Semantic Web ontologies for Ambient Intelligence

Runtime Monitoring of Semantic Component Constraints

Klaus Marius Hansen and Weishan Zhang and Joao Fernandes and Mads Ingstrup
Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
{klaus.m.hansen,zhangws,jfmf,ingstrup}@daimi.au.dk

ABSTRACT

Semantic web-based context modeling is widely used in pervasive computing systems to achieve context awareness which is essential for Ambient Intelligence (AmI). In the Hydra middleware for pervasive services, context awareness is extended for self-management purposes, which is an integral part of Hydra. To achieve this, a set of self-management ontologies called *SeMaPS (Self-Management for Pervasive Services)* are developed, where the dynamism of device state changes and service invocation are taken into account. To show the effectiveness of these ontologies, in this paper, we focus on our Component ontology that extends OSGi's Declarative Service specification to add capabilities for expressing architectural (especially global) and functional constraints (e.g. contextual constraints), and show how to use it to verify component configurations by using a pervasive service compiler at runtime.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Validation*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures*; I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalisms and Methods—*Representations (procedural and rule-based)*

General Terms

Languages, Design, Verification.

Keywords

OWL/SWRL ontology, context awareness, ambient intelligence, Configurations, OSGi declarative service

1. INTRODUCTION

Ambient Intelligence (AmI) aims to make pervasive computing[10] more usable through natural interaction, personalized and efficient services, and context awareness. Having context awareness is very important in knowing when and where a service can happen, what triggers a service provision, how to provide a service to whom and so on. Thus context awareness is essential to achieving AmI and decisions and services should be based on current or historical contexts. To enable AmI in the Hydra project (IST-2005-034891), we are building self-management capabilities into the Hydra middleware, supported by context ontologies based on semantic web technologies (OWL[1] and SWRL (Semantic Web Rule Language[2])).

Semantic web-based context modeling is arguably a powerful approach for context modeling [11], since it can provide reasoning potentials for contexts, a capability not easily achievable by other context modeling approaches. In order to support self-management, the context models should have dynamic and runtime information of the system available in order to take appropriate actions based on these dynamic contexts. Hence, the dynamic information should be reflected in the context models and used for self-management in order to make decisions on what actions should be taken to react to the changes that are needed for self awareness.

Existing pervasive computing context ontologies, such as SOUPA[1] and Amigo[4], are not targeting self-management and almost no dynamic and runtime models of the underlying pervasive systems are considered. This makes these existing ontologies unsuitable for the self-management purposes, which depend on the timely reporting of the status of devices, network, and even on running processes.

To realize our vision of semantic context awareness-based self-management in Hydra [14, 16], we designed *SeMaPS* (Self-Management for Pervasive Services), a set of self-management ontologies in Hydra. The context ontologies in SeMaPS are considering runtime contexts which are necessary for self-management. These dynamic contexts include device runtime status, service call/response relationships, and service execution time. SWRL rules are developed to handle self-management features such as malfunction diagnosis, device and system status monitoring, and service selection based on QoS parameters. When there are state changes or service calls, the dynamic running information is fed into the related self-management context ontologies, which then trigger execution of self-management rules for adaption, monitoring, diagnosis, and other aspects of self-management.

For this paper, we will demonstrate the utility of SeMaPS

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

[1]OWL homepage. http://www.w3.org/2004/OWL/
[2]SWRL specification homepage. http://www.w3.org/Submission/SWRL/

by focusing on one of its ontologies, Component ontology that extends OSGi's Declarative Service specification through *OWL-DS* to add capabilities for expressing architectural and functional constraints over component configurations, and show how to use it to verify the component configurations at runtime by using a pervasive service compiler (*Limbo* [3]) as an example.

The rest of the paper is structured as follows: The design of SeMaPS context ontologies and their structure are presented in Section 2. Next, we discuss how we extended OSGi DS (Section 4) followed by a more detailed case study of applying OWL-DS to the Limbo compiler (Section 5). We present the implementation of runtime validation in Section 6. Then we discuss related work in Section 7. The paper is concluded in Section 8.

2. SEMAPS ONTOLOGIES

In Hydra, to make full use of context awareness, a semantic web based self-management approach is being adopted [14, 16], with the support of the SeMaPS context ontologies. The high level structure SeMaPS ontologies is shown in Figure 1. The dynamic contexts are modeled with runtime concepts and properties in the related ontologies.

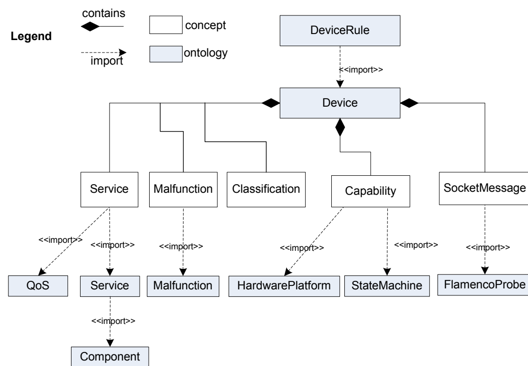

Figure 1: Structure of the SeMaPS ontologies

The Device ontology presents *HydraDevice* (as a concept) type classification (e.g. Mobile Phone, PDA, Thermometer). This is based mainly on the device classification in Amigo project ontologies [4]. To facilitate self-diagnosis, there is a concept called *HydraSystem* to model a system composed of devices that provide services. A corresponding object property *hasDevice* is added, which has the domain of *HydraSystem* and range as *HydraDevice*. The *HydraDevice* concept has a data-type property *currentMalfunction* which is used to store the inferred device malfunction diagnosis information at runtime.

The HardwarePlatform ontology defines concepts such as *CPU*, *Memory*, and relationships to devices (in the Device ontology), for example *hasCPU*. This ontology is based on the hardware description part from W3C's deliveryContext ontology[3]. Power consumption concepts and properties for different wireless network are added to the HardwarePlatform ontology to facilitate power-awareness, including a *batterLevel* property for monitoring battery consumption at runtime.

[3]Delivery Context Overview for Device Independence. http://www.w3.org/TR/di-dco/

The device Malfunction ontology is used to model knowledge of malfunction and recovery resolutions. It provides the classification of device malfunctions (for example, *BatteryError*). The malfunctions are defined into two categories: *Error* (including device totally down) and *Warning* (including function scale-down, and plain warning) according to severeness. There are also two other concepts, *Cause* and *Remedy*, which are used to describe the origin of a malfunction and its resolution.

The QoS ontology defines some important QoS parameters, such as availability, reliability, latency, error rate, etc. Furthermore, properties for these parameters are defined, such as their nature (dynamic, static) and impact factor. There is also a *Relationship* concept in order to model the relationships between these parameters. The QoS ontology is developed based on Amigo QoS ontology [4].

To model device state changes, a state machine ontology is developed based on [2] with many improvements to facilitate self-management work: the State concept has data-type property *isCurrent* to indicate whether a state is current or not for the purpose of device monitoring, a *doActivity* object property is added to the *State* in order to specify the corresponding activity in a state, and also a data-type property *hasResult* is added to the *Action* (including activity) concept in order to check the execution result at runtime, together with three data-type properties that are added to model historical action results in order to conduct history based self-management work.

To model the invocation of services, a FlamencoProbe ontology is developed to monitor the liveness of computing node, and facilitating the monitoring of QoS, such as the request/response time of a corresponding service call. The *SocketProcess* concept is used to model a process running in a client or service, and *SocketMessage* to model a message sent to/from between client and service. There is also a concept called *IPAddress*, which is related to *HydraDevice* with a property *hasIPAddress* in the Device ontology. The object properties *invoke*, *messageSourceIP*, and *messageTargetIP* are used to build the invoking relationships, and data type property *initiatingTime* is used to model the time stamp for a message.

The Component ontology is based on the OSGi's Declarative Service specification [5] as we are adopting OSGi as the underlying component model in Hydra. It specifies the *Component* (as a concept) dynamic status, for example whether it is *enabled*, and also static characteristics such as its *reference* to other service, its *implementation* interface, and *services* provided. Figure 2 shows partially the details of the Component ontology.

In the following sections, we will demonstrate the capabilities of these ontologies from a different angle by applying the Component ontology and SWRL rules to verify configurations of Limbo. We will first introduce OSGi's Declarative Services (OSGi DS) on which we build the Component ontology, and then we discuss OSGi DS' shortcomings, exemplified with requirements for Limbo configurations. Then we propose an approach called *OWL-DS* based on the Component ontology and SWRL rules, to verify the configurations for Limbo at runtime. Please note that the OWL-DS approach is not limited to Limbo, but can be applied to all situations where configuration need to be verified using the OSGi component model.

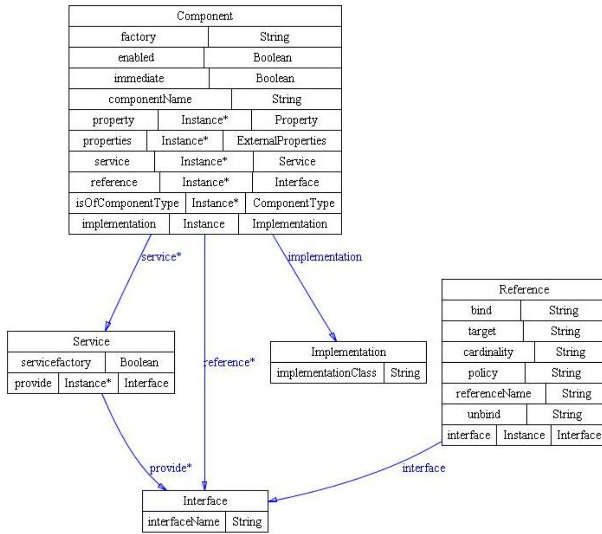

Figure 2: Component ontology (simplified)

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="com.eu.hydra.limbo">
  <implementation class="com.eu.hydra.limbo.Limbo"/>
  <service>
    <provide interface="com.eu.hydra.limbo.generator.Generator"/>
  </service>
  <reference name="BACKEND"
     interface="com.eu.hydra.limbo.backend.Backend"
     cardinality="1..n"
     policy="dynamic"
     bind="addBackend"
     unbind="removeBackend"/>
  <reference name="FRONTEND"
     interface="com.eu.hydra.limbo.frontend.Frontend"
     cardinality="1..n"
     .../>
  <reference name="REPOSITORY"
     interface="com.eu.hydra.limbo.repository"
     .../>
</component>
```

Figure 3: OSGi DS Example

3. OSGI DECLARATIVE SERVICES

OSGi provides a set of services per default [5], one of which is *Declarative Services* management. OSGi's Declarative Services Specification [5] enables developers on the OSGi platform to declaratively manage service composition at runtime. Concretely, OSGi DS allows OSGi bundle developers to provide a XML-based description of *components* that may be instantiated at runtime to provide and require services. Figure 3 shows an example of such a description which specifies the main component of the Limbo compiler.

Figure 4 shows a logical view of (a part of) Limbo's software architecture. *Limbo* provides a *Generator* service that *Frontends* and *Backends* may use. Frontends process source artifacts (such as Web Service Description Language (WSDL[4] files) whereas Backends produce output artefacts (such as web service stubs and skeletons). Both Backends and Frontends may use a single *Repository*. At runtime Limbo selects and uses a set of Backends (and Frontends) based on Limbo's

[4]`http://http://www.w3.org/TR/wsdl`

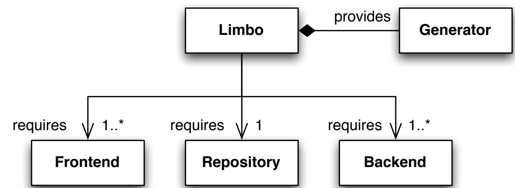

Figure 4: Limbo Logical Architecture

configuration.

This description essentially makes sure that the runtime architecture of Limbo is as shown in Figure 4: The Limbo component (inside an OSGi bundle) will be instantiated by the OSGi DS runtime and that, in this case, will need to implement the *Generator* interface since this is a service provided by the component. Furthermore, the Limbo component requires the presence of at least one *Frontend* and at least one *Backend* and one *Repository*. When these services are available, the references are said to be *satisfied* and the component may be *activated*.

Essentially, OSGi DS provides a way for components to specify provided and required services (in the form of Java interfaces) declaratively so that the OSGi framework can resolve service dependencies dynamically. Even though this is a convenient and powerful composition mechanism, we argue that it should be extended. The Limbo compiler case exemplifies a number of limitations of OSGi DS:

- *Global constraints are not supported.* This means that one cannot express architectural constraints that are non-local. An example of this could be that there must be exactly one instance of the *Repository* service for consistency reasons.
- *Contextual constraints are not supported.* The OSGi DS constraints are closed in the sense that they are specified at packaging time in the OSGi bundles. An example of where this is insufficient in the Limbo case is that it could not express that JME and OSGi server is not a legal combination.

4. EXTENDING OSGI DS THROUGH OWL-DS

We use an approach as in SAWSDL[5] to extend OSGi DS. In doing so, OSGi components reference the Component ontology and configuration rules. The Component ontology is built based on OSGi DS XML Schema as shown in Figure 2. This approach to extending OSGi DS is called OWL-DS.

To conveniently model component types, we add a concept *ComponentType* into the OWL model. This is important for specifying the constraints based on the component types. In fact, the services provided by a component (specified by interfaces) can be used to identify a component type, but this may be counter-intuitive for a user to reference the component type in this way. We are also using SWRL rules to link the *ComponentType* with the component interfaces. In order to model the configuration based on the semantic components, we use a *SystemConfiguration* concept to model the set of configurations that components can have.

[5]`http://www.w3.org/TR/sawsdl/`

There is a way to specify some local constraints using OWL capabilities (using cardinality restriction). OWL cardinality restrictions are referred to as local restrictions as they are stated on properties with respect to a particular class. That is, the restrictions constrain the cardinality of that property on instances of that class[6]. This is a simple and feasible way to implement constraints for component configurations. However, this approach is not sufficient to specify component configurations for our purposes for the following reasons:

Low usability : The model developer has to specify all the components as concepts (for every component) and properties explicitly, and then use the concepts and properties to specify constraints. This is not a practical way as it is hard to enumerate components in practice.

Not flexible : All constraints are explicitly stated with every component, if a new component is added, the model has to be changed accordingly. There should be a scalable way to specify constraints.

Not powerful : OWL in itself is criticized for its limited expressiveness, e.g., not being able to use math expressions or string operators that may be involved in constraints.

We are intending to use SWRL to specify the constraints for the configuration of components. This is feasible with SWRL (e.g. using the SWRL APIs from Protege[7]) because of the extensibility of SWRL.

If we know the current set of components that are available to OSGi DS, we can apply SWRL to a semantic description of this set. Here we can use SWRL built-ins, such as the mathematical built-ins, string built-ins, and Abox and Tbox built-ins. The basic idea is to retrieve these current components, and then use SWRL to specify what is a valid configuration, and what is an invalid combination following by reporting of violations of configurations.

5. LIMBO RUNTIME VALIDATION

In this section, we will use SWRL to specify the validation of component configuration. A SWRL rule is composed of an antecedent part (body), and a consequent part (head). Both the body and head consist of positive conjunctions of atoms. A SWRL rule means that if all the atoms in the antecedent (body) are true, then the consequent (head) must also be true. SWRL is built on OWL DL and shares its formal semantics. In our practice, all variables in SWRL rules bind only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. In our example SWRL rules, the symbol $\wedge$ means conjunction, and $?x$ stands for a variable, $\rightarrow$ means implication, and if there is no ? in the variable, then it is an instance.

Our approach is general and not limited to Limbo, the component model and configuration model are generic and can be applied to any cases of component semantic descriptions and component semantic configurations.

Taking Limbo as a case, the following steps are involved in the semantic validation. In practice, all the steps can be executed in a whole instead of step by step.

[6]http://www.w3.org/TR/owl-features/

[7]http://protege.stanford.edu/

Check the services required by a component. The rule retrieves all components in the current configuration. If a component has a reference which has cardinality at least one, then there must be component provide the required service. Or else, there is something wrong with the configuration. This step is not necessary if OSGi DS is used, but it is necessary if the OSGi component model applied to other situations. The rule is shown in Figure 5.

```
ComponentBased(?con) ∧
hasComponent(?con, ?comp1) ∧
osgicomponent:reference(?comp1, ?ref1) ∧
osgicomponent:cardinality(?ref1, ?car1) ∧
swrlb:containsIgnoreCase(?car1, "1.") ∧
osgicomponent:interface(?ref1, ?inter1) ∧
osgicomponent:interfaceName(?inter1, ?name1) ∧
hasComponent(?con, ?comp2) ∧
osgicomponent:service(?comp2, ?ser2) ∧
osgicomponent:provide(?ser2, ?inter2) ∧
osgicomponent:interfaceName(?inter2, ?name2) ∧
swrlb:equal(?name1, ?name2)
  → sqwrl:selectDistinct(?con, ?comp1, ?comp2, ?name1, ?name2) ∧
sqwrl:select("valid references for Configuration")
```

Figure 5: rule for checking component reference

Check component platform. All components should support the required targetted platform for Limbo compilation. A component should have this supported platform specified in its property and will be retrieved by the rule and compare with the specified targeting platform, if it is not supported, then it is not valid for this configuration. The rule is shown in Figure 6.

```
ComponentBased(?con) ∧
hasComponent(?con, ?comp1) ∧
osgicomponent:property(?comp1, ?pro1) ∧
osgicomponent:propertyName(?pro1, ?proname1) ∧
osgicomponent:value(?pro1, ?value) ∧
targetingPlatform(?con, ?plat) ∧
swrlb:notEqual(?value, ?plat)
  → sqwrl:selectDistinct(?con, ?comp1, ?pro1, ?value, ?plat) ∧
sqwrl:selectDistinct("invalid plaform combination for targeted and component supported")
```

Figure 6: rule for checking component supporting platform and the targeted platform

Check generation combination. Some of the generation combinations are not meaningful. For example, if *JME* is a targeting platform, then an *OSGi* server is not an option because OSGi is not supported on JME currently in our environment. This rule is shown in Figure 7.

Check number of component type limitations. Sometimes it is important to limit the number of component with a specific type, in a running configuration. We will first retrieve the component and assert its type according to its provided interfaces, and then count the number of this kind of components, and programatically check with its limit. This rule is shown in Figure 8.

In order to specify the limitation of different type of components a configuration has in the rule body, the SWRL builtins should be extended, and will be out of scope of

```
CurrentConfiguration(?con) ∧
currentBackend(?con, ?currentb) ∧
osgicomponent:hasComponentInstance(?currentb, ?inst) ∧
abox:hasURI(?inst, ?uri) ∧
swrlb:containsIgnoreCase(?uri, "JME") ∧
CurrentConfiguration(?con1) ∧
currentBackend(?con1, ?currentb1) ∧
osgicomponent:hasComponentInstance(?currentb1, ?inst1) ∧
abox:hasURI(?inst1, ?uri1) ∧
swrlb:containsIgnoreCase(?uri1, "OSGi")
  → sqwrl:select(?con, ?inst) ∧
sqwrl:select(?con1, ?inst1) ∧
sqwrl:select("invalid JME + OSGi server Configuration")
```

Figure 7: rule for checking OSGi server on JME platform is invalid

```
CurrentConfiguration(?con) ∧
hasComponent(?con, ?comp) ∧
osgicomponent:service(?comp, ?service) ∧
osgicomponent:provide(?service, ?interface) ∧
abox:hasURI(?interface, ?uri) ∧
swrlb:containsIgnoreCase(?uri, "Repository")
  → sqwrl:select(?comp) ∧
sqwrl:count(?comp) ∧
osgicomponent:LimboRepository(?comp) ∧
sqwrl:countDistinct(?comp)
```

Figure 8: rule for counting Limbo repository and assert a component as a repository component

open world assumption of OWL/SWRL. Therefore programatically controlling this limit is a good option for flexibility and soundness. This later approach is the one we are using.

The component reference rule (Figure 5) is a generic rule that can be applied to all other situations where the declarative service model is used. The second rule for platform checking (Figure 6) is also generic, in situations where a component's supported platform needs to be checked. The checking of components' type limits (Figure 8), can be generalized through parameterization of SWRL that will be coming later when this feature is available from SWRL APIs. The only rule that is very Limbo specific is the rule for checking the generation combination. All in all, this semantic validation algorithm can be applied to multiple situations.

6. DESIGN AND IMPLEMENTATION

We have implemented the usage of the Component ontology and Limbo runtime validation using Protege-OWL APIs, based on the ideas presented in the former sections. Specifically, we have (1) Designed and realized the Component ontology based on Declarative Services using the Protege ontology editor, (2) Extended Eclipse Equinox's[8] Declarative Services implementation to discover and maintain a model of the component instance topology, and (3) Run time monitoring and validating of Limbo components combination using Protege-OWL/SWRL APIs.

A static Component & Connector view of the implementation is shown in Figure 9

The Equinox DS bundle has been extended with a Binding Listener that knows when service are bound to (and unbound from) components. Whenever such an event happens, the Binding Listener uses the standard OSGi Event Admin that provides a topic-based publish/subscribe service. This enables the OWL-DS Monitor to maintain a model of component instances, their services, and their relationships.

[8]http://www.eclipse.org/equinox/

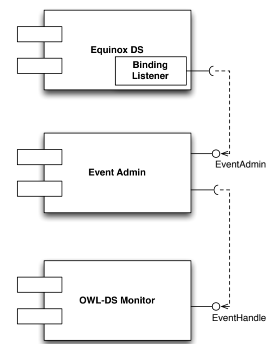

Figure 9: Static Component & Connector View of Current OWL-DS Implementation

Based on this information, the OWL-DS Monitor validates component configurations as they are made by the Equinox DS implementation, notifying whether they are valid or invalid according to semantic constraints.

7. RELATED WORK

None of the existing ontologies for pervasive computing, such as SOUPA and Amigo[4], are considering self-management concepts and requirements, however. The SeMaPS presented self-management ontologies in this paper are the key to enable various self-management tasks. At the same time, we can model complex contexts using SWRL with the Hydra ontologies [16]. Work in [7] applied SWRL-based context modeling, and illustrated three cases of applying SWRL. We go beyond it by the dynamic state-based monitoring and diagnosis using context ontologies.

Using semantic techniques (such as OWL) to extend existing component models has partially been attempted before. Sillitti and Succi[9] use XML schema to specify facets of components but they do not provide details on how to configure components. Furthermore, the implementation of the approach uses ontologies, but it is not detailed how.

In the area of product lines and feature configuration, Wang et al. [13] present a feature modelling approach in which legal feature combinations in which configuration rules are expressed with OWL. We explored this approach in the first versions of Limbo, but the approach cannot adequately specify global constraints due to the limited expressiveness of OWL. The approach of Wang et al. is furthermore mainly a design time method and is not able to cope with new contexts at runtime. The SWRL based configuration constraints that we propose could provide a more flexible solution.

An architecture description language can be used to describe configurations and styles, and then map a formally rigorous ontological specification of styles to that ADL such that its semantics are well defined. Pahl et al.[6] describe an ontology-based modeling framework for architectural styles, and show how it can be used in connection with the ACME ADL. However this work is theoretical and supports rather than overlaps with our own, in that it makes precise the benefits to an ontology based model compared to one based purely on an ADL.

Redondo et al. [8] present work to enhance the semantics of OSGi services (rather than components) to support en-

hanced service matching, based on OWL-S [9]. Redondo et al. do not use the OSGi DS specification that makes the configuration of component easier. A possible future step for us would be to incorporate OWL-S in our model to enhance service matching.

Finally, the work in [12] extends OSGi component descriptions with a special-purpose description language which is used in the CACI context awareness infrastructure. CACI proposed a context aware component model which defines that a component has application ports as well as context ports. We basically have similar mechanisms for context information such as Quality-of-Service. Additionally, we are applying semantic web technologies that have more reasoning capabilities than what is provided by [12].

8. CONCLUSIONS AND FUTURE WORK

The SeMaPS ontologies consider runtime context information of pervasive systems by incorporating pervasive service computing characteristics. These ontologies are important in supporting the envisioned semantic-web based self-management approach adopted in Hydra [14][16]. The semantic web-based self-management is suitable for the openness of pervasive computing as explored in [15]. As semantic web-based context modeling is extensively used in pervasive computing, it is beneficial to uniformly make use of these assets for self-management purposes.

Dynamic service applications require extensive development and runtime support. This paper presented OWL-DS that extends OSGi's Declarative Services with support for global (architectural) constraints on composition and support for dynamic, contextual constraints. The approach has been validated through the Limbo web service compiler case study which shows that the Limbo architecture benefits from the ability to specify advanced component constraints.

Besides self-diagnosis as presented in previous papers, we are working on extending the application of SeMaPS ontologies for quality of service-based service selection and adaption, and other self-management work which also depends on contexts. Furthermore, the investigation of applying the OWL-DS idea and the Component ontology to the .NET platform is also under investigation. This idea can be potentially used for Hydra to validate whether a configuration of an application, or the middleware is legally configured at runtime, to make sure various constraints are met dynamically.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891).

9. REFERENCES

[1] H. CHEN, T. FININ, and A. JOSHI. An ontology for context-aware pervasive computing environments. *The Knowledge Engineering Review*, 18(03):197–207, 2004.

[2] P. Dolog. Model-driven navigation design for semantic web applications with the uml-guide. *Engineering Advanced Web Applications*, Dec. 2004.

[3] K. M. Hansen, W. Zhang, and G. Soares. Ontology-enabled generation of embeddedweb services. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 345–350, Redwood City, San Francisco Bay, USA, Jul. 2008.

[4] IST Amigo Project. Amigo middleware core: Prototype implementation and documentation, deliverable 3.2. Technical report, IST-2004-004182, 2006.

[5] OSGi Alliance. OSGi Service Platform – Service Compendium. Technical Report Release 4, Version 4.1, OSGi, April 2007.

[6] C. Pahl, S. Giesecke, and W. Hasselbring. An ontology-based approach for modelling architectural styles. In F. Oquendo and F. Oquendo, editors, *ECSA*, volume 4758 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2007.

[7] D.-J. Plas, M. Verheijen, H. Zwaal, and M. Hutschemaekers. Manipulating context information with swrl. *I/RS/2005/117, Freeband/A-MUSE/D3.12*, 2006.

[8] R. Redondo, A. Vilas, M. Cabrer, J. Arias, J. Duque, and A. Solla. Enhancing Residential Gateways: A Semantic OSGi Platform. *Intelligent Systems*, 23(1):32–40, 2008.

[9] A. Sillitti and G. Succi. Reuse: From components to services. In H. Mei, editor, *10th International Conference on Software Reuse (ICSR2008)*, volume 5030 of *LNCS*, pages 266–269, Beijing, China, 2008. Springer Verlag.

[10] R. Sterritt and M. Hinchey. Radical Concepts for Self-managing Ubiquitous and Pervasive Computing Environments. *LNCS*, 3825:370, 2006.

[11] T. Strang and C. Linnhoff-Popien. A Context Modeling Survey. *Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp*, pages 34–41, 2004.

[12] A. v. H. Tom Broens and M. van Sinderen. Infrastructural support for dynamic context bindings. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland, Nov 2005. LNCS, Springer-Verlag.

[13] H. Wang, Y. F. Li, J. Sun, H. Zhang, and J. Pan. A Semantic Web Approach to Feature Modeling and Verification. In *1st Workshop on Semantic Web Enabled Software Engineering (SWESE'05)*, Galway, Ireland, Nov 2005. LNCS, Springer-Verlag.

[14] W. Zhang and K. M. Hansen. An owl/swrl based diagnosis approach in a web service-based middleware for embedded and networked systems. In *The 20th International Conference on Software Engineering and Knowledge Engineering*, pages 893–898, Redwood City, San Francisco Bay, USA, Jul. 2008.

[15] W. Zhang and K. M. Hansen. Semantic web based self-management for a pervasive service middleware. In *Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Oct. 2008. To appear.

[16] W. Zhang and K. M. Hansen. Towards self-managened pervasive middleware using OWL/SWRL ontologies. In *HCP-2008 Proceedings, Part II, MRC 2008 – Fifth International Workshop on Modelling and Reasoning in Context*, pages 1–12, June 2008.

[9] `http://www.w3.org/Submission/OWL-S/`