Ontology-based Software Architectural Pattern Recognition and Reasoning

Nacha Chondamrongkul[1], Jing Sun[2] and Ian Warren[3]
Department of Computer Science, University of Auckland, New Zealand
[1]ncho604@aucklanduni.ac.nz, [2]jing.sun@auckland.ac.nz, [3]i.warren@auckland.ac.nz

Abstract— **Designing software architecture is a knowledge-intensive task that typically involves textual and diagrammatic notation. Using these kinds of notation is often inconsistent, misleading, and ambiguous. Ontology representation is, therefore, a suitable approach, as it can semantically define architectural design model that can be automatically verified through reasoning. However, a large-scale software system is usually complex and applies more than one architectural styles with various behavioral patterns. Therefore, the scalability of automated verification for a complex software architecture design is a challenge. We propose an approach that helps to formally define complex architectural design model and automate different verifications such as consistency checking, architectural styles recognition, and behavioral sequence inference. Ontology Web Language (OWL) is used to semantically define basic architectural elements and architectural styles, while a set of rules defined in Semantic Web Rule Language (SWRL) helps to capture behavioral pattern according to style. We evaluated the scalability of our approach. The result shows that different levels of complexity in architectural design model has a minor impact on the verification performance.**

I. INTRODUCTION

Software architecture is typically a conceptual design that decomposes a software system into a set of logical components. At the early phase of software development process, software architecture is designed to meet specific functional requirements, non-functional requirements and business goals. Software architecture, therefore, encapsulates set of early design decisions tradeoffs and constraints, which provide a guideline to implement software system throughout the development lifecycle. Unfortunately, software architecture is often abstract and informally presented by the combination of textual and graphical notation that are often misleading, ambiguous and inconsistent. Although, several standardized architecture description languages (ADL) have been proposed, such as ISO/IEC/42010 [1] and UML [2], they have little or no formal semantic support. Without semantic constraints, the verification of architecture design is, therefore, a daunting task. Moreover, the large-scale software system is usually a complex entity that applies predefined architectural styles, each style characterizes specific type of component and their behavior. Even though, a few ADLs, such as ACME [3], support abstraction of architecture into reusable styles but they have no semantic that enforces style constraints. ADLs has little popularity among practitioners because they are lack of tools to support, and yet require high learning curve [4]. The inadequate mechanism of producing accurate software architecture model catalyzes applying formal methods into this area.

Formal methods have played an important role in software engineering research for some time. A number of researchers have applied ontology technique to software development lifecycle [5], in order to resolve ambiguous, prevent errors and minimize cost in different phases, from requirement gathering [6] to software maintenance [7]. For software architecture, in particular, architectural design model is formally specified, in pursuance of automated verification [8]. However, the performance of automated verification in large-scale software is still an open issue for existing approach such as Alloy [9]. Although, Wong et al [10] proposed a solution that allows model to be decomposed, in order to parallelize verification process. However, dependencies between components still require verification process to be executed in sequential manner. The ontology has been proposed to apply in designing software architecture because of its strength in effective large-scale reasoning that can automate consistency checking and hierarchy inference in the design model [11, 12, 13]. Pahl et al. [14] integrated ontology into ACME, in order to verify consistency of an architecture and its behaviors, but process modeling notation is still a limitation. In pursuance of consistency checking automation, style recognition, and communication inference, Sun et al. [15] proposed to use Ontology Web Language (OWL) [16] to formally specify different entities and relationships in architectural design. The communication flow can be captured by rules based on Semantic Web Rule Language (SWRL) [17]. However, the performance of automated verification was not evaluated, and the range of provided architectural styles is limited.

The main challenge is how we can semantically define complex architectural design based on multiple styles, and evaluate how the proposed method impacts to the automated verification performance. We propose an approach as shown in Figure 1. The ontology library includes basic architectural element, architectural styles, and behavioral rules. OWL is used to define basic architectural elements, such as component and connector. These basic architectural elements can be extended to define various architectural styles, and a design instance that represents the architectural design model of a specific software system. As the ontology for architecture design is inevitable complex, we use description logic (DL) based languages, in order to take advantage of existing DL reasoning engine that is

DOI Reference Number: 10.18293/SEKE2018-012

effective in performing large-scale automated reasoning. The consistency in architectural styles and design instance can be automatically checked by the reasoning engine, based on the ontology's constraints defined in basic architectural elements. In order to recognize architectural styles applied in the design, reasoning engine classifies architectural elements into different ontological classes specific to architectural style. After styles are recognized, reasoning engine processes behavioral rules to capture architectural configuration and generate behavioral sequences, which manifest interactions between components.

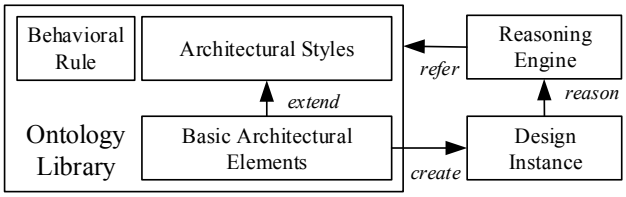

Figure 1 Overview of architectural design approach

The rest of the paper is organized as follows. We present ontology-based architectural styles in Section 2. Section 3 illustrates how a software architecture can be modeled and verified against architectural styles. This paper concludes in Section 4 with future research direction.

II. Ontology Modeling for Architecture Styles

The ontology library is based on Component & Connector (C&C) [18] view, which aims to exhibit how the system works at runtime. Software architects use C&C view for reasoning about key system quality attributes such as performance, security, and reliability [19]. In C&C, a component represents a processing unit within the software system, while connectors define interaction mechanism between components. The component has a set of ports that serves communication to another component, whereas connector has a number of roles, each has specified set of actions it specifically performs. A component can be associated with a connector by attaching its port to a connector's role. Based on C&C view's concept, our ontology library has consisted of ontological classes representing fundamental architecture elements, namely *Component*, *Connector*, *Port*, *Role,* and *Action*. *hasAttachement* is defined as an object property to associate component's port with connector's role. *Action* is assigned to *Role* via *hasAction* property with minimum cardinality restriction, in order to make sure that a role has at least one assigned action. Below are ontology classes expressed in description logic syntax.

$ArchElement \sqsubseteq Component \sqcup Connector \sqcup Interface \sqcup Action$
$Interface \sqsubseteq Port \sqcup Role$
$Component \sqcap Connector \sqcap Interface \sqcap Action \equiv \bot$
$Port \sqcap Role \equiv \bot$
$Component \equiv ArchElement \sqcap \exists hasPort\ Port$
$Connector \equiv ArchElement \sqcap \exists hasRole\ Role$
$Port \equiv Interface \sqcap \exists hasAttachment\ Role$
$Role \equiv Interface \sqcap \geq 1\ hasAction\ Action$

C&C can be characterized by various architectural styles. Each style specifies a particular set of component, connector, and behavioral pattern. The following are some architectural styles included in our ontology library.

A. Client-Server Style

The client and server are two key component types in this style. *Cns:Client* and *Cns:Server* are defined as classes, extended from *Component*. *Request* and *Response* are port type attached to client and server respectively. The port attachment is defined by *hasPort* property with existential restriction, as a component can be classified as several component types.

$CnS{:}Client \equiv Component \sqcap \exists hasPort\ Request$
$CnS{:}Server \equiv Component \sqcap \exists hasPort\ Response$
$Request \equiv Port \sqcap \exists hasAttachment\ Consumer$
$Response \equiv Port \sqcap \exists hasAttachment\ Provider$

A corresponding connectivity is defined as a connector class to incorporate two roles, *Consumer*, and *Provider*. *Consumer* requests services on the server, while *Provider* performs actions to process the request and return the result back to *Consumer*. *SendRequest*, *ReceiveResult*, *ServerInvoked* and *ReturnResult* are defined as subclasses of *Action*, in order to represent different activities and events in client-server style.

$CnSConnector \equiv Connector \sqcap \exists hasRole\ Consumer \sqcap \exists hasRole\ Provider$

$CnS{:}Consumer \equiv Role \sqcap \exists hasAction\ SendRequest \sqcap \exists hasAction\ ReceiveResult$
$CnS{:}Provider \equiv Role \sqcap \exists hasAction\ ServerInvoked \sqcap \exists hasAction\ ReturnResult$

$SendRequest \sqsubseteq Action$ $\quad$ $ServerInvoked \sqsubseteq Action$
$ReceiveResult \sqsubseteq Action$ $\quad$ $ReturnResult \sqsubseteq Action$

The behavioral rules are defined to associate relevant actions as a sequence according to the style's behavioral pattern. In order to generate a behavioral sequence, the *hasNextAction* property is used to define what action comes next in the sequence. Below is a rule defined in SWRL, it captures behavioral pattern as follows. At first, when the client sends a request, the server will be invoked. This rule hence implies *ServerInvoked* as the next action to *SendRequest*. After the server finishes processing and returns the result, the client will receive the result. This rule, therefore, implies *ReceiveResult* as the next action of *ServerReturn.*

$CnSConnector(?cns) \sqcap hasRole(?cns, ?cr) \sqcap CnS{:}Server(?server) \sqcap isPortOf(?p, ?server), SendRequest(?sreq) \sqcap hasAction(?cr, ?sreq) \sqcap Provider(?pr), isAttachmentOf(?pr, ?p) \sqcap, Consumer(?cr) \sqcap ReceiveResult(?rres) \sqcap hasRole(?cns, ?pr) \sqcap hasAction(?pr, ?invs) \sqcap ServerInvoked(?invs), ServerReturn(?sret), hasAction(?pr, ?sret) \sqcap hasAction(?cr, ?rres)$
$\rightarrow hasNextAction(?sreq, ?invs) \sqcap hasNextAction(?sret, ?rres)$

Below is another rule that captures an occurrence when the server is invoked to process the request. After that, the result will be returned to the client. This rule hence implies *ServerReturn* as the next action of *ServerInvoked*

$Response(?p) \sqcap isPortOf(?p, ?server) \sqcap hasAction(?p, ?iser) \sqcap Provider(?pr) \sqcap isAttachmentOf(?pr, ?p) \sqcap CnSConnector(?cns) \sqcap CnS{:}Server(?s) \sqcap Consumer(?cr) \sqcap hasRole(?cns, ?pr) \sqcap Request(?r) \sqcap hasAction(?p, ?sret) \sqcap ServerInvoked(?iser) \sqcap$

ServerReturn(?sret), isAttachmentOf(?cr, ?r) ⊓ hasRole(?cns, ?cr) → hasNextAction(?iser, ?sret)

B. N-Tier Style

A number of clients and servers can form a multi-level hierarchy, a tier has consisted of clients that invoke servers on the upper tier. Each tier runs on the separate physical environment so it can be maintained independently of other tiers, however, interaction between tiers rely on each other. For example, the business application typically has 3 tiers namely client, business logic, and data management. A request to service on business logic consequently triggers a request to data management tier. To semantically define this, connector's reliance is defined by *hasLink* property, so a class for tier can be formally expressed as follows:

NTier:Tier≡ Component ⊓ ∃ hasPort
(Port ⊓ ∃ hasAttachment
(Role ⊓ ∃ isRoleOf
(Connector⊓ ∃ hasLink Connector)))

Below is a rule defined to capture behavioral pattern between tiers. When a server on a tier is requested and invoked, it may make a request to the upper tier. When the result is received, it is forwarded to the lower tier. The actions between tiers are related by *hasDivertNextAction* property.

CnSConnector(?cns) ⊓ hasRole(?cns, ?pr) ⊓ hasRole(?cns, ?cr) ⊓ Provider(?pr) ⊓ Consumer(?cr) ⊓ hasAction(?pr, ?invs1) ⊓ hasAction(?pr, ?sret1) ⊓ ServerInvoked(?invs1) ⊓ ServerReturn(?sret1) ⊓ hasLink(?cns, ?cns2) ⊓ CnSConnector(?cns2) ⊓ hasRole(?cns2, ?pr2) ⊓ hasRole(?cns2, ?cr2) ⊓ Provider(?pr2) ⊓ Consumer(?cr2) ⊓ hasAction(?cr2, ?rret2) ⊓ hasAction(?cr2, ?sreq2) ⊓ ReceiveResult(?rret2) ⊓ SendRequest(?sreq2)
→ hasDivertNextAction(?invs1, ?sreq2) ⊓
hasDivertNextAction(?rret2, ?sret1)

C. Publish-Subscribe Style

This style has components interacting to each other through events. *Pns:Publisher* is a subclass of *Component* for publisher, a component type that announces events to subscribed component, while *Pns:Subscriber* is a subclass for subscriber, a component type that listens to the events. *Announce* and *Register* are ports for publisher and subscriber respectively. The defined classes for component types and its ports type can be formally expressed as follows:

PnS:Pulisher≡ Component ⊓ ∃ hasPort Announce
PnS:Subscriber≡ Component ⊓ ∃ hasPort Register
Announce≡ Port ⊓ ∃ hasAttachment Publisher
Register≡ Port ⊓ ∃ hasAttachment Subscriber

Publisher and *Subscriber* are defined as role class in this style. The connector is an event bus that coordinates these two roles.

PnSConnector≡ Connector ⊓ ∃ hasRole Publisher
⊓ ∃ hasRole Subscriber
Publisher ≡ Role ⊓ ∃ hasAction SubscribeToEvent
⊓∃ hasAction EventAnnouced
⊓∃ hasAction DeliverEvent
Subscriber ≡ Role ⊓ ∃ hasAction ReceiveEvent
⊓∃ hasAction RequestSubscription

RequestSubscription ⊑ Action *EventAnnounced ⊑ Action*
SubscribeToEvent ⊑ Action *DeliverEvent⊑ Action*
ReceiveEvent ⊑ Action

The behavioral rule for publish-subscribe style is shown below. This rule captures two occurrences in this style: 1) Subscription: If a subscription is requested by a component, the publisher will acknowledge and subscribe requesting component to an event. Therefore, this rule implies *SubscribeToEvent* to be the next action of *RequestSubscription.* 2) Event Publishing: When a publisher announces an event, the event will be delivered to all subscriber. This rule below infers the sequence of actions as *EventAnnounced*, *DeliverEvent* and *ReceiveEvent* respectively. This sequence is sorted through last two *hasNextAction* property assertions in the rule's implication.

PnSConnector(?cns) ⊓ Publisher(?p) ⊓ Subscriber(?s) ⊓ hasAction(?p, ?feven) ⊓ hasAction(?p, ?seven) ⊓ hasAction(?p, ?neven) ⊓ FireEvent(?feven) ⊓ NewEventOccur(?neven) ⊓ SubscribeToEvent(?seven) ⊓ hasAction(?s, ?reqs) ⊓ hasAction(?s, ?reven) ⊓ RequestSubscription(?reqs) ⊓ ReceiveEvent(?reven)
→ hasNextAction(?reqs, ?seven) ⊓
hasNextAction(?neven, ?feven) ⊓ hasNextAction(?feven, ?reven)

D. Repository Style

The repository style organizes how data is accessed and stored in software system through centralized repositories. Data repository and data accessor are two major component types in this style. Data repository (*RP:DataRepository*) persists data, manages concurrent access, and supports access control. Data accessor (*RP:DataAccessor)* reads and writes data at one or more repositories.

RP:DataRepository≡ Component ⊓ ∃ hasPort
(Port ∃ hasAttachment Store)
RP:DataAccessor≡ Component ⊓ ∃ hasPort
(Port ∀ hasAttachment (Reader ∪Writer))

We create two connector classes corresponding to writing and reading function in this style. Both connectors associate *Store* role to address where the data persists. *Writer* role identifies the component that requests to write data on the repository, whereas *Reader* role identifies the component that requests to read data on the repository.

DataReadConnector≡ Connector ⊓ ∃ hasRole Store
⊓ ∃ hasRole Reader
DataWriteConnector≡ Connector ⊓ ∃ hasRole Store
⊓ ∃ hasRole Writer
Store≡ Role ⊓∀ hasAction (ReadData ∪WriteData)
Writer≡ Role ⊓∃ hasAction RequestWrite
Reader≡ Role ⊓∃ hasAction RequestRead
ReadData⊑Action *RequestRead ⊑Action*
WriteData⊑Action *RequestWrite⊑Action*

The behavioral sequence is captured by two rules below. The first rule support reading function so it implies *RequestRead* as precedence action to *ReadData*, likewise, the second rule implies *RequestWrite* as precedence action to *WriteData* action.

DataReadConnector(?con)) ⊓ RequestRead(?reqr)) ⊓
hasAction(?store, ?read)) ⊓ ReadData(?read)) ⊓

hasAction(?reader, ?reqr)) ⊓ hasRole(?con, ?reader)) ⊓ hasRole(?con, ?store)) ⊓ Store(?Store)) ⊓ Reader(?reader) → hasNextAction(?reqr, ?read)

DataWriteConnector(?con) ⊓ Writer(?writer) ⊓ RequestWrite(?reqw) ⊓ hasAction(?writer, ?reqw) ⊓ hasRole(?con, ?store) ⊓ Store(?store) ⊓ WriteData(?write), hasRole(?con, ?writer) ⊓ hasAction(?store, ?write) → hasNextAction(?reqw, ?write)

III. CASE STUDY & EVALUATION

The online shopping application system is used as a case study to demonstrate our approach. This case study is a sample of complex software system that applies multiple architectural styles. Figure 2 shows its software architecture design that has consisted of four components namely, *TransactionLog*, *PaymentGateway*, *Shopping Mobile App* and *ShopService*. The ports are depicted as small box attached to the components such as *LoggingRequest,* and *PayResponse*. Shopping mobile app has user interfaces that allow the user to purchase the products and make a payment through payment gateway. When a payment is submitted to the payment gateway, a transaction will be recorded by transaction logger. If the user subscribes to price alert service, the notification will be sent when price is updated.

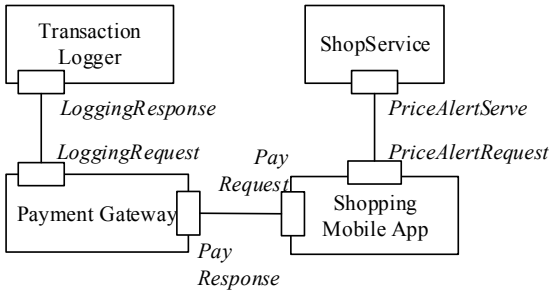

Figure 2 Software architecture design for shopping application

We create ontology instance representing an architectural design model for our case study. The design elements namely actions, roles, port, connectors, and components, are created as individuals that are instances of classes defined in the ontology library. The object properties are used to relate these individuals together, in order to establish a structure in the design model. Due to page limit, we can not show all individuals definition in this paper. The compelete definition for this case study can be found at *https://goo.gl/4ugkLB*

The individuals are created for actions with one or more types specified, and they can be formally expressed in OWL abstract syntax as follows:

Individual(ex: ActRequestToPay type(ex: SendRequest))
Individual(ex: ActRequestToLog type(ex: SendRequest) type(ex: RequestWrite))
Individual(ex: ActLogTransaction type(ex: ServerInvoked) type(ex: WriteData))

The roles are defined as individuals with *hasAction* property to the action individuals defined previously. Below are some individuals defined for roles.

Individual(ex:PaymentProvider
value(ex: hasAction ex:ActProcessPayment)
value(ex: hasAction ex:ActReturnPayResult))

Individual(ex: PaymentRequester
value(ex: hasAction ex: ActRequestToPay)
value(ex: hasAction ex: ActReceivePayResult))

The following are sample individual defined for port. These individuals have one or more relationship to the role individuals through *hasAttachment* property.

Individual(ex: PayRequest
value(ex: hasAttahment ex: PaymentRequester))
Individual(ex: PayResponse
value(ex: hasAttahment ex: PaymentProvider))

A number of individuals are created corresponding to communication lines shown in Figure 2. The roles individual are assigned to each connector through *hasRole* property.

Individual(ex: PaymentService
value(ex: hasRole ex: PaymentProvider)
value(ex: hasRole ex: PaymentRequester)
value(ex: hasLink ex: LoggingService))
Individual(ex: LoggingService
value(ex: hasRole ex: LoggingProvider)
isolate *value(ex: hasRole ex: LoggingRequester))*
Individual(ex: NotificationService
value(ex: hasRole ex: NotificationPublisher)
value(ex: hasRole ex: NotificationSubscriber))

Each of the components has the corresponding individual created, each component individual is attached with one or more port individuals through *hasPort* properties.

Individual(ex: PaymentGateway
value(ex: hasPort ex: PayResponse)
value(ex: hasPort ex: LoggingRequest))
Individual(ex: ShoppingMobileApp
value(ex: hasPort ex: PayRequest)
value(ex: hasPort ex: PriceAlertRequest))
Individual(ex: ShopService
value(ex: hasPort ex: NotificationPublisher))
Individual(ex: TransactionLogger
value(ex: hasPort ex: LoggingProvider))

As mention previously, the architectural design is defined based on OWL/SWRL, in order to take advantage of classification performed by reasoning engine. The classification results in automating architectural consitency checking, architectural style recognition, and behavioral sequence generation.

A. Architectural Consistency Checking

The architectural consistency checking relies on the ontology classification process that verifies consistency in the ontology model and computes hierarchies of defined classes. Figure 3 (a) shows inferred hierarchy of the ontology library when it is consistent. If the classes are consistent, they will be classified into subclasses of basic architectural elements such as component, connector, port, and interface. The inconsistency may be caused by a number of reasons. For example, incompatible classes are associated with domain or range of an object property, or a class has two parents that are disjoint classes. Figure 3 (b) shows a scenario when the ontology library is inconsistent. In this scenario, a class definition for tier

(*NTier:Tier*) contains an axiom ∃ *hasLink Component*, which violates *hasLink* property's constraints that requires *Connector* class as a range. *NTier:Tier* class is, therefore, inconsistent, and it is denoted as a subclass of *owl:Nothing*.

(a) Consistent ontology *(b) Inconsistent ontology*

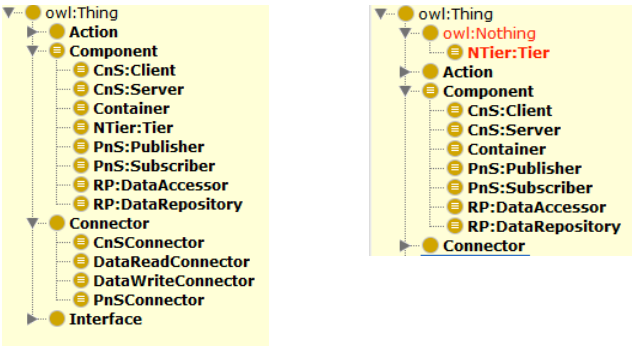

Figure 3 Inferred Hierarchy

When a set of individuals are defined for an architectural design model, careful reader may notice that action is the only design element defined as individual with explicit types. If the archtectural design model is consitent, the type of individuals, representing role, port, component and connector, is transitively inferred based solely on their relationships. The architectural styles can be recognized along with inferred types. For example, *Cns:Consumer* in client-server style is a role that has actions namely *ReceiveResult* and *SendRequest*. *PaymentRequester* is thus inferred as an instance of *Cns:Consumer* role, because it has relations to two action individuals namely *ActReceiveResult* and *ActRequestToPay,* which are instances of *ReceiveResult* and *SendRequest* respectively. According to ports class definition, *Request* is a port with some attachment to consumer role. *PayRequest* is thus inferred as an instance of *Request* port, because *PayRequest* is attached to *PaymentRequester* as shown in Figure 4. As defined in *Cns:Client* class, a component is client, if it has some *Request* port. Therefore, *ShoppingMobileApp* is inferred as an instance of *Cns:Client* due to its relation to *PayRequest* port, as shown in Figure 5.

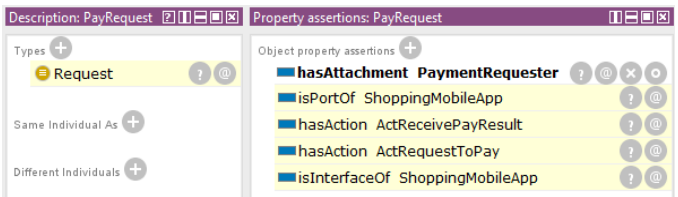

Figure 4 PayRequest Port

B. Composite Architectural Style Recognition

When more than one architectural styles are applied to the design model, the classification can also identify a composite architectural styled component. The composite architectural styled component is a component that is an instance of several classes from not only the same style but also different styles. As shown in Figure 5, *ShoppingMobileApp* is not only a client but also a tier (*NTier:Tier*) in the multi-tier style. Because *ShoppingMobileApp* transitively relates to *PaymentService* connector, which has a link to *LoggingService* connector. According to *NTier:Tier* class definition, a tier is a component that transitively relates to a connector, which has a link to another connector. For the same reason, *PaymentGateway* and *TransactionLogger* are also denoted as instances of *NTier:Tier*. Also, *ShoppingMobileApp* has a *Subscribe* port namely *PriceAlertRequest*, it is hence a subscriber in publish-subscribe styles too.

Figure 5 ShoppingMobileApp Component

C. Behavioral Sequence Generation

After the reasoning engine identifies the type of individuals and the architectural styles are recognized, the reasoning engine will automatically capture the sequence of behavioral activities based on the behavioral rules specific to style. Figure 6 depicts the payment sequence in the online shopping application system. The behavioral rules logically imply *hasNextAction* and *hasDivertNextAction* properties to the action individuals, in order to connect series of action individuals as a sequence. The behavioral rule of client-server style implies *ActProcessPayment* as the next action of *ActRequestToPay*. As *ActProcessPayment* is also involved in N-tier style, it has thus value of *hasDivertNextAction* property as *ActRequestToLog,* implied by the behavioral rule of N-Tier style. According to the behavioral pattern of N-Tier style, when a payment is requested to *PaymentGateway*, *PaymentGateway* will process the request and call *TransactionLogger* in the upper tier to log a transaction. The behavioral rule of client-server style also implies *ActReturnPayResult* as the next action of *ActProcessPayment,* in case the payment result is returned without logging transaction (for example, when an error occurs during processing a payment). Other behavioral sequence, such as price alert, can also be generated in the same way using the behavioral rule for publish-subscribe style.

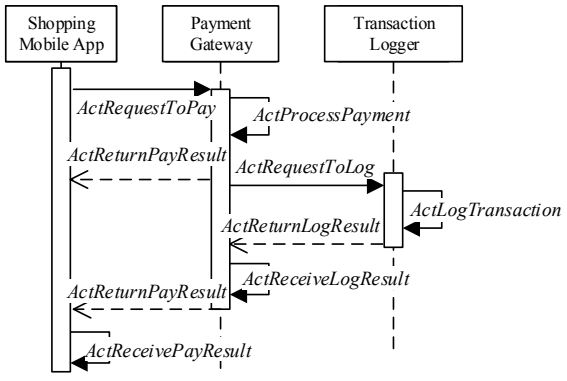

Figure 6 Part of payment process

D. Performance Evaluation

We evaluated performance of reasoning process. This evaluation focuses on measuring two parameters that impact the performance of automated verification: 1) number of architectural style applied to software design, and 2) software size that can be reflected by the number of axioms. The more axioms the ontology has, the larger scale a software is. We ran regression testing 50 times on four ontologies that have different parameter values as follows, A contains 0 styles with 144 axioms, B contains 2 styles with 216 axioms, C contains 3 styles with 246 axioms, and D contains 4 styles with 276 axioms.

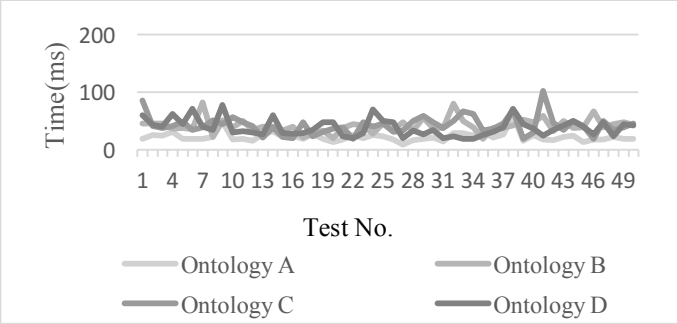

Figure 7 Result of performance testing

This evaluation was carried out using an Intel Core i7-7500U CPU @ 2.7GHz with 8.00 GB Ram computer, and we used HermiT as the reasoning engine. The time taken to reason ontologies are shown as a graph in Figure 7. The horizontal axis represents the number of time we run reasoning process. According to the test result, average time spending on reasoning is between 20-60 milliseconds and shows insignificant variation between test ontologies. Therefore, we can conclude that our approach supports scalability for complex software architectural design, as the number of applied styles and software size has minor impact on the reasoning performance.

IV. Conclusion

An architectural design model for a complex software system can be formally specified and verified with our approach. The ontology library includes extensible architectural elements that are defined semantically by OWL, whereas SWRL rules are used to capture dynamic behavior within the design. We demonstrate our approach by creating an ontology instance for an architectural design model. The reasoning engine performs classification that automates verification as follows: 1) architectural consistency is checked against constraints in the ontology, 2) architectural elements and styles are recognized, 3) behavioral sequences are automatically generated according to rules specific to architectural style. We found that complexity level in architectural design has minor impact on the automated verification performance. With automated verification, the user can concentrate on determining whether the design meets requirements, which are the most significant aspect of the software architecture design.

This paper only takes a small step toward our ultimate goal, which we aim to prevent architectural design erosion and lower maintenance cost. We plan to achieve this by extending proposed approach in this paper and integrate it to the software evolution cycle.

References

[1] R. Hilliard, "ISO/IEC/IEEE 42010," [Online]. Available: http://www.iso-architecture.org/42010/.

[2] J. Rumbaugh, I. Jacobson and G. Booch, Unified Modeling Language Reference Manual, The (2nd Edition), Pearson Higher Education, 2004.

[3] D. Garlan, R. T. Monroe and D. Wile, "Acme: Architectural Description of Component-Based Systems," in *Foundations of Component-Based Systems*, Cambridge University Press, 2000, pp. 47-68.

[4] M. Ozkaya, "Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling?," *Information and Software Technology,* vol. 95, pp. 15-33, 2018.

[5] L. Kaur and AshutoshMishra, "Software component and the semantic Web: An in-depth content analysis and integration history," *Journal of Systems and Software,* no. 125, pp. 152-169, 2017.

[6] H. Kaiya and M. Saeki, "Using Domain Ontology as Domain Knowledge for Requirements Elicitation," in *14th IEEE International Requirements Engineering Conference (RE'06)*, 2006.

[7] H.-H. Song and Z.-X. Zhang, "Study on Approach of Software Maintenance Based on Ontology Evolution," in *International Conference on Computer Science and Software Engineering*, 2008.

[8] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 6, no. 3, pp. 213-249 , 1997.

[9] J. S. Kim and D. Garlan, "Analyzing Architectural Styles with Alloy," in *Workshop on the Role of Software Architecture for Testing and Analysis 2006 (ROSATEA06)*, Portland, 2006.

[10] S. Wong, J. Sun, I. Warren and J. Sun, "A Scalable Approach to Multi-style Architectural Modeling and Verification," in *13th IEEE International Conference on Engineering of Complex Computer Systems (iceccs)*, 2008.

[11] S. Schröder and M. Riebisch, "Architecture Conformance checking with Description Logic," in *The 11th European Conference on Software Architecture*, 2017.

[12] J. Simmonds and M. C. Bastarrica, "Description Logics for Consistency Checking of Architectural Features in UML 2.0 Models," 2004.

[13] E. Yuan, "Towards Ontology-Based Software Architecture Representations," in *IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 2017.

[14] C. Pahl, S. Giesecke and W. Hasselbring, "Ontology-based modelling of architectural styles," *Information and Software Technology,* vol. 51, no. 12, pp. 1739-1749, 2009.

[15] J. Sun, H. H. Wang and T. Hu, "Design Software Architecture Models using Ontology," in *International Conference on Software Engineering and Knowledge Engineering (SEKE)*, 2011.

[16] I. Horrocks, P. F.Patel-Schneider and F. Harmelen, "From SHIQ and RDF to OWL: the making of a Web Ontology Language," *Web Semantics: Science, Services and Agents on the World Wide Web,* vol. 1, no. 1, pp. 7-26, 2003.

[17] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof and M. Dean, "SWRL: A Semantic Web Rule Language," 2004. [Online]. Available: https://www.w3.org/Submission/SWRL/.

[18] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford, Documenting Software Architectures: Views and Beyond (2nd Edition), Pearson Education, 2011.

[19] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice (2nd Edition), Addison-Wesley, 2003.