Modeling and Verifying Web Service Behaviors Based on Live Sequence Chart Specifications

Honghao GAO, Yuanyuan ZHANG†, Jun LIU

School of Computer Engineering and Science, Shanghai University, Shanghai 200072, China
College of Information Technology, Zhejiang Chinese Medical University, Hangzhou 310053, China

Abstract

Web Services have been widely used in Service-Oriented Architecture (SOA) framework. Due to the complexity of interactive behaviors, formal verification plays a critical role in Web services-based application engineering. In this paper, we mainly use Live Sequence Chart Specifications (LSC) to specify the complex behaviors among multiple Web services, and then translate LSC to automata model ELTS (Extended Labeled Transition System) for formal verification with model checking technique. To check the behavioral correctness, we propose a projection approach to check the consistency of functional requirements against ELTS model. After that, we express temporal logic properties into CTL formulae. Ultimately, model checker NuSMV (A new Symbolic Model Checker) is employed to automatically perform the verification process. After validating an abnormal behavior, we can modify the LSC specification according to the counterexample reported by NuSMV verifier. In conclusion, our approach affords an underlying guideline for guaranteeing the correctness of Web services composition.

Keywords: Web Services; Requirements Specifications; LSC; Behavioral Correctness; Verification.

1. Introduction

The process of combining Web services (WS) to provide value-added services has received much interest form e-commerce researchers, which aims to support the business services integration in the distributed Internet environment. Presently, many XML-based standards and protocols are employed to overcome platform and language dependence issues, such as BPEL, OWL-S, ebXML, and WS-CDL. However, due to Web services’ interoperability, perhaps developed at different platforms, published by different people, and possibly with different designs and implementations, it is more realistic to specify and verify Web service behaviors in development phase before deploying for business applications.

Traditionally, the verification is to check the structural correctness. For each operation of a WS composition’s “requires” interface there is at least one identically Web service whose “provides” interface matches with respect to number, sequence, and types of parameters. The technique of structural correctness focuses on interface reasoning. Once an assumption is made at the initial state’s input interface, structural correctness is to assert the guarantee of the final state’s output after invoking a Web service. But they are not help in understanding system and software behaviors clearly because it lacks of temporal logic

† Corresponding author.

Email addresses: zyy@zjtcmed.net (Yuanyuan ZHANG).
verification, especially in Web services composition.

To check the correctness of Web services applications, more and more researches have focused on studying the interactive behaviors, such as Petri-Net[1,2], Automata[3,4] or a Process Algebra[5,6], to formalize service behaviors and then utilize its theories and assistant tools to accomplish automatic checking and verification. For example, X. Fu et al. [7] presented formal semantics of Web services based on translating a BPEL to Finite State Processes (FSP). His work focuses on message sequence charts and the core of verification mechanism is to check trace equivalence, but the temporal logic (TL) property is not considered. T. Gardner [8] describes a UML profile for automated business processes which allows BPEL4WS processes to be modeled using an existing UML tool. But the UML model is not fit to formal verification because UML is not a complete formal language. Despite significant efforts, however, the correctness remains a major challenge, which is the key to ensure the reliability of Web services. The primary reason is hard to check whether or not the behavior of two services is compatible because of its additional complexity of verification, such as deadlock property and livelock property.

In our paper, we use the graphical languages LSC to describe the specifications of Web service behaviors, including the atomic (composite) Web service. Although a variety of model checking-based approaches has been proposed for Web service verification [9], differently, our approach focus on formally verifying Web service behaviors based on LSC specifications. Particularly, we consider the design phase and propose a method for checking the correctness of the UML design. Due to the informal semantics of LSC, we translate the LSC to the extended LTSs model (ELTS). To check the correctness of Web services, a projection approach is used to check the consistency, and then the CTL formulae are introduced to express properties. Finally, model checker NuSMV is used to perform the automated verification.

The remainder of this paper is organized as follows. Section 2 introduces LSC to describe the behaviors of Web services. Section 3 proposes an algorithm to translate the LSC to ELTS, and discusses the way of checking the correctness. The final section gives the conclusions.

2. LSC-based Behaviors Description for Web Services

Live Sequence Charts (LSCs for short) has become the mostly used and scenario-based language, which expresses the communication behavior requirements of multiple processes/agents that interact with each other. As basic features, the solid and dashed elements in LSC represent the mandatory and optional behaviors, respectively. There are two scenarios for specifying the requirements involving existential charts and universal charts. In existential charts, the behaviors drawn with a dashed rectangle may happen in the system. In universal charts, the behaviors drawn with a solid rectangle must exhibit at least once when the Prechart is satisfied. The other concepts, such as Processes, Cut and Temperatures, are referred to the literature [10,11].

As the number of available Web services increases, there is a growing demand to realize the complex business processes by combining and reusing the available basic Web services on Internet. Normally, the behaviors are interactions among the WS composition and its component Web services, such as the message exchange sequence. We use LSC to describe these complex behaviors. By adopting LSC, we can describe not only Web services composition behaviors, but also the uncertainty including mandatory and optional behaviors.
To describe the communication behaviors of Web service, the general LSC is slightly improved as follows: each instance line uses locations to track the state of the associated interactions. We mark each location by digital number. Given an instance line \( i \) and the max cursor \( \text{max} \), the set of locations is given by \( \text{Loc}(i) = \{ l_0, l_1, ..., l_{\text{max}} \} \). And all of the locations in a chart are denoted as \( \text{Loc} = \{ \text{Loc}(0), \text{Loc}(1), ..., \text{Loc}(n) \} \). The set of request (or response) messages is denoted as \( E \) where each message is extended by simulating the inputting messages (denoted as \( + \)), outputting messages (denoted as \( - \)) and the shared messages (denoted as \( \emptyset \)) for Web service interface operations. The interactive communication is specified as a triple \( (l, e(x), l') \), where \( l \in \text{Loc}, l' \in \text{Loc}, e \in E \) and \( x \in \{ +, -, \emptyset \} \). The set of all message communications for a chart is given by the relation \( R = \bigcup_{1 \leq i \leq n} \{ (l_i, e(x), l'_i) \mid l_i \in \text{Loc} \land l'_i \in \text{Loc} \land e \in E \land x \in \{ +, -, \emptyset \} \} \). Additionally, the conditions in LSC are described as \( G = \{ \text{Cond}(0), \text{Cond}(1), ..., \text{Cond}(n) \} \) which guard the messages communication. We employ \( \text{Man} \) and \( \text{Opt} \) to represent the mandatory and optional behavior, respectively.

In Fig.1, the example of On-line Car Sales System (OCSS) is depicted, where five Web service instances are involved: Customer (C), Vendor (V), Payment (P), Order (O), and Logistics (L). The example describes a process in which the customer interacts with the sales vendor for purchasing a car. After processing the received purchase order, the vendor sends invoice information to payment service. Once the order is confirmed, the logistics department starts to handle the delivery management. User can reject the Goods directly if the delivered car is not satisfied. Otherwise, user receives the car and ends the business after noticing the bank to pay the money in advance to vendor. To illustrate the related concepts, we give an example. For instance, The Vendor (V) is defined as follows:

\[
\text{Loc}(\text{Vendor}) = \{ C.1, C.2, C.3, C.4, C.5 \}; R(\text{Vendor}) = \{ (C.1, \text{SelectAccessories}(+), C.2), (C.2, \text{SelectColor}(+), C.3), (C.3, \text{VirtualDrive}(+), C.4) \}; E(\text{Vendor}) = \{ \text{SelectCarType}, \text{SelectAccessories}(+), \text{SelectColor}(+), \text{VirtualDrive}(+), \text{Payment} \}; G(\text{Vendor}) = \{ \text{Color}(+), \text{Test}(+) \}; \text{Man}(\text{Vendor}) = \{ \text{SelectAccessories}(+), \text{Color} (+), \text{SelectColor}(+) \}; \text{Opt}(\text{Vendor}) = \{ \text{Test}(+), \text{VirtualDrive}(+), \text{Payment} \}.
\]

![Fig.1 On-line Car Sales System](image)

3. **Translate LSC to Automata ELTS**

The graphical specification trends to be checked manually, which is not easy implemented, error-prone and informal. Extracting the formal specifications from LSC becomes an important role in formal modeling,
testing and verifying Web services. We consider translating LSC specification to automata structure. Past research in the area of transforming LSCs to automaton has primarily revolved around the generation of positive automaton for detecting chart completions. While, in our study, we use ELTS to formally model the behavior of Web services interactions.

**Definition 1 (ELTS).** An ELTS is a quintuple $M = (S, s_0, F, \Sigma, T, \lambda)$ where $S$ is a finite non-empty set of states, $s_0 \in S$ is an initial state, $\Sigma$ is a finite set of actions where we introduce $\tau$ to denote the empty action , $F$ is a finite set of final states, $\lambda = \{+, -, \emptyset\}$ is the set of operation symbols identifying the inputting and outputting events, and $T \subseteq S \times \Sigma \times \lambda \times S$ is a certain transition relation set.

Generally, ELTS is a labeled directed graph in which nodes represent steps of WS execution and edges represent the interaction flow among different steps. A path trace of an ELTS $M$ is a finite alternating sequence of states and actions $<s_1, a_1, s_2, \ldots , a_m, s_{m+1}>$ such that $s_1 \in s_0$ and for each $i \in [1, m]$, $s_i \in S$, $a_i \in \Sigma$, $\gamma_i \in \lambda$, and $(s_i, a_i, \gamma_i, s_{i+1}) \in T$. We denote by $Path(M)$ the set of all paths of $M$. Each path is considered as an implemented function of Web service.

In Table.1, The unwinding algorithm LSC2ELTS, which recursively invokes the method $GenELTS(i)$ to generate the behaviors of the each Web service instance, returns its corresponding ELTS model from LSC. The algorithm is to enumerate the locations of LSC for further processing and added to the state set $S$. After that, each message and guard condition is distinguished. The set of actions $\Sigma$ between state transitions is identified according to the relation $R$. Finally, to handle the mandatory and optional behaviors, the method $TranFromManOpt(StaA)$ is used to capture the empty action $\tau$ for the state transition.

Using LSC2ELTS algorithm can get the formalized automata for Web service. For example, Fig.2 illustrates two examples. In Customer automata, the states C.1,C.2,C.3,C.4, C.5, C.6, C.7, C.8, C.9 are extracted from LOC($Customer$) in LSC, the other states V.1,O.1,P.2,L.2,L.3,L.5,L.7 are the interactive locations which belong to other instance lines of Web service. For an instance line of LSC, the empty action $\tau$ is employed to describe the transition between two neighbor locations when their evolution is executed without any message exchange.

However, some dynamic e-business solutions or large business require collaborative works to accomplish the complex tasks. Thus, the ELTS model should include more complex behaviors. In the case, we consider the compatibility, and introduce an automata-based layering graph method to address the issues of Web services composition, which is derived from layering FSM [12] and Team automata [13,14]. We combine the communicated instances into more complex ELTS by means of composition through synchronized on shared actions.

**Definition 2 (Compatibility).** Let $M_1, M_1,\ldots, M_n$ be ELTSs, $e$ is the shared action between $M_i=(S_i, s_{i,0}, F_i, \Sigma_i, T_i, \lambda_i)$ for $1 \leq i \leq n$, their interactive communication is given by $M_j \otimes (M_1 \cap M_2 \cap \ldots \cap M_n)$ where $j \notin (1, n)$ and $\exists j, \forall i \in \{1, 2, \ldots, n\} \bullet \Sigma_j \cap \Sigma_i \neq \emptyset \land ((l, e, \gamma, l') \in T_i \cap T_j$.

The Fig.3 shows the global behaviors of a composite Web services, where the high-level behavior graph Customer (C) interacts with the sub-level behavior graph which involves Vendor (V), Payment (P), Order (O), and Logistics (L). The black node is the shard states, e.g., the state V.1 is a state of Customer(C) in Fig.2. Simultaneously, the state V.1 is a state of Vendor(V). After that, we can integrate these two services together.
ALGORITHM: LSC2ELTS

INPUT: The LSC specification $c$ with $(LOC, E, R, Man, Opt)$

OUTPUT: The corresponding ELTSs for each service

FOR each $i$ in $c$

RETURN GenELTS ($i$);

IF $l'$ isNotInstanceOf $LOC(i)$ \\ F.add($l'$); \\ $\Sigma.add(e)$;

METHOD: GenELTS (instance $i$)

SET $S=NULL; s_0=NULL, F=NULL, \Sigma=NULL,$

$T=NULL, \lambda=[+, -, \emptyset]$

FOR each $i$ in $LOC(i)$

IF $l$ is initial $LOC(i)$

\[ s_0.add(l); \]

IF $l$ is final $LOC(i)$

\[ F.add(l); \]

\[ S.add(l); \]

IF $l$ is initial $LOC(i)$

\[ s_0.add(l); \]

IF $l$ is final $LOC(i)$

\[ F.add(l); \]

\[ S.add(l); \]

Put $l$ into stack $StaA$

SET $n = sizeof(StaA)$

IF($n>2$) {

IF $\{\forall l, \exists l': \exists e \in E \bullet (l, e(+), l') \in R(i)\} \lor \{\forall l, \exists l'\}$

\[ \exists e \in E \bullet (l', e(+), l) \in R(i) \]

\[ \Sigma.add(e(+)); \]

\[ T.add(l, e, +, l'); \]

IF $\{\forall l, \exists l'; \exists e \in E \bullet (l, e(-), l') \in R(i)\} \lor \{\forall l, \exists l'\}$

\[ \exists e \in E \bullet (l', e(-), l) \in R(i) \]

\[ \Sigma.add(e(-)); \]

\[ T.add(l, e, -, l'); \]

IF $\{\forall l, \exists l'; \exists e \in E \bullet (l, e, l') \in R(i)\} \lor \{\forall l, \exists l'\}$

\[ \exists e \in E \bullet (l', e, l) \in R(i) \]

\[ Tx.add(l, e, \emptyset, l'); \]

\[ TranFromManOpt(StaA); \]

METHOD TranFromManOpt(Stack $sta$)

WHILE $sta$ IS NOT RMPTY

\[ StaTmp ← sta; sta ← POP(sta); \]

\[ l ← POP(sta); \]

\[ l' ← POP(sta); \]

IF $\exists e \in E \exists \in \lambda \bullet (l, e, l') \in R(i)v (l, e, l') \in R(i)\forall (l, e, l')$

\[ e(R(i)) \text{ and the next communication } (l, e, x, l') \text{ is not Man message} \]

TO find the first Opt Message near to $l$ under $StaTmp$;

IF $(l_{opt} e^+, l_{opt} l') \in R(i) \lor (l_{opt} e^-, l_{opt} l') \in R(i)\forall (l_{opt} e, l_{opt} l') \in R(i)$

\[ l ← l_{opt} \]

\[ T.add(l, e, \emptyset, l'); \]

ELSE Break;

IF $l$ and $l'$ has not any message exchange

\[ Stmt ← StaTmp; StaTmp ← POP(StaTmp); \]

Fig.2 Two Examples of the ELTS-based Web Service Behaviors
4. Checking the Behavioral Correctness

The behavioral correctness depends on the consistency between the behaviors specification and functional requirements. In literature [15], behavioral claims split into safety claims and liveness claims. A notion of liveness states that, under certain conditions, something will ultimately occur. A notion of safety states that, under certain conditions, an undesirable event will never happen. We call that the behavioral correctness are satisfied if and only if:

1) The consistency requires that the liveness of functional behaviors needs to be guaranteed. Thus, each requirement should be correctly implemented in ELTS model;

2) The temporal logic (TL) property also needs to be satisfied so as to guarantee the safety. In this case, the uncontrollable event or well-hidden actions are not allowed to be occurred.

For our purpose, we aim to reduce the correctness problems to projection techniques for the functional behavior verifications. And CTL formulae [16] are used to express the temporal logic properties.

**Definition 3 (Checking Consistency).** Given that symbol $\downarrow$ is used as projection operator. The set of all paths in the current ELTS $M$ model is denoted as $\text{Path}(M)$. The requirements specifications include the actions $ra_i$ and the states $rs_i$ among the message communications, which is denoted as $\xi = <rs_1, ra_1, rs_2, \ldots, ra_m, rs_{m+1}>$, $|\xi| \geq 1$. The symbol $\prec$ represents a partial order; $prj(n)$ is the the $n$th element of a projection path. The consistency checking is defined as follows:

- **Partial Consistency**: $\{prj | \forall n \in \xi, \exists prj \in \text{Path}(M) \cdot prj = (\text{Path}(M) \downarrow n) \land prj \neq \emptyset\}$.

- **Complete Consistency**: $\{prj | \forall n \in \xi, \exists m \in \xi, \exists prj \in \text{Path}(M) \cdot n \prec m \land prj = ((\text{Path}(M) \downarrow n) \downarrow m) \land prj \neq \emptyset \land prj(n) \prec prj(m)\}$.

The partial correctness requires that all functional behaviors of requirements specification $\xi$ can be extracted from $\text{Path}(M)$. Here we discuss the combinatory $EF$ of CTL formulae to describe the liveness property. Formally, $EF \phi$ means that a path Exists(E) such that $\phi$ holds in some Future(F) state. The CTL formulae of the partial correctness are defined as follows: (1) $EF(s)$, where $\forall s \in \xi$ that the described requirements will be enabled (or implanted) in future starting from the initial state. For example, in OCSS of Fig.1, the formula $EF(V.2)$ requires V.2(Payment) to be occurred.

The complete correctness requires that the temporal behaviors of functional requirements should also be
satisfied. So, the unsecure events, such as deadlock, livelock, should be avoided. Here we discuss the combinatory $\text{AG}$ of CTL formulae to describe the safety property. Formally, $\text{AG}\varphi$ means that for all paths the property $\varphi$ holds globally ($G$). The CTL formulae of the complete correctness are defined as follows:

\[\text{AG}(s_k \rightarrow \text{EX}(s_{k+1} \rightarrow \text{EF}(s_{k+2})));\]

\[\text{AG}(a_k \rightarrow \text{EX}(a_{k+1} \rightarrow \text{EF}(a_{k+2})));\]

\[\text{AG}(s_k \rightarrow \text{EX}(a_k \rightarrow \text{EX}(s_{k+1}))).\]

The complete correctness is guaranteed when the verification results are outputted without any counterexample. For example, in OCSS of Fig.1, if user trigger the action “SelectCarType” in Customer(C), then the Vendor(V) will be engaged to support service. To this end, the property $\text{AG}(C.3 \rightarrow \text{EX}(\text{SelectCarType} \rightarrow \text{EX}(V.1)))$ is used to prove the correctness of the above specification.

All of properties can be automatically verified in model checker NuSMV. Fig.4 shows a verification cycle of our approach. If any error is found, NuSMV output a counterexample for user. In this case, the verification process contains a “loop” from verification back to the modeling of specifications until the specific properties are verified in ELTS model.

5. Case Study and Experimental Results

Consider again the case OCSS. First of all, it does not exist a transition between instance Customer(C) and instance Order(O) specifying the cancel order action. Hence, for $\forall i$ and $\forall j$, the property $\text{AG}(C.i \rightarrow \text{cancel} \land \text{EF} O.j)$ is not satisfied. Besides, after inputting an order, the Order(O) service may refused the order due to the selected Goods is out of stock. The absence of this transition is verified by the property $\text{AG}(O.j \rightarrow \text{refuse} \land \text{EF} V.k)$, which is also unsatisfiable. So we add the cancel and refuse action to describe these potential behaviors. Fig.5 shows the modified behavior model. The transition $(C.10, \text{cancel}, O.9)$ and $(O.10, \text{refuse}, V.6)$ are added to the OCSS.

A tool called the Play-Engine [17] allows users to create LSC requirements using a point-and-click interface, which provides the ability to create and execute LSC requirements using features called play-in and play-out. In our paper, in behaviors specification phase, we introduce Play-Engine to create the LSC requirements. In verification, we use the LSC2ELTS algorithm to translate the LSC requirements to ELTS and employ CTL formulae to verify the consistency properties. To demonstrate the effectiveness of our approach, we insert some anonymous mistakes in OCSS for investigating the capability of error detection. Tab.2 shows the experimental results using model checker NuSMV to verify LSC. In general, our verification approach performs twice as fast as the traditional approach and can find more errors in design phase.
### Table 2 Results for LSC Verification using NuSMV

<table>
<thead>
<tr>
<th>Id</th>
<th>Traditional Verification</th>
<th>Our Improved Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>States</td>
<td>Time</td>
</tr>
<tr>
<td>10001</td>
<td>1573036</td>
<td>2312312</td>
</tr>
<tr>
<td>10002</td>
<td>879408</td>
<td>372464</td>
</tr>
<tr>
<td>10003</td>
<td>97500</td>
<td>23423</td>
</tr>
<tr>
<td>10004</td>
<td>89323</td>
<td>18127</td>
</tr>
</tbody>
</table>

### 6. Conclusion

This paper proposes an approach to verify Web service behaviors based on LSC specifications. By using LSC, we describe the interaction specifications of Web service behaviors. Considering the fact that the correctness is hard to be realized by manually checking the complex message communication between instance lines of LSC, we extend the classical LTS model to describe Web service behaviors and design an algorithm to translate LSC to ELTS. In order to check the correctness, a projection approach is discussed to check the implement of functional behaviors, and CTL formulae of temporal logic are employed to express the properties which are atomically verified in model checker NuSMV. As part of our future work, we plan to extend the time constrains to LSC specifying the timed Web service composition.

### References