Towards Open World Software Architectures with Semantic Architectural Styles, Components and Connectors

Weishan Zhang¹ and Klaus Marius Hansen¹,² and João Fernandes¹
¹Department of Computer Science, University of Aarhus
Aabogade 34, 8200 Århus N, Denmark
²Department of Computer Science, University of Iceland
Sudurgata, IS 101 Reykjavík, Iceland
{zhangws,marius, jfmf}@cs.au.dk

Abstract

There is a growing trend to develop open world software with the forthcoming of the pervasive computing era. Traditional research on software architecture, components, and connectors is not geared towards the characteristics of open world software, and are not easily used in open world environments. In this paper, we present an extensible knowledge base called SACoCo (Semantic Architectural styles, Components, and Connectors), by applying Web Ontology Language (OWL) and Semantic Web Rule Language (SWRL) where an Open World Assumption (OWA) is adopted. Runtime validations of component configurations and software architectural styles can be specified with SWRL rules. SACoCo improves the semantics of classical architectural styles, components and connectors, and can be used to dynamically validate software architectural styles and component configurations. Experiments with a pervasive web service compiler using OSGi and the Repository style show that the knowledge base is effective in improving the semantics of components and connectors, and is effective to validate component configurations and architectural styles, especially in open world environments.

1 Introduction and Motivation

Software architecture [25] has been widely researched. This includes work on software architectural styles, software components [26], and connectors [18]. There are also several industrial and research-based component models, such as OSGi DS (declarative services) [21], Fractal [5], Java Beans, Enterprise Java Bean (EJB)¹, and CORBA’s CCM [9]. Additionally, a family of software architectures form an architectural style, such as Pipe-Filter, Repository, Layered architecture, and Publish-Subscribe.

With the proliferation of pervasive computing [23] systems, which are operated as open systems [2], these existing technologies are not well prepared for open world computing. This is partially because the existing technologies are not designed particularly for open world computation, and that there are no adequately specified semantics for the used architectural styles, components and connectors, which makes this knowledge not easily sharable across system boundaries on the Internet. An example case is the OSGi [21] component model and its implementation as in Eclipse Equinox²: component interface mismatches are not sufficiently handled, for example, when there are multiple components providing the same service (interface) with the same name, Equinox just picks the component with the lowest ID to execute, and does not care about the exact semantics of the components and their relationships.

In open world settings, having to choose the most appropriate component(s) from a set of similar ones happens more often than in closed world computing where all components stay relatively static, whereas services can come and go anytime and anywhere in the open world settings. For open world software, service bindings may be changed at runtime [2], which makes it necessary to also validate architectural styles and configurations at runtime. As pointed out by Baresi et al. in [2], formal specifications for components are mandatory for open world software. Semantic Web descriptions of software architecture knowledge is a step towards richer formalization of component properties.

Therefore, we need to enhance these traditional component and connector technologies to add more semantic meanings. As preliminary work on ontology driven architecture³ has shown, the Semantic Web [3], especially the

¹http://java.sun.com/products/ejb/
²http://www.eclipse.org/equinox/
³Ontology Driven Architectures and Potential Uses of
OWL\textsuperscript{4}[13] is useful in addressing the shortcomings mentioned above. The Semantic Web is designed to share knowledge across the global Internet through ontologies knowledge base, and can be utilized for sharing architecture knowledge across system boundaries. OWL-DL (as a sub-language of OWL) and its foundation Description Logic (DL) [19] can formally and rigorously define vocabularies and relationships among the vocabularies for architecture knowledge, and can provide reasoning capabilities for checking model consistencies and inferring new relationships among concepts. Additionally, SWRL\textsuperscript{5}[12] can be used to specify architecture constraints, which is vital for architectural styles and configurations validation.

The Open World Assumption (OWA) is characterized by that statements about knowledge that is not included in or inferred from the knowledge explicitly recorded in a system is considered unknown, rather than wrong or false as assumed in the closed world assumption. The OWA applies to knowledge representations where the system of knowledge can never be known to have been completely described in advance, which is quite consistent with the characteristics of open world environments like pervasive computing systems, which are intrinsically open and dynamic. Therefore it is natural to apply the OWA to pervasive computing systems where the characteristics of OWA can be utilized to build open world software [2]. OWL and SWRL are adopting the OWA, and can provide reasoning capabilities based on incomplete knowledge, which is arguably a more natural fit for the open world settings as discussed in [30].

Hence in this paper, we propose an OWL-DL and SWRL based knowledge base for software architectural styles, components and connectors (called SACoCo). Runtime validation of component configurations and software architectural styles can be specified with SWRL rules. SACoCo improves the semantics of classical architectural styles, components and connectors, which can be used to dynamically validate software architectural styles and configurations. Experiments with a pervasive web service compiler using OSGi and the Repository style, show the effectiveness of our approach, especially in the open world environments.

The rest of the paper is structured as follows: Section 2 presents a high level overview of the SACoCo models. Then in Section 3, we present the semantic architectural style model. The design of the semantic component and connector models is described in Section 4 and 5. We then show how to conduct an architectural style validation in Section 6. Section 7 shows a prototype implementation to conduct architecture validation at run time. In Section 8 we exemplify the usage of SACoCo with a Repository style and OSGi DS. We compare our work with the related work in Section 9. Conclusions and future work end the paper.

2 Overview of the SACoCo models

We developed SACoCo models using OWL-DL. There are a number of objectives for SACoCo: formally specifying vocabularies and properties for software architectural styles and components/connectors; providing reasoning capabilities to understand relationships between the related concepts and hence to better support design decisions for architectural style selection and component/connector selection; and runtime validating component configurations following certain architectural styles.

SACoCo has a number of ontologies, including one for atomic connectors (AtomicConnector ontology as in Figure 4) and another one for composite connectors (CompositeConnector ontology), an ontology for high level component concepts (ServiceComponent ontology) which imports separate ontologies for specific component models (e.g. OSGi), an ontology for architectural styles (ArchStyle ontology), and additionally we have another ontology to specify architecture constraint rules using SWRL (ArchRule ontology). The relationships between these ontologies in SACoCo are shown in Figure 1.

![Figure 1. SACoCo ontologies structure](image)

According to Shaw and Garlan [25], the core concepts in software architecture are Components (main computation elements) and their services that are required or provided through Ports which define the points of interactions, and Connectors which link the Components through Roles which identify the participants in interactions [14].

A Component in SACoCo may be modeled with different flavors, for example OSGi and EJB (where details are modeled in separate ontologies according to their component model specifications). These different kinds of component models are sub-classes of Component concept. Similarly, different types of atomic connectors are sub-classes of the Connector concept, and the CompositeConnectors are composed of atomic connectors. The key to view a component and connector is to understand the services they can provide and which they require. Therefore we modeled the

\textsuperscript{4}http://www.w3.org/2001/sw/BestPractices/SE/ODA/
\textsuperscript{5}http://www.w3.org/TR/owl-ref/
Service concept using Service Profile (modeled with the ServiceProfile ontology) and Service Model (modeled with the Process ontology), applying the same ideas from OWL-S6 ontologies for Semantic Web services. This is the reason why we call the semantic component model ServiceComponent (Figure 1). Figure 2 shows the main concepts and relationships of these concepts in SACoCo.

![Diagram of concepts in SACoCo](http://www.w3.org/Submission/OWL-S/)

**Figure 2. Overview of concepts in SACoCo**

### 3 Semantic architectural styles

The semantic models for several architectural styles are developed based on published work. For example, the models for a Client-Server style, a Pipe-Filter style are based on [10], a Publish-Subscribe style, and a Peer-to-Peer style are based on [6], and other work on software architectural styles [24] [4] in which topology constraints are clearly specified.

An important issue for semantic architecture modeling is to model software architecture constraints, which are design decisions behind the rationale of architectural styles. We model three different types (called Type A, Type B, and Type C constraints respectively in the order as listed below) of architecture constraints:

- **Structural parts in an architectural style.** This will make sure that an architectural style has correct types of components, connectors, roles, and ports. For example, a Repository Style should have at least one Repository component which has port Provide, at least one Access connector that has roles Provider and User.
- **Connections between structural parts, including topological constraints[4]/[24].** That is to say, how a component, connector, role, and port are connected with each other. For example, the Repository Style has a “Star” control topology and data topology, the port Provide in the Repository component should be connected to the role Provider in the Access connector.
- **Externally visible properties of the structural parts.** This includes some quality attributes, such as performance, fault tolerance, feature combination constraints, and also some functional properties, for example the required software and hardware platforms to run a component.

OWL-DL can model vocabularies and their relationships in a formal way. It provides good capabilities for specifying the first architecture constraints (Type A) where architecture vocabularies are modeled as concepts and/or properties. Additionally, component and connectors can be annotated with functional and nonfunctional properties to model the third type of constraints (Type C). There are also cardinality and universal/existential constraint constructs in OWL-DL that can specify constraints of the second type (Type B), but mainly local constraints of Type B can be modelled.

OWL-DL itself could not specify architecture constraints span across multiple concepts and concept properties (relationships) [16], which is critical to express architecture constraints, especially the global Type B architecture constraints. In such cases, SWRL is used instead to achieve this as shown in Section 8. That is to say, to specify global constraints and other types of constraints that are beyond the capabilities of OWL-DL, we are applying SWRL to enhance constraint specification capabilities of OWL-DL. An example of such a rule named “check_OSGi_REFERENCE_DETAILS” is shown in Section 8.

Additionally, when we build the knowledge base for architectural styles, their quality attributes and application domains as classified in [20] are considered, in order to facilitate a designer in choosing the right style to achieve required quality requirements. For example, the Pipe-Filters are used in situations where data stream processing is needed, and has Simplicity, Reusability, and Maintainability as its quality attributes.

### 4 Semantic Components

Lau and Wang presented a relatively comprehensive survey on software component models in [15], but did not cover OSGi components, which are widely used both in industry and in academia for realizing service orientation. We refer to this work as a guide for various component models (except OSGi). In this work [15], some high level concepts of a component, such as component composition language, and component definition language are defined, which are also modeled in our ServiceComponent ontology. When
it comes to the detailed modeling of a specific component model, we are following the component specifications for that component model. Different component models are siblings as subclasses of the `Component` concept. The existential constraints in OWL-DL are used to define the relationships between these high level concepts.

Some industry component models are evolving quickly, for example EJB. For EJB3, there is no need to explicitly implement remote and home interfaces as previously did, and an Entity bean is not part of EJB3 as it is the same as a normal Java Bean (not Enterprise specific anymore). We model such a situation with different concepts (and of course properties) for different versions of EJB.

The semantic model for the Fractal [5] component is developed mainly on its ADL specification\(^7\) and other Fractal publications. The Fractal ADL can specify component compositions, which is modeled as a `hasComposition-Comp` object property in the Fractal component ontology.

The OSGi component ontology is based on the OSGi’s Declarative Service specification [21]. It specifies the `Component` (as a concept) dynamic status, for example whether it is enabled, and also static characteristics such as its reference(s) to other service(s), its implementation interface(s), and service(s) provided. As we are going to illustrate the SWRL rules using OSGi component as examples, Figure 3 shows partially the details of the OSGi component ontology.

![Figure 3. OSGi component ontology (Partial)](image)

We adopted OWL-S 1.2 as the base for the semantic models for both traditional component services and web services. The ServiceProfile ontology can be used to categorize services for better service matching. Services are modeled as processes (in the Process ontology) in which input, output, precondition, and results are used to describe a service. An atomic process models an action that a service can perform in a single interaction, whereas a simple process is an abstraction mechanism to provide multiple views of the same process. Having a look at the component models of OSGi, EJB, etc., every operation in a component can be modeled as an atomic process, and then a simple process for the whole component is composed of these atomic processes using the `realizedBy` object property in the Process ontology.

As an example, Table 1 shows component dependencies for the Limbo pervasive service compiler [11], following a Repository style (“Y” stands for having a reference to an operation). There is one Repository component and four Repository clients. The Limbo Repository component has five operations which are defined as five atomic processes in the Process ontology, where the return types are modeled as process Outputs, and method signatures are modeled as Inputs. An instance of SimpleProcess is defined which is composed with these five AtomicProcesses correspondingly. In these Repository Client components, instances of SimpleProcess for each operation are defined in a similar way. If there is a reference from the Repository Client to the Repository component in a method, the atomic process for this method will have a Participant instance which should be the Repository component. The inputs of this atomic process will contain both the signature of the referenced operation together with its return type, and method signature of itself.

<table>
<thead>
<tr>
<th>Limbo Repository operations</th>
<th>Limbo</th>
<th>UPnP</th>
<th>State Machine</th>
<th>Probe</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI getOntologyURI()</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Definition getWSDL()</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>File getWSDLFile()</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HashMap&lt;String,String&gt; getLimboConfiguration()</td>
<td>Y</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URI getHydraOntologyExtension(File wsdlFile)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Table 1. Limbo component references](image)

In this way, all component models are enhanced with capabilities for knowing their interface details, i.e., the details of a component behavior. It provides a unified way for modeling the details of component services, including service dependencies among components, which can be used at runtime to conduct validations of architectural styles and configurations. This is the key to find the correct component when there are multiple components where interfaces should be matched.

An important benefit of these semantic component models are that they are clarifying the relationships between some of the concepts for different component models, which can facilitate the usage and understanding of these component models. For example, the concept of OSGi component `Reference` is a subclass of `ComponentRequiredService`, and Fractal `Binding` is a subclass of `ComponentReference`.

Other component models such as the CORBA CCM\(^8\), Microsoft .NET, COM/DCOM are also included in the set of component ontologies.

5 Semantic Connectors

Although there are a number of classifications for software connectors in literature, the one from Mehta [18] is arguably the most accepted as it provides a comprehensive list of connectors, which are used as the basis for our semantic connector model. All the eight type of connectors, including Procedure Call, Event, Linkage, Distributor, Arbitrator, Stream, Data Access, and Adaptor are modeled in the AtomicConnector ontology as shown in Figure 4.

Figure 4. Atomic connector model (Partial)

Connectors can be composed to form a composite connector, for example, thirteen different types of distribution connectors are explored for highly distributed data intensive systems by Mattmann [17]. In his dissertation, details on how these connectors are composed with properties are specified in XML DTDs. We encode all this knowledge in our connector models. For the modeling of a composite connector where the order of connectors matters, we use OWL sequences as proposed in [8]. From [17], we can say that a basic distribution connector is composed with three connectors in this order: Data Access, Stream, and Distributor. This is a basic distribution connector, and a SOAP\(^9\) connector is one specialization of it where a SOAP procedure call is first involved.

In practice, connectors are not always first class citizens in design and implementation. Therefore connectors may not easily be spotted and may be mixed with components, and are not easily separated from components. In this case, we will allow that components are directly connected with each other, rather than through a trivial connector.

6 SACoCo-based architectural styles and configurations validation at runtime

Due to the open and dynamic nature of pervasive computing and Internet scale computing, software components providing services can join and leave a system anytime, hence component configurations can be changed more often than those running in closed world environments. This naturally requires that these new configurations should be valid and still follow certain architectural styles. That is to say, we should dynamically validate component configurations and architectural styles.

We assume that instances of components and connectors are annotated with their roles in a semantic architectural style encoded in the SACoCo models. It does not matter which and how many roles they can play. This can be achieved at design time to designate the nature of a component and connector. For example, a Repository style has at least a Repository component and a Repository Client component. We also assume that details of components (for example, they follow a certain component model) and connectors (for example, they follow the details as classified in [18] [17]) are known when they are validated for a certain configuration. This may seemingly contradict to the OWA, but it is reasonable as a starting point towards an open world software architecture. As component and connectors may change at runtime, we will check that the new configuration is still following a designated architectural style. Therefore, if there are the required components and connectors, the number of component/connectors are meeting constraints and other requirements, and the components/connectors are correctly referenced by each other, then these components are following a designated architectural style.

An application or system can have multiple architectural styles implemented. Therefore the validation of architecture styles will be a loop over these implemented architecture styles. Figure 5 shows a general approach for architectural styles and configurations validation using SWRL rules in SACoCo models. Firstly, there should be components/connectors corresponding to an architectural style (maybe connectors are not used explicitly for some situations). This will ensure that a specific configuration following an architectural style has the corresponding architecture.

---

\(^8\)http://ditec.um.es/ dsevilla/ccm/

\(^9\)http://www.w3.org/TR/soap/
elements. The total number of the component/connectors should be the same as these ones under consideration (or larger when a component/connector plays multiple architecture roles if simply sum the figures got from SWRL queries for the components/connectors numbers).

Then these components/connectors need to be correctly referenced by each other. For example, a Repository client should be connected to a Repository component, not necessarily directly connected to, but possibly with a connector as an intermediate element. This will also make sure that the correct topology will be respected (in the Repository style, it is a “Star” topology as said). If there is a Repository that has no reference(s), then it is a dangling component, which may need to be deactivated. In OSGi, this can be achieved by setting its enabled property to “false”. If there is only one Repository component, and it is a dangling component, then it is an invalid configuration. As component services may not be exactly matched, for example interface signature mismatch, the mismatches should be identified according to the semantic details of components and services.

Some other property constraints for an architectural style should be checked, such as quality of service requirements, software/hardware platform constraints, and other contextual constraints. An example of such constraints is given in Section 8.5.

Using the Equinox DS service, we can easily get the whole list of component instances to be validated. After checking the valid references, invalid references, and possible mismatches between components, then we can get a list of components/connectors for a valid configuration. If there are mismatches, we can get some suggestions to resolve the mismatches through rule inferring, and may help to add new components/connectors into the current configuration.

7 Prototype Implementation

In accordance with the problem that Equinox OSGi is not working well in the open computing environments as discussed in the introduction, we implemented a prototype to enhance Eclipse Equinox, and the prototype has capabilities of awareness of component semantics based on SACoCo ontologies, and are validating architectural styles and component configurations at run time. The architecture of the prototype implementation is shown in Figure 6, in which there are EnhancedEquinoxDS bundle, SACoCoConfigValidation bundle.

This EnhancedEquinoxDS bundle is used (which is an extension to the original implementation) to discover and maintain a model of the component instance topology, where there is a Binding Listener that knows when services 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 SACoCo configurator to maintain a model of component instances, their services, and their relationships. The registered services of the component are retrieved from the bundle context of the component.

In order to get the detailed information about services provided by components that are loaded in an OSGi framework, the SACoCoConfigValidation bundle gets a list of enabled component description properties from the EnhancedEquinoxDS bundle (as another extension to the original DS implementation). These component details are then updated into the SACoCo models as a set of component instances to be validated. Based on this information, the SACoCoConfigValidation bundle validates component configurations by executing the related SWRL rules, and inferring whether they are valid or invalid, whether services are matching or not according to semantic constraints specified as outlined in the former sections.

The Protege-OWL/SWRL APIs10 (which are the only SWRL APIs currently available) are used in the prototype to demonstrate the SACoCo usage and architectural styles and configurations validation. In the prototype, all rules in a step are executed with rule grouping features, for example the checking of the Repository style elements can be executed with the key word of “check_RepositoryStyle” from the rule names, and the key words can be combined with boolean operations. From our experience, SWRL rule grouping will enhance the performance of validation [30].

The RuleProcessing component is used to execute a single rule and to retrieve the corresponding results. The RuleProcessing component is responsible for the execution of a rule group. Currently the rule grouping is based on the name of the SWRL rules, which can be logically combined (e.g. AND, OR). The rule processing features are generic and can be used to process all different kinds of rules, i.e. they are independent of architecture styles, com-

10http://protege.stanford.edu/
ponents/connectors.

In the next section, we will illustrate how to use SACoCo models to express architecture constraints using SWRL. To save space, we assume that there is only one component which meets the semantic requirements. We will report the choosing of multiple components in another paper based on the component utilities using the SACoCo models.

8 An example: the Repository style and OSGi DS with Limbo

As introduced in [11], Limbo components are implemented as OSGi bundles. The Component Limbo provides a Generator service that Backends use, and will produce web service stubs and skeletons. At runtime Limbo selects and uses a set of Backends based on its configuration. The Limbo component requires the presence of at least one Backend and exactly one Repository. SWRL is used to specify architecture constraints for Limbo based on the SACoCo models.

SWRL is a Horn rules extension to OWL-DL and shares its formal semantics. 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. In our practice, all variables in SWRL rules are bound only to known individuals in an ontology in order to develop DL-Safe rules to make them decidable. In a SWRL rule, the symbol “∧” means conjunction, and “?x” stands for a variable, “→” means implication, and if there is no “?” in the variable, then it is an individual. Now we will show the constraints rules in the way as shown in Figure 5.

8.1 Check Architecture elements and their numbers

The rule “check_RepositoryStyle_Repository” is used to retrieve the Repository components that plays the “Repository” role in a Repository Style in the current configuration. Similarly there are rules for checking the repository clients named “check_RepositoryStyle_Repositoryclient”.

**Rule: check_RepositoryStyle_Repository**

```
archstyle : CurrentConfiguration(?con) ∧
archstyle : hasArchitecturePart(?con, ?comp1) ∧
architectureRole(?comp1, ?role1) ∧
archstyle : archPartName(?role1, ?rolename) ∧
surlb : equal(rolename, “Repository”) → sqwrl : selectDistinct(?comp1)
```

The checking of the number of architecture elements can be conducted in two ways: the first approach is to count the number of above rule query results using Java code. The second approach is using another set of rules, for example “check_RepositoryStyle_Repository_count” (rule name ending with “_count”), which has the same kind of body as “check_RepositoryStyle_Repository”, but the head is changed to `sqwrl : countDistinct(?comp1)`.

8.2 Check reference relationships

This step is related to a specific component model, since different component models have different ways to model component relationships. For the OSGi component model and Repository style, the rule “check_OSGi_Reference_noDetails” retrieves all Repository components in the current configuration. If a component has a reference which has cardinality of the form “1..n” (at least one reference to other service), then there must be a component providing that required service.

**Rule: check_OSGi_Reference_noDetails**

```
archstyle : CurrentConfiguration(?con) ∧
archstyle : hasArchitecturePart(?con, ?comp1) ∧
osgi : componentName(?comp1, ?compname1) ∧
osgi : reference(?comp1, ?ref1) ∧
osgi : cardinality(?ref1, ?card1) ∧
surlb : containsIgnoreCase(?card1, “1.”) ∧
osgi : interface(?ref1, ?inter1) ∧
osgi : interfaceName(?inter1, ?name1) ∧
archstyle : hasArchitecturePart(?con, ?comp2) ∧
ar\text{chitectureRole(?comp2, ?role2) ∧
}\text{architectureRole(?comp2, ?role2) ∧
archstyle : archPartName(?role2, ?rolename) ∧
surlb : equal(?rolename, “Repository”) ∧
osgi : service(?comp2, ?serv2) ∧
osgi : provide(?serv2, ?inter2) ∧
osgi : interfaceName(?inter2, ?name2) ∧
osgi : componentName(?comp2, ?compname2) ∧
surlb : equal(?name1, ?name2) → sqwrl : selectDistinct(?comp1, ?comp2)
```

8.3 Identifying valid component references

Assume that we have two Repository components loaded by DS and the two components are implementing two Repository interfaces that differ only with the following operation (refer to Table 1):

- URI getHydraOntologyExtension (File wsdlFile);
- URI getHydraOntologyExtension (String wsdlFile);

Limbo needs to be bound to the Repository interface that has the signature as the first method. The Eclipse Equinox DS bundle binds Limbo to the Repository component that has the lowest bundle id, without respecting the interface signature it has. If it is bound to the correct Repository, then Limbo can run successfully. But if the Repository with the lowest bundle id is the one that provides the same method but with a String parameter, Limbo will not work.

The solution to this is to provide more semantic meaning to the DS service to facilitate making choices for such situations. As introduced in Section 4, the ServiceComponent ontology and the Process ontology can help with fully specifying valid OSGi component references taking interface signatures into consideration. Here in order to
simplify the writing of rules (the reason being that current Protege SWRL APIs can not parse rdf:XMLLiteral), we changed the range of the data type property parameterValue to xsd:string, which is different to the original OWL-S Process ontology. To correctly justify a reference, the component package and component name, method name, method signature including data type and order, and return type, should be consistent in both referencing and referenced components. Using the service details as provided by the ServiceProfile and Process ontologies, we can then retrieve the details of the services and its method signatures.

SWRL builtins can then be applied to conduct comparisons and other simple math calculations. This rule is named “check_OSGi_Reference_Details” as follows. The BODY_OF_RULE_check_OSGi_Reference_Details means that the body part of the rule “check_OSGi_Reference_noDetails” should be included here. This will save space which is limited for the paper.

**Rule: check_OSGi_Reference_Details**

```swrl
```

As can be noted from this rule, in a referencing component, the references to another component is modeled in the hasInput datatype property in the Process ontology, in the format as “component name(including package name)+operation name#input types with orders$Return type”. Then this information is compared with that from the referenced component with respect to a specific interface, which is modeled as an atomic process as discussed in Section 4. If they are exactly matched, then the references are valid.

### 8.4 Reference mismatch identification and reconfiguration for matching

From the above rule, now it becomes clear how to detect the mismatch of service interfaces using the semantic information in semantic components and connectors. For example, the rule “check_OSGi_Reference_SignatureMismatch” is used to identify the signature mismatch of interfaces. Additionally if a connector resolving the interface mismatch exists (its instance is adaptConnector1 as in the AtomicConnector ontology), we can then infer that an adaptor connector needs to be added to the current configuration in order to make the references matching, shown in the last line of the rule. Similarly, the BODY_OF_RULE_check_OSGi_Reference_Details (before the first swrlb:stringConcat line) means that all lines before the last 9th line of rule “check_OSGi_Reference_noDetails” will be included in the “check_OSGi_Reference_SignatureMismatch” rule.

**Rule: check OSGi Reference SignatureMismatch**

```swrl
```

As can be noted from this rule, in a referencing component, the references to another component is modeled in the hasInput datatype property in the Process ontology, in the format as “component name(including package name)+operation name#input types with orders$Return type”. Then this information is compared with that from the referenced component with respect to a specific interface, which is modeled as an atomic process as discussed in Section 4. If they are exactly matched, then the references are valid.

### 8.5 Contextual constraints specification

The Limbo compiler can generate pervasive service code for the JavaME and JavaSE (standalone and OSGi) platforms, with client and/or server as options, and also UPnP descriptions for services and devices. Some of the generation combinations are not meaningful. For example, if a UPnP component is started, and the generation type is only for a web service client, then this combination is not valid as the UPnP service is only meaningful in the server. This can be specified with the following rule.

```swrl
archstyle : hasArchitecturePart(?icon, adaptConnector1)∧ archstyle : hasArchitecturePart(?icon, adaptConnector1)∧ osgi : implementationClass(?temp, ?class)∧ limboGenerationType(?icon, “UPnPComponent”)∧ swrlb : stringConcat(?str1, “U PnPComponent”)?
```

### 8.6 Discussion

With a Thinkpad T61p under WindowsXP, the prototype needs around 300ms to process the validation rules as show above, which is acceptable in practice taking into consideration the complexity involved in OWL/SWRL reasoning. From these experiments, we can see that SACoCo
models are adding important values to the existing component/connector technologies, and are effective for runtime validation of architectural styles and configurations. Although the rules shown above are architectural style specific (Repository style), and component model specific, they can easily be adapted to validate other architectural styles in two ways: the first is to parameterize the architectural style roles (for example, “Repository” vs. “Pipe”), and dynamically generate related rules for an architectural style at run time. The second approach is to implement new rules based on the existing rules.

9 Related work

Validating and enforcing architectural constraints have been previously researched. [1] presented some examples of architectural constraints. [27] presented an architectural constraint language (ACL) based on UML and OCL, within the context of model transformation and model driven engineering. The strong point of this approach is that it can be used in the whole component based development life cycle. On the other hand, this approach is hard to handle global constraints and contextual constraints that may have a dynamic nature. And this approach does not scale well: for every component model, a different ACL profile should be defined to cover the transformations.

Armani was designed to capture software architecture knowledge [7], and is focused on the structural properties of an architecture. It supports component and connector refinement, and can be used to specify component number restrictions in an architecture, and traverse component instances to check whether they are reachable. The Semantic Web-based approach of in this paper is adopting description logic to conduct reasoning to check refinement of architectural styles, components, and connectors, and SWRL rules can be used to traverse component and connector instances to infer the conformation of an architectural style and validate a configuration. This approach is more suitable for open world settings.

Kim and Garlan [14] applied Alloy to specify architectural constraints. Another work in this aspect by Wong et al. [29] pointed out the scalability problem of the work by Kim and Garlan, arisen from the large search space for complex situations. These works did not consider dynamism of open world computation either. The SACoCo models based architectural style validation could cover the style refinement, style combination as shown by Pahl [22].

An interesting work by Pahl in [22] applied the basic description logic ALC to model software architectural style concentrating on the static structure of an architectural style. The SHOIN(D) description logic behind OWL-DL is more expressive than ALC description logic [19], therefore we can express more with OWL-DL than what is presented in [22], for example to define hasComponent and hasConnector as sub-properties of hasArchitecturePart. Besides static structure, we are focusing on dynamism of component configurations in the open world environments where architectural styles and configurations can be validated at runtime, in which SWRL rules play a key role.

The knowledge base presented in [20] is focusing on wireless services. It provides service classification specialized for wireless service engineering, and links quality attributes to modeled styles/patterns. We incorporated this knowledge in our ontologies. SACoCo is a relatively comprehensive OWL/SWRL knowledge base covering software architectural style, component/connector, not limited to one domain. SACoCo is open and provide reasoning potential for open world software where dynamism is considered.

Wang et al. [28] presented a feature modeling approach in which legal feature combinations configuration rules are expressed using OWL. This approach is mainly a design time method and is not able to cope with new contexts at runtime. SACoCo models can be used at design time and runtime constraints are mainly implemented with SWRL rules. The SWRL based configuration constraints specification is more suitable for dynamic constraints specification and is more flexible, which can express the specifications not achievable with OWL-DL itself.

10 Conclusions and future work

Semantic software architecture assets, including architectural styles, component/connector provide a good solution to dynamic service bindings to address challenges for open world software [2]. We present an OWL-DL/SWRL based semantic models for software architecture assets called SACoCo, which are used to dynamically validate software architectural styles and configurations at run time in open world environments. The SACoCo models are describing both structural and behavioral knowledge of component/connectors, which are open and can be used in open world services where services can come and go anytime and anywhere.

We target making the SACoCo models a full-fledged and unified knowledge base for both design and run time usage. Currently SACoCo models have not covered all component models, software architectural styles, and connectors. We will add those remaining gradually, but will focus on embedded and pervasive computing domain knowledge. The SACoCo models are part of ontologies for supporting a Semantic Web-based self-management approach as proposed in [30], where architecture based self-configuration and self-healing will be conducted. Finally, we will explore the open world software architecture in a larger scale, starting with the complete Hydra middleware using the SACoCo models, and then extend the experimentation with other per-
Assistive computing systems.

Acknowledgments

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891). Thanks to Mads Ingstrup, Kristian Ellebæk Kjær and Jeppe Brønsted for comments on the paper.

References