Design Software Architecture Models using Ontology

Jing Sun¹, Hai H. Wang² and Tianming Hu³

¹Department of Computer Science, The University of Auckland, New Zealand
j.sun@cs.auckland.ac.nz

²School of Engineering and Science, Aston University, United Kingdom
h.wang10@aston.ac.uk

³Department of Computer Science, Dongguan University of Technology, China
tmhhu@ieee.org

Abstract

Software architecture plays an essential role in the high level description of a system design, where the structure and communication are emphasized. Despite its importance in the software engineering process, the lack of formal description and automated verification hinders the development of good software architecture models. In this paper, we present an approach to support the rigorous design and verification of software architecture models using the semantic web technology. We view software architecture models as ontology representations, where their structures and communication constraints are captured by the Web Ontology Language (OWL) and the Semantic Web Rule Language (SWRL). Specific configurations on the design are represented as concrete instances of the ontology, to which their structures and dynamic behaviors must conform. Furthermore, ontology reasoning tools can be applied to perform various automated verification on the design to ensure correctness, such as consistency checking, style recognition, and behavioral inference.

1. Introduction

Software architecture plays an essential role in the high level design of a software system. Analogy to civil engineering, it represents the fundamental structural and behavioral descriptions of the system under design. Despite its importance in the software engineering process, the lack of formal description and automated verification hinders the development of good architecture models. Traditionally, software architectures are specified using diagrammatic and textual notations [8, 9]. However, architecture models defined in this manner are likely to be inconsistent and error prone due to the informality of the description and the lack of means of rigorous verification to ensure the correctness. As a result, formal modeling techniques have been applied to software architecture descriptions [1, 4], which is aimed at achieving precise specification and rigorous verification of the intended structures and behaviors in the design. The main advantage of such verifications is the ability to determine whether a modeled structure can correctly satisfy a set of given properties in the requirements of a system in order to assure quality of design. However, one drawback of many existing approaches in the automated verification of software architecture models is their limited scalability, as large architecture models are computationally expensive to process. For example, a recent work by Kim and Garlan [7] proposed the modeling and verification of architecture styles using the Alloy language and its analyzer. In their approach, a few architecture styles based on ACME descriptions were translated and verified using the Alloy analyzer. Although it offers a useful insight to the ability of applying formal modeling in automating the verification of architecture descriptions, the performance issue is a practical limitation of the research. The problem arose from that large-sized architecture models dramatically expand the search spaces of the verification in the Alloy SAT solver. To overcome this problem, Wong et al. [11] recently proposed a model splitting approach for the parallel verification of Alloy based architecture models using their underlying styles. The approach improved the performance of the verification. However, the overheads of the model decomposition as well as the dependency issues among the sub-models during the parallel verification phase still remain as challenges.

On the other hand, there are mature large-scale reasoning means from the Semantic Web [2] technology that can be adopted to provide more promising solutions towards the problem, both in terms of knowledge representation and automated verification. The semantic web vision recommended by the World Wide Web Consortium (W3C) has emerged as the next generation of the web from the late
nineties. It extends the current web by assigning web content with a well-defined meaning, aimed at enabling intelligent machine processing of web resources. Description Logic (DL) based ontology languages, such as the Web Ontology Language (OWL) [6] and Semantic Web Rule Language (SWRL) [5], have been proposed to meet the needs of representing the complicated relationships among different entities. One of the advantages of the ontology languages, such as OWL and SWRL, is that DL reasoning engines, such as Pellet [10] and Jess [3], can be effectively used to perform large-scale automated reasoning on ontologies and their instances, e.g., subsumption reasoning, consistency checking, classification and knowledge inference. The scale of such verifications is usually in terms of thousands of ontological individuals (instances). To a certain extent, the semantic web approach is essentially an application of formal methods into the web community, where web resources are formally specified using description logic notations and rigorously verified using ontological reasoning engines.

In this paper, we explore the synergy between software architecture modeling and the semantic web ontological reasoning. We view software architecture models as ontology representations, where their structures and communications must hold. The overall approach of our ontological based software architecture modeling and verification is presented in Figure 1. As shown from the diagram, we first define ontology models that are specific to software architecture design using the OWL/SWRL languages based on different architecture styles. OWL is used to specify the structure and relationship constraints among the architecture terminologies, e.g., the participating components, connectors and their connections. SWRL is used to capture the dynamic interactions within the architecture models as additional constraints, e.g., the requester and provider communication protocol of a Client and Server style. The OWL ontology plus the SWRL rules form the meta-ontology that can be used to create customized architecture specifications. When users define their own customized architecture models by extending the meta-ontology (styles), all the meta-level rules are automatically enforced upon the customized models, i.e., the particular structures and communications are carried over onto the new design. A customized ontology can also consist of system specific terminologies and rules. For example, we can customize the meta-ontology into defining the architecture model of a specific internet application that extends the Client and Server style and yet has its own additional communication behaviors. After creating the customized ontology, architecture configurations can be defined as instances of the underlying ontology design, e.g., specific numbers of web servers and named clients involved in the system layout, etc.

Furthermore, there are two levels of verification that can be performed in our approach to ensure the correctness of an architectural design, namely, the ontology and the instance levels. The former ensures the correctness of the architecture model itself, e.g., no conflicting connection and communication constraints among the components. The latter ensures the conformance of a particular configuration with respect to its design, e.g., a certain configuration should always have its pre-defined structures and perform its own communication protocols. Moreover, ontology reasoners can be used to fully automate the verification process. For example, the Pellet reasoner can be used to check the consistencies of the ontology and instance models, and the Jess rule engine can be used to automatically derive the communication sequences on configurations, etc. In addition, ontology classifiers can be used to automatically recognize the style patterns that are used in a design at the ontology level.

The remainder of the paper is organized as follows. Section 2 presents ontology definitions for software architecture modeling which includes commonly used architecture styles. Section 3 illustrates the use of the architecture ontology in modeling and verifying a customized Three-Tier network architecture as a case study. Finally, section 4 concludes the paper and discusses the future work.

2. Ontology for Software Architecture Design

Software architecture modeling focuses on the high level description of a system in term of the elements in the system and the interactions among the elements. Components and connectors are two fundamental units in an architecture description [1]. Components describe the identifiable computation entities of a system, where connectors specify the patterns of communication among these entities. Each component consists of a set of ports as its external visible interfaces. A connector consists of a set of roles that are used to describe the patterns of a particular kind of communication. In order to establish a connection between different components of the system, the roles of a connector must be attached to the ports of the participating components.
The use of connectors separates the concern of computation from communication in a system description. It further promotes encapsulation and reusability in architecture modeling. In OWL, we define the following classes and properties to represent the different entities and their relationships in an architectural design.

\[
\begin{align*}
\text{Component} & \sqsubseteq T \\
\text{Connector} & \sqsubseteq T \\
\text{Port} & \sqsubseteq T \\
\text{Role} & \sqsubseteq T \\
\text{Process} & \sqsubseteq T \\
\text{Id} & \sqsubseteq T \\
\text{Component} \sqcap \text{Connector} \sqcap \text{Port} \sqcap \text{Role} \sqcap \text{Process} \sqcap \text{Id} & = \bot
\end{align*}
\]

The above defines Component, Connector, Port, Role and Process as mutually disjoint classes. They are top level entities in the architecture description, where the Process represents an atomic behavior that a port or a role can perform. The Id defines the message identification number that is shared by different processes involved in a communication. For example, the processes in a client and server communication, e.g., sending a request, invoking the server, server returning the result, receiving the result back by the client, may share the same message id for referring to the same client request interacting through the client and the server.

\[
\begin{align*}
\geq 1 \text{hasPort} & \sqsubseteq \text{Component} \quad T \sqsubseteq \forall \text{hasPort.Port} \\
\geq 1 \text{hasRole} & \sqsubseteq \text{Connector} \quad T \sqsubseteq \forall \text{hasRole.Role} \\
\geq 1 \text{hasAttachment} & \sqsubseteq \text{Role} \\
T & \sqsubseteq \forall \text{hasAttachment.Port} \quad T \sqsubseteq \forall \text{hasAttachment.Rate} \\
is\text{AttachedTo} & = (~\text{hasAttachment}) \\
\geq 1 \text{hasAction} & \sqsubseteq (\text{Port} \cup \text{Role}) \\
T & \sqsubseteq \forall \text{hasAction.Process} \\
\geq 1 \text{hasCommunication} & \sqsubseteq \text{Process} \\
T & \sqsubseteq \forall \text{hasCommunication.Process}
\end{align*}
\]

After defining the basic entities, we use OWL property definitions to further capture the relationships among them. For example, the hasPort is a relation from a Component as its domain to a Port as its range, which simply denotes that a component has a set of ports as its interfaces. Similarly, the hasRole represents that a connector has a set of roles as its participating parties in the connection. The attachment of a role to a specific port is defined as the property hasAttachment. Note that hasAttachment is defined as a functional property from a role to a port, which means that a port can be attached to different roles, but not vice versa. The relation isAttachedTo is an inverse relation to the hasAttachment function, which maps a port back to its attached role. We also define two more relationships to capture the dynamic interactions within a component or a connector and their communications. The property hasAction describes the set of atomic processes that a port or a role is able to perform. The property hasCommunication captures the interaction sequences of a particular communication (protocol). In addition, the connectivity constraint need to be enforced upon the model, i.e., in order to establish a valid attachment between a role and a port, the port must be able to perform the behaviors (actions) that the role has. Otherwise, the port can not 'play' the role in the connection due to behavior mismatch. This can be easily specified using a SWRL rule as follows.

\[
\text{hasAttachment}(r, ?p) \land \text{hasAction}(r, ?x) \\
\rightarrow \text{hasAction}(?p, ?x)
\]

The above states that if a role ?r is attached with a port ?p and the role has a process ?x in its action, then the port must also have the same process ?x in its action set. Note that a port may have more behaviors than that of its attached role, which is allowed during an attachment. This also gives the flexibility for a port to be able to participate in different roles of connections. The ontology reasoning engine asserts the above rule when checking the correctness of the attachments in the architecture design.

After defining an ontology for the component and connector model, we will further demonstrate the construction of commonly used software architecture styles by extending this base ontology. In the following subsection, we use the Client and Server style as an example to illustrate the modeling. Other styles can be defined in a similar manner.

### 2.1. Ontology for the Client and Server style

The Client and Server structure is one of the most widely adopted styles in software architecture description, especially in distributed and network based computer systems. It offers a loosely coupled multi-connection mechanism in a consumer and provider fashion. A server is a component that provides a set of services that are exposed to various clients for consumption. A client is a component that acts as a consumer to those services. We define the following OWL classes for the Client and Server style by extending the component and connector ontology.

\[
\begin{align*}
\text{Client} & \sqsubseteq \text{Component} \\
\text{Request} & \sqsubseteq \text{Port} \\
\text{CnS} & \sqsubseteq \text{Connector} \\
\text{Provider} & \sqsubseteq \text{Role} \\
\text{SendRequest} & \sqsubseteq \text{Process} \\
\text{InvokeServer} & \sqsubseteq \text{Process}
\end{align*}
\]

The above extends the component and connector model and defines the basic entities involved in a Client and Server structure, i.e., client, server, connector, ports, roles and participating processes. Note that the Request port belongs to the Client component and the Provide port belongs to the Server component. CnS is a connector that has a Provider role and a Consumer role. We refine the basic behaviors of the style into four communication processes, i.e., SendRequest, ReceiveResult, InvokeServer and ServerReturn, where the first two belongs to the consumer and the last two belongs to the provider. These relations are specified using OWL properties as follows.
In addition, we defined a necessary and sufficient condition for the CnS connector, where there is a closure axiom on the hasRole property. This enforces the classification on any connector that only has a provider and a consumer as its roles to be a connector of the Client and Server style. Note that the necessary and sufficient conditions contribute to the automatic recognition of architecture styles through ontology classification, which will be demonstrated later.

With the structure information of the Client and Server style defined, we then use SWRL rules to capture its dynamic communication behaviors (protocol) as follows. We divided the entire communication of the Client and Server style into two parts, i.e., the communication of the connector and the internal behaviors within the server. The former consists of two action sequences - (1) receive a request from the consumer role and invoke the server through the provider role, (2) receive a server return value from the provider role and send the result back through the consumer role. This is captured by the first SWRL rule below using the hasCommunication property. The latter represents the internal actions of a server that contributes to the overall communication of the service consumption, i.e., whenever there is an invocation of request on the server, there should always be a corresponding response returned from the server. This is captured using the second SWRL rule shown below.

\[
\begin{align*}
\text{CnS}(?cn) \land \text{Consumer}(?cr) \land \text{Provider}(?pr) \land \\
p1 : \text{Id}(?id) \land \text{SendRequest}(?r) \land \text{InvokeServer}(?i) \land \\
\text{ServerReturn}(?ret) \land \text{ReceiveResult}(?res) \land \\
p1 : \text{hasRole}(?cn, ?cr) \land p1 : \text{hasRole}(?cn, ?pr) \land \\
p1 : \text{hasAction}(?cr, ?r) \land p1 : \text{hasAction}(?cr, ?res) \land \\
p1 : \text{hasAction}(?pr, ?i) \land p1 : \text{hasAction}(?pr, ?ret) \land \\
p1 : \text{hasId}(?r, ?id) \land p1 : \text{hasId}(?i, ?id) \land \\
\quad \rightarrow p1 : \text{hasCommunication}(?r, ?i) \land \\
\quad \quad p1 : \text{hasCommunication}(?ret, ?res) \\
\end{align*}
\]

\[
\begin{align*}
\text{Server}(?s) \land \text{Provide}(?p) \land \text{InvokeServer}(?i) \land \\
\text{ServerReturn}(?r) \land p1 : \text{hasPort}(?s, ?p) \land \\
p1 : \text{hasAction}(?p, ?i) \land p1 : \text{hasAction}(?p, ?r) \land \\
p1 : \text{Id}(?id) \land p1 : \text{hasId}(?i, ?id) \land p1 : \text{hasId}(?r, ?id) \land \\
\quad \rightarrow p1 : \text{hasCommunication}(?i, ?r) \\
\end{align*}
\]

After completing the ontology definition for the Client and Server style, we can invoke the Pellet reasoner to check the consistency on the ontology. In addition, we can create particular configurations of the style and infer new knowledge on the models. We will demonstrate these verification aspects with an example in the next section.

3. A Three-Tier Network Architecture

In this section, we demonstrate the modeling and verification of a Three-Tier network example by using the architecture style ontology that we defined earlier. In modern web applications, the systems are constructed using a multi-layered structure, where a middle-ware layer is introduced to encapsulate the business logics. The middle layer is named as the ‘Application Server’, which can be considered as a mediator between the clients and the data storage facilities. The application server acts as both a server to the user requests and a client to the database server for data manipulations. We model the Three-Tier architecture using the following OWL classes by extending the style ontology that we defined in the previous sections.

\[
\begin{align*}
\text{ThinClient} & \sqsubseteq \text{Component} \\
\text{ApplicationServer} & \sqsubseteq \text{Component} \\
\text{DatabaseServer} & \sqsubseteq \text{Component} \\
\text{CnA} & \sqsubseteq \text{Connector} \\
\text{AppRequest} & \sqsubseteq \text{Port} \\
\text{AppProvide} & \sqsubseteq \text{Port} \\
\text{AppConsumer} & \sqsubseteq \text{Role} \\
\text{DBRequest} & \sqsubseteq \text{Port} \\
\text{DBProvide} & \sqsubseteq \text{Port} \\
\text{SendApp} & \sqsubseteq \text{SendRequest} \\
\text{AppResult} & \sqsubseteq \text{ReceiveResult} \\
\end{align*}
\]

We define the ThinClient, ApplicationServer, DatabaseServer as basic components involved in the structure. In addition, CnA is a connector that connects the thin client with the application server, while And is the connector that relates the application server to its database server. The above also gives the definitions of the ports used in the model, i.e., AppRequest, AppProvide, DBRequest, and DBProvide. The roles used in the structure include AppConsumer, AppProvider, DBConsumer and DBProvider. Finally, we define the processes that involved in the communication of the structure, e.g., SendApp extends the SendRequest process of the Client and Server style ontology, which denotes the client sending a request to the application server. Other processes, such as AppResult extends the ReceiveResult process, AppReturn extends the ServerReturn process, and so on, can be defined in a similar manner. With the entities of the Three-Tier ontology specified, we can add property definitions to relate the classes, e.g., the client only has application request ports, the application server has both a provide port to its client and a request port to the database server, and so on. Due the space limit, we can not list all of them here. Furthermore, the communication behaviors of the Three-Tier architecture need to be redefined in terms of the specific processes used in the new structure. For example, the interactions of the application server are defined using the following SWRL rule.

\[
\begin{align*}
\text{ApplicationServer}(?as) & \land \text{AppProvide}(?ap) \land \\
\text{DBRequest}(?dr) & \land \text{AppAccess}(?laa) \land \text{AppReturn}(?lar) \land \\
\text{SendDB}(?sd) & \land \text{DBResult}(?dres) \land p1 : \text{Id}(?id) \land \\
\end{align*}
\]
The above states that ApplicationServer has two ports i.e., AppProvide and DBRequest. The AppProvide port acts as the interface for providing services to the client and has AppAccess and AppReturn as its actions, while the DBRequest port acts as the interface for requesting data access service from the database server and has SendDB and DBResult as its actions. The behavior of the application server mainly consists of two types of communication sequences, i.e., (1) \(<\text{AppAccess}, \text{SendDB}\) - invokes service from a client request via AppAccess and sends the converted data request to database server via DBRequest; (2) \(<\text{DBResult}, \text{AppReturn}\) - receives data result from the database server via DBResult and returns the converted result to the client via AppReturn. This concludes the dynamic behaviors of the application server. Similarly, the communication behaviors of other entities in the model, such as, the ThinClient and DatabaseServer components, the CnA and AnD connectors can be redefined by simply cloning the server and connector communication rules from the Client and Server style ontology.

### 3.1. Styles recognition via classification

Up to now, we have completed the ontology definition of the Three-Tier network architecture model. Careful readers might have already noticed that the entities of the Three-Tier ontology mainly extend the component and connector model, apart from its specific process definitions. This was ‘deliberately’ chosen to demonstrate the automatic style recognition via ontology classification. In Section 3.2, we mentioned that the necessary and sufficient conditions with closure axioms can contribute to the automatic recognition of different architecture styles through ontology classification. Figure 2 represents the inferred hierarchy of the classes in the Three-Tier example after running the ‘classify taxonomy’ function of the Pellet ontology reasoner. As shown in the diagram, the left-hand side panel contains the asserted (original) class hierarchy of the Three-Tier ontology by directly extending the component and connector model, the right-hand side panel contains the inferred class hierarchy after the classification. We can see that except for the ApplicationServer, all other classes that we defined in the Three-Tier ontology have been re-classified into the corresponding categories under the Client and Server ontology. For example, the ThinClient is moved to a subclass of the Client, and the DatabaseServer is now under the Server class. The CnA and AnD connectors have both been moved to subclasses of the CnS connector. Similarly, the ports and roles that we defined earlier have also been re-classified under the specific ports and roles of the Client and Server style ontology. Note that in this example, because we defined a server to be only having the ‘Provide’ ports and a client to be only having the ‘Request’ ports, the ApplicationServer in Figure 2 was not classified as either a client or a server, since it has both an AppProvide port to the thin client and a DBRequest port to the database server.

![Figure 2. Automated style recognition of the Three-Tier architecture via classification.](image)

Effectively speaking, the above indicates that the ontology classifier have automatically recognized the Three-Tier ontology to be composed of two basic Client and Server structures. We believe that this kind of automated style recognition can be very useful during software architecture modeling. The users only need to create their customized architecture models by extending the top level component and connector ontology, and let the ontology classifier to automatically recognize the styles that involved in the design. It not only helps the users to realize the architectural design patterns, but also assists the designers in decomposing a complex system into clear structured and verifiable sub-models.

### 3.2. Communication generation via inference

With the Three-Tier ontology defined, we can create specific configurations of this structure through the instance declarations. For example, we defined a network configuration with twenty thin clients, two application servers and one database server. Each application server
handles about ten clients and they both connected to the same database server. Note that such a configuration involves 294 instances in total, which consists of the instances for all the components, connectors, ports, roles, processes, and client identifiers. Once the instances are created, we attach the desired roles to their corresponding ports to form the communication topology. When ports and roles are connected, we can then invoke the Jess rule engine to automatically infer the communication sequences on the instance configuration. For example, a typical inferred communication sequence for Client₁ could be – ⟨SendApp₁, AppAccess₁⟩, ⟨AppAccess₁, SendDB₁⟩, ⟨SendDB₁, DBAccess₁⟩, ⟨DBAccess₁, DBReturn₁⟩, ⟨DBReturn₁, DBResult₁⟩, ⟨DBResult₁, AppReturn₁⟩, ⟨AppReturn₁, AppResult₁⟩. This effectively illustrates exactly what happened in the model in terms of communication interactions among the entities, i.e., starting from the client submits a request to the server until the client receives the corresponding result. The Jess rule engine generates 300 inferred axioms on the configuration model, which includes twenty separately identified communication sequences with respect to the number of clients involved in the configuration (each contains the above seven communication pairs with different client identifier labeled) to simulate the interactions among the instances of the clients and servers.

4. Conclusion

In this paper, we proposed a formal approach to software architecture modeling and verification using the semantic web technology. We represented architecture models as ontology descriptions and applied DL reasoners to perform automated verification on the design. The OWL definitions specify the structure information of the model, where the SWRL rules capture the dynamic communication in the styles. Users can easily extend the style ontology in defining their customized architecture models. In the aspect of automated verification, we demonstrated two levels of reasoning, i.e., style recognition via ontology classification and communication sequence generation using rule inference. The former works on the ontology level and can automatically recognize the style patterns used in the design, where the latter applies to the instance level and enables the users to automatically derive the interactions within the configuration of a design. As a complex architecture model usually consists of a combination of different architecture styles, these two levels of verification can be effectively used together to enhance the quality of a design. For example, when creating a customized architecture ontology model, the user only needs to define the model by extending the top level component and connector ontology and then invoke the classifier to automatically recognize the architecture styles used in the design. Once the customized architecture model is classified into recognizable styles, the users can then create specific configurations (e.g., partially) of the design (ontology) and invoke the rule engine to automatically infer new knowledge and derive the communication sequences on the model. To demonstrate the effectiveness of our approach, we illustrated the design and verification of a Three-Tier network architecture model as the case study.

In the future, we plan to develop a visual tool support that aims at assisting the design and verification of architecture models based on the proposed ontology approach. The tool should allow the designers to build their architecture models graphically without knowing the underlying syntax and knowledge of OWL/SWRL. It should integrate all the ontology definitions and verification steps described in the paper and automated them in one coherent visual interface.

References