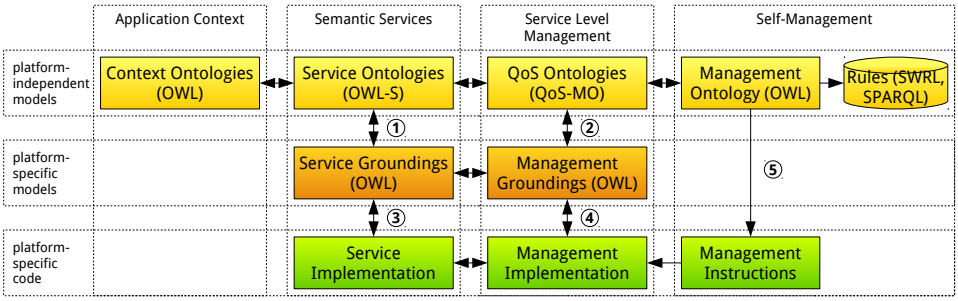

Figure 1: Self-Management Layers

Semantic Services are defined using the OWL-S ontologies. Providing the information required to control service execution, the *Service Level Management* must be integrated with the semantic services model delivering additional management information such as QoS properties. SLM-related properties are defined using the QoS-MO ontology. This modeling concept of this approach has been presented in [S+12] in detail already.

The last functional area, *Self-Management*, relies on a custom management model and accompanying management rules to evaluate formal service definitions against existing implementations at runtime to control service execution. Thus, the accompanying management rules can access both platform-independent and platform-specific models. On the platform-independent ontology layer, the *Quality of Service* (QoS) and *Service* ontologies are defined independently from actual *Service Grounding* and *Management Grounding* implementation technologies. This concept follows the *Separation of Mechanism and Policy* design principle as management queries can be adapted to specific requirements and can be replaced at runtime.

3.2 Grounding Mapping Ontologies

This approach features management logic that is independent from the actual technologies used to implement and manage services. Abstractly defined management decisions are only concretized before execution. Following the MDA paradigm, this separation requires mappings from platform-independent (abstract) to platform-specific (concrete) ontologies. As a result, each to be integrated service implementation technology only requires a new target grounding and accompanying mappings, which abstract from available service and management agent implementation details (e.g. methods and parameters).

To demonstrate the applicability of the selected approach, mapping ontologies from OSGi to OWL-S and *Java Management Extensions* (JMX) to QoS-MO have been developed. OSGi was selected as it is becoming the dominant service implementation technology in smart homes (cf. related work in section 2); JMX was selected as it is the OSGi standard management technology [OSG12b].

Figure 2 (left) shows how OSGi services are mapped to OWL-S entities (transition (1) in Figure 1). In OSGi, services feature an abstraction level themselves: They consist of a service interface (called *Service* in the mapping), whose implementation is provided in a bundle (*ServiceImplementation*). On an abstract level, an OWL-S service has a *Grounding* for each registered OSGi service interface and an *AtomicProcessGrounding* for each actual OSGi service implementation.

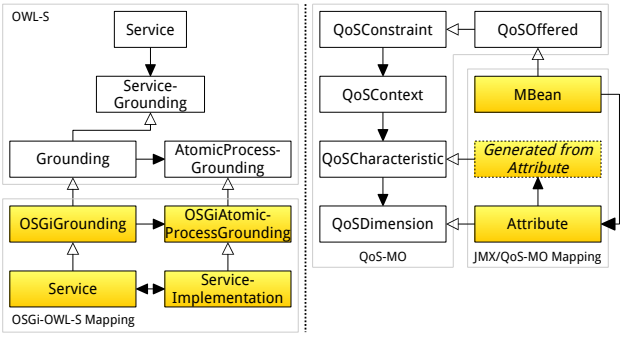

Figure 2: Ontology Groundings

Figure 2 (right) shows how JMX MBeans are mapped to QoS-MO entities (transition (2) in Figure 1). In JMX, managed entities are represented by MBeans, which are realized as Java classes implementing an associated MBean interface. For the approach presented here, services managed through JMX have to implement at least one OSGi service interface and MBean interface each. By convention, MBean attributes should be prefixed with "Current", so that the respective *QoSCharacteristic* sub-concept can be selected during runtime mapping (example: An *Attribute*/*QoSDimension* called *CurrentProcessingTime* results in the selection of the OWL concept *ProcessingTime*). MBeans and their attributes represent *QoSOffered* properties, which can be evaluated against QoS requirements.

3.3 Management Ontology and Rules

The management ontology defines a main concept called *Action*, which is used in the management cycle. Actions describe countermeasures for QoS violations. The ontology is accompanied by two types of management rules: First, there are generic management rules that constantly check predefined QoS requirements against QoS offerings of currently available service implementations. For example, QoS requirements define limits for QoS properties that service implementations have to honor. Depending on the qualification of the constraint (i.e. hard or soft limit), a violation can result in a warning or service fault. Second, service consumers can supply additional QoS requirements that mediated service implementations have to deliver.

3.4 Self-Manager Architecture

The *Self-Manager* (SM), whose architecture is depicted in Figure 3, is a semantic autonomous controller based on the MAPE-K control loop model. The SM consists of five modules, which communicate through events (publish/subscribe model): The *Monitor*, *Analyze*, *Plan* and *Execute* modules provide common functionality for their respective module type. These modules use a shared semantic knowledge base provided by the *Knowledge* module. The SM uses *SPARQL Protocol And RDF Query Language* (SPARQL) [W3C13] management queries representing *Event Condition Action* (ECA) rules to replace administration tasks usually executed by humans. It also extends service platforms that do not necessarily support semantics by providing a *Semantic Service Registry* (SSR), which is used to register services and their *Management Agents* (MA).

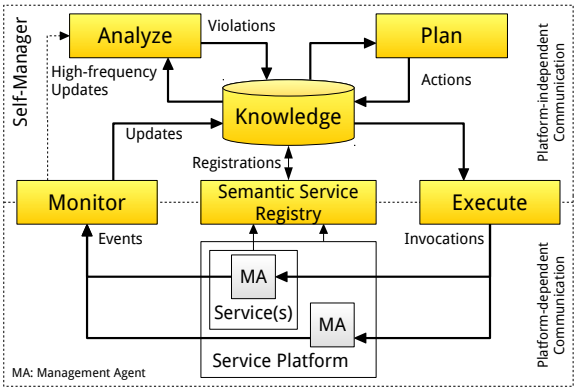

Figure 3: Self-Manager Architecture

Like the ontology layers presented in section 3.1, the SM also follows the MDA paradigm: The monitor and execute modules as well as the SSR map from implementation technologies to platform-specific and platform-independent ontology statements and vice versa (transition ③ in Figure 1). The other modules (analyze, plan and knowledge) only process platform-independent ontology statements resulting in a generic management logic. This allows reusing management rules, even if new service implementation technologies are introduced.

Monitor Modules retrieve platform-specific runtime management information and map it to the ontological representation of the service instance in the knowledge base as QoS offered properties using the corresponding ontology concepts described in section 3.2 (transition ④ in Figure 1). From this moment on, the properties can be used for the management of the service instance or as semantic filter properties by other services during the service selection and binding process (more details on this in the subsequent section 3.5).

Analyze Modules are notified, if QoS properties of monitored services have changed. Whenever an analyze module detects a QoS constraint violation (e.g. exceeded response time limit), it marks the respective QoS offering. Depending on the module implementation, analyze modules might also add facts derived from collected updates to the knowledge

base. The SPARQL query shown in Listing 1 serves as an example, which is used in the prototypical implementation. Here, the current execution time of a service implementation, i.e. of one of its atomic processes, is evaluated against the defined maximum execution time. In case the former is higher than the latter, a QoS violation has been detected.

```
CONSTRUCT {
  ?qosOffered itm:isInViolation true .
}
WHERE {
  ?atomicProcess a proc:AtomicProcess .
  ?qosRequired a qos:QoSRequired .
  ?qosRequired qom:hasTarget ?atomicProcess .
  ?reqExecTime a qos:ExecutionTime .
  ?qosRequired qos:hasContext ?reqExecTime .
  ?maxExecTime a qos:MaxExecutionTime .
  ?reqExecTime qos:hasDimension ?maxExecTime .
  ?maxExecTime qos:hasPrimitiveValue ?maxExecTimeValue .
  ?qosOffered a qos:QoSOffered .
  ?qosOffered qom:hasTarget ?atomicProcess .
  ?offExecTime a qos:ExecutionTime .
  ?qosOffered qos:hasContext ?offExecTime .
  ?offExecTime qos:hasDimension ?qosDimension .
  ?qosDimension a qos:CurrentExecutionTime .
  ?qosDimension qos:hasPrimitiveValue ?currExecTimeValue .
  FILTER( ?currExecTimeValue > ?maxExecTimeValue ) .
}
```

Listing 1: Analyze Query Example

Plan Modules are notified, if violations were added to the knowledge base. These modules are responsible to compute the response to a violation. As they are able to execute queries against the knowledge base, they select a management action based on service-related knowledge and management information provided by the targeted managed service. The response action is stored in the knowledge module and an update event is sent to available execute modules. The SPARQL query shown in Listing 2 is taken from the prototypical implementation: a service restart action is created for each service implementation that exceeds the predefined maximum execution time of its associated semantic service.

```
CONSTRUCT {
  ?qosOffered itm:requiresAction itm:RestartAction .
}
WHERE {
  ?qosOffered a qos:QoSOffered .
  ?qosOffered itm:isInViolation true .
  ?atomProcGrnd a grnd:AtomicProcessGrounding .
  ?qosOffered qom:hasTarget ?atomProcGrnd .
}
```

Listing 2: Plan Query Example

Execute Modules look for management actions they can execute on the managed service implementation respectively its platform. In contrast to analyze or plan modules, however, monitoring and execute modules are platform-specific, as they have to interact with the actual service implementations. The SPARQL query shown in Listing 3 serves as an example, which is used in the prototypical implementation (transition ⑤ in Figure 1). Here, the execute module queries the knowledge base for actions targeted at OSGi services executed in an OSGi framework manageable via JMX. For this, it needs the type of action

to be executed, the bundle ID of the targeted service and the URL of the MBean server. The action can be any action applicable to bundles deployed in an OSGi framework.

```
SELECT
  ?action ?bundleID ?url
WHERE {
  ?qosOffered a qos:QoSOffered .
  ?qosOffered itm:requiresAction ?action .
  ?target a osgi:ServiceImpl .
  ?qosOffered qom:hasTarget ?target .
  ?bundle a osgi:Bundle .
  ?target osgi:hostedByBundle ?bundle .
  ?bundle osgi:hasBundleID ?bundleID .
  ?mbean a jmx:MBean .
  ?mbean jom:manages ?target .
  ?mbs a jmx:MBeanServer .
  ?qosOffered jmx:hostedByServer ?mbs .
  ?mbs jmx:hasURL ?url .
}
```

Listing 3: OSGi/JMX Execute Query Example

If violations without counteractions are detected after the planning phase (i.e. when all plan modules are done processing), the self-manager is unable to handle the violation autonomously . In this case, the issue has to be escalated by specialized execute modules supporting escalation actions (e.g. by sending e-mails to human specialists).

Depending on the underlying service platform, the SSR can retrieve service registrations from existing (non-semantic) service registries to allow mixed usage of semantic and non-semantic services, which means that not all (existing) services have to be outfitted with semantic descriptions.

If a monitored source delivers high frequency updates or if raw data should be preprocessed first (e.g. high-volume sensor data), updates can be made available via a publish/subscribe message channel instead of being added to the knowledge module, which reduces ontology processing effort (i.e. model changes and reasoner updates). Analyze modules can subscribe to this topic allowing for modules working on either ontological knowledge or raw monitoring data (e.g. for *Complex Event Processing* (CEP) applications).

3.5 Manageable Semantic Service Development

The manageable semantic service development process consists of the phases depicted in Figure 4: During *Service Modeling*, the semantic service is defined in OWL-S; QoS properties and constraints, which are required from service implementations, are added during *QoS Modeling* (optional). During *Technology Integration*, the actual service and management agent implementations and the OWL-S and QoS-MO grounding definitions are created, which allows mapping implementations and semantic definitions. This step is required only once for each service/management implementation combination. At *Runtime*, the self-manager's monitor modules create facts reflecting the service's runtime state.

The development process for semantic applications presented here is a hybrid approach. An

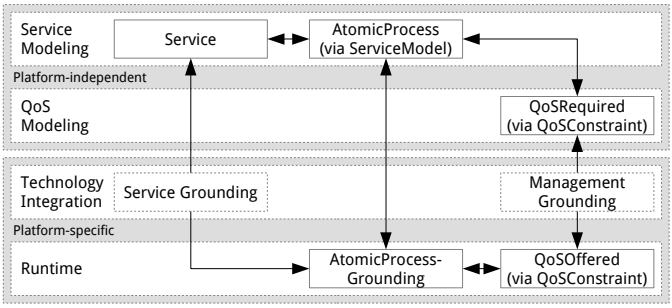

Figure 4: Manageable Semantic Service Development Phases

application can be implemented in a native programming language with platform-specific service interactions, if the application is unaware of semantics. However, to use semantic services (i.e., select services based on QoS properties), the client application has to define a semantic filter that is interpreted by the SSR and can feature requirements for the semantic service and actual instances (sample shown in Listing 4). To provide a semantic service, an application requires an ontological service description defined — if the formalized service or service version is unknown to the registry — and a service grounding describing how the syntactic service interface can be mapped to the abstract description. If both exist, the service can define additional QoS-related properties.

```
# semantic service requirement (predefined)
(MaxExecutionTime<=300)
# service instance requirement (monitored at runtime)
(CurrentExecutionTime<=150)
# Combined expression
(&(MaxExecutionTime<=300)(CurrentExecutionTime<=150))
```

Listing 4: Semantic service selection filter (sample)

Whenever a semantic service registers with the SSR, its provided ontologies are imported into the knowledge base and facts for the new service instance are created. If the service provides a management agent, the service is added to the list of monitored services. The monitoring module either subscribes to or polls for service attribute changes, which are used to update the available ontological facts. In contrast to traditional service property definitions, this approach supports dynamically changing service properties.

4 Prototypical Implementation

This section briefly describes the prototypical implementation of the self-management concept presented in section 3, which uses OSGi and JMX. For ontology processing, the *Apache Jena* framework is used in combination with the *Pellet OWL 2 Reasoner*.

OSGi event listeners are used to keep the models in the knowledge base and the appli-

cation's runtime state synchronized; they intercept bundle start notifications and import provided ontologies. OSGi service find hooks intercept registrations of semantic service implementations; They create OWL representations for them and map these to the associated abstract semantic services.

The JMX grounding is integrated with the QoS-MO ontology and allows representing runtime information provided by MBeans as QoS characteristics. Here, MBean server properties such as thread counts or memory consumption can be used as well as custom MBean attributes, constructors, operations and their respective parameters. Similar to the OSGi grounding, MBean notification listeners are used to synchronize the ontologies in the knowledge module with the MBean server's state. In addition, the prototypical implementation uses listeners to monitor MBean registrations and deregistrations as well as attribute changes and updates the knowledge module accordingly.

Both QoS dimensions defined for the associated semantic service (i.e. requirements) and QoS dimensions monitored at runtime (i.e. offerings) are added to the service properties of OSGi service instances, so that service consumers can use semantic filter expressions as shown in Listing 4 to express their requirements for mediated services. This allows defining service dependencies in a way that the standard OSGi service registry can bind and rebind services and consumers automatically depending on QoS properties and requirements.

A first evaluation of the prototype focused on the self-manager's module interaction and update processing capabilities. It featured a simple, arbitrary semantic test service, whose execution time was artificially increased with each processed request by 50ms starting at 100ms. The execution time was exposed via JMX and semantically represented as a QoS offering. A single consumer invoked the service continuously without additional delays between requests.

To demonstrate the self-manager's semantic management capabilities, the QoS requirement of a maximum execution time of 500ms was defined for service groundings implementing the associated semantic service. Once an exceeding execution time monitored by the JMX monitoring module was detected by the analyze module, the latter created a QoS violation alert, which resulted in the selection of a counteraction by the plan module (here: service grounding restart) and executed by the JMX execute module. After the restart, invocation times once again increased with each request. Table 1 presents the measurement results for 26853 service invocations recorded on a Core i5 (all times in milliseconds).

The difference between JMX monitor module updates and analyze module activity (26853 vs. 24803) was caused by management cycle activity: If an update arrives during cycle activity, the update is stored until the next cycle. However, only the latest update is buffered. If the self-manager received updates without QoS violations, the cycle ended after the analysis taking 30.3ms on average. If a violation was found and an action had to be taken, the cycle took 67.2ms on average (from MBean notification to finished OSGi service restart). Using the *Oracle VisualVM* diagnostics tool, it was verified that large parts of the management cycle execution time is spent on implicit ontology consistency checks that take place before each SPARQL query, which delays each query execution.

Service mediation, on the other hand, was not affected by ontology handling. There was a minimal delay updating OSGi service properties (<1 ms), whenever a service implementa-

Tag	Average	Min	Max	Std.Dev.	Count
JMX Monitor Module (Update)	0.2	0	56	0.9	26853
Inference Model Synchronization	5.1	4	85	1.2	24803
Analyze Module	23.7	21	199	3.0	24803
Plan Module	2.0	1	15	1.8	2502
JMX Execute Module (Query)	5.2	4	93	2.9	2502
JMX Execute Module (Action)	27.3	24	76	2.9	2502
Management Cycle (No Violation)	30.3	26	229	3.4	22301
Management Cycle (Execution Complete)	67.2	60	151	5.6	2502

Table 1: Execution Times

tion's QoS properties changed

5 Conclusion and Future Work

The semantic self-management concept and the accompanying autonomic manager prototype presented in this paper show that the integration of Semantic Web, Model Driven Architecture and Autonomic Computing concepts is viable. The separation of service modeling and implementation aspects allows integrating new service or management agent implementation technologies without modifications to the descriptions of the semantic services or existing management rules.

The initial evaluation of the prototype showed, however, that if real-world state changes are committed to the knowledge base immediately (without pre-processing using, for example, a CEP engine as described in section 3), model updates and synchronization delays caused by reasoner updates result in management cycle execution times, which might not be suitable for time-critical applications. For personal living environments with a limited amount of participating nodes, however, they might be acceptable.

For semantic service mediation, the prototype assumes that the services are implemented as OSGi services as methods, parameters and data types are not matched semantically during service look-up (i.e. the service name must be an OSGi interface name). However, semantic QoS properties are used as OSGi service properties (and continuously updated), which allows using them as service look-up filters and to control mediation between consumers and services through standard OSGi mechanisms (e.g. service rebinding in case of performance degradation). Since OSGi is becoming the dominating service platform technology in personal living environments, however, this does not restrict its applicability.

Although this semantic self-management approach focuses on home service platforms, it is not limited to this application area but can be used in the enterprise as well. In addition, the importance of autonomic properties and semantic web technologies is constantly increasing in especially but not limited to the areas of cyber-physical systems and machine-to-machine communications (e.g. [ETS12]), which opens up new possibilities for future applications.

The prototype has been evaluated in the WieDAS research project funded by the *German Federal Ministry for Education and Research* (BMBF) under contract number 17040B10. Here, the testing focus was on the general real-world applicability of the approach in personal living environments.

References

[ETS12] ETSI Technical Committee M2M Communications. Machine to Machine Communications (M2M); Study on Semantic support for M2M Data, December 2012.

[F+10] João Ferreira et al. A-OSGi: A Framework to Support the Construction of Autonomic OSGi-Based Applications. In Athanasius V. Vasilakos, Roberto Beraldi, Roy Friedman, et al., editors, *Autonomic Computing and Communications Systems*, volume 23, pages 1–16. Springer Berlin Heidelberg, 2010.

[G+04] T. Gu et al. Toward an OSGi-based Infrastructure for Context-Aware Applications. *Pervasive Computing, IEEE*, 3(4):66 – 74, 2004.

[H+11] Sten Hanke et al. universAAL – An Open and Consolidated AAL Platform. In Reiner Wichert and Birgid Eberhardt, editors, *Ambient Assisted Living*, pages 127–140. Springer Berlin Heidelberg, Berlin, Germany, 4. AAL-Kongress edition, 2011.

[KC03] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, January 2003.

[Obj08] Object Management Group. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification (Version 1.1), April 2008.

[OMG03] OMG Model Driven Architecture Working Group. MDA Guide Version 1.0.1, 2003.

[OSG12a] OSGi Alliance. OSGi Service Platform v5.0 Core Specification, March 2012.

[OSG12b] OSGi Alliance. OSGi Service Platform v5.0 Enterprise Specification, March 2012.

[R+08] R.P. Diaz Redondo et al. Enhancing Residential Gateways: A Semantic OSGi Platform. *IEEE Intelligent Systems*, pages 32 –40, Jan.-Feb. 2008.

[R+11] Daniel Romero et al. The DigiHome Service-Oriented Platform. *Software: Practice and Experience*, 2011.

[S+12] Jan Schaefer et al. QoS-based Testing and Selection of Semantic Services. In *ARCS 2012 Workshops (ASPRIT)*, Lecture Notes in Informatics (LNI), pages 15–26. Gesellschaft für Informatik e.V. (GI), Koellen Druck+Verlag GmbH, Bonn, February 2012.

[TS08] Gustavo Fortes Tondello and Frank Siqueira. The QoS-MO Ontology for Semantic QoS Modeling. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 2336–2340, New York, NY, USA, 2008. ACM.

[W3C04] W3C. OWL-S: Semantic Markup for Web Services (W3C Sub.), November 2004.

[W3C13] W3C. SPARQL 1.1 Query Language (W3C Recommendation), March 2013.