ARTICLE IN PRESS

Information Sciences xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Behavior modeling and automated verification of Web services

Quan Z. Sheng [a,*], Zakaria Maamar [b], Lina Yao [a], Claudia Szabo [a], Scott Bourne [a]

[a] School of Computer Science, The University of Adelaide, Adelaide, SA 5005, Australia
[b] College of Information Technology, Zayed University, Dubai, United Arab Emirates

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Web service
Cloud computing
Service behavior
Conversation message
Symbolic model checking

ABSTRACT

Cloud computing has been rapidly adopted over the last few years. However, techniques on Web services, one of the most important enabling technologies for cloud computing, are still not mature yet. In this paper, we propose a novel approach that supports dependable development of Web services. Our approach includes a new Web service model that separates service behaviors into operational and control behaviors. The coordination of operational and control behaviors at runtime is facilitated by conversational messages. We also propose an automated service verification approach based on symbolic model checking. In particular, our approach extracts the checking properties, in the form of temporal logic formulas, from control behaviors, and automatically verifies the properties in operational behaviors using the NuSMV model checker. The approach presented in this paper has been implemented using a number of state-of-the-art technologies. We conducted a number of experiments to study the performance of our proposed approach in detecting design problems in services. The results show that our automated approach can successfully detect service design problems. Our system offers a set of tools assisting service developers in specifying, debugging, and monitoring service behaviors.

1. Introduction

Over the past few years, cloud computing is gaining a considerable momentum as a new computing paradigm for providing flexible and dynamic services and infrastructures on demand [2,33,41]. Cloud computing holds the potential to transform the landscape of the IT industry by making software more attractive as services and shaping the way IT hardware is designed and purchased.

Service-oriented architecture (SOA) and Web services in general are one of the most important enabling technologies for cloud computing in the sense that resources (e.g., software, infrastructures, and platforms) are exposed in the clouds as services [37]. Most research in cloud computing so far focuses on topics such as virtulization, reliability, scalability, security and privacy of cloud services [28,25,20,30]. Although these are all important, Web services, the fundamental topic that underpins the cloud computing paradigm, have not received enough attention. In fact, despite active research into, and development of, Web services over the last decade, Web services are still not fully mature yet. According to a recent study in Europe [11], the Web currently contains 30 billion Web pages, with 10 million new pages added each day. In contrast, only 12,000 Web services exist on the Web. Even worse, most of these Web services have been deployed with dependability problems (e.g., unexpected behaviors, delayed or even no responses) [36,27,40]. One significant challenge is that, to the best of our knowledge,

* Corresponding author.
E-mail addresses: qsheng@cs.adelaide.edu.au (Q.Z. Sheng), zakaria.maamar@zu.ac.ae (Z. Maamar), lina.yao@adelaide.edu.au (L. Yao), claudia.szabo@adelaide.edu.au (C. Szabo), scott.bourne@adelaide.edu.au (S. Bourne).

0020-0255/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.ins.2012.09.016

there lacks of novel approaches and tools that would enable service developers to check the soundness and completeness of their services design so that the design problems can be identified and addressed at early stages, which ultimately ensuring the quality of the services released to clouds. Given the quick adoption of cloud computing in industry (e.g., Amazon Web Services, Google AppEngine, and Microsoft Azure), more and more cloud services will emerge, which support the development of numerous applications including mission-critical applications such as health care, air traffic control, and stock trading. This calls for the urgent need to develop novel techniques for producing highly dependable cloud services.

In this paper, we present our approach on modeling Web services so that design problems and errors can be early identified and addressed. In particular, we propose to divide Web service behaviors into two types: *operational behaviors* and *control behaviors*, based on the separation of concerns design principle [17]. The operational behavior, which is application dependent, illustrates the business logic that underpins the functioning of a Web service. The control behavior, which is application independent, acts as a controller over the operational behavior and guides its execution progress. The interactions between control and operational behaviors are modeled as *conversation sessions* (i.e., sequences of messages exchanged between the control and operational behaviors). By analyzing conversational messages and checking service behavior specifications, it is possible to verify the service design. The main contributions of this paper are as follows:

- A service behavior model that decouples operational and control behaviors of Web services. This separation of Web service behaviors eases not only the development and maintenance, but the verification (e.g., soundness and completeness checking), testing, and debugging of Web services. To the best of our knowledge, this is the first effort that identifies two behaviors of Web services.
- A service verification approach based on symbolic model checking [10]. Our approach extracts the checking properties, in the form of temporal logic formulas, from control behaviors, and automatically verifies the properties in operational behaviors.
- A fully functional prototype system that offers a set of tools for the specification of Web services and automated verification of the service design.

The reminder of the paper is organized as follows. Section 2 describes the details of our new Web service behavior model. Section 3 presents a symbolic model checking approach for verifying service designs. Section 4 focuses on the implementation and validation of the proposed system. Finally, Section 5 overviews related work and Section 6 provides some concluding remarks.

2. Service behavior model

Cloud services are normally exposed as Web services that follow the industry standards such as Web Services Description Language (WSDL). Unfortunately, WSDL does not show how they function or how their executions can be overseen. As a result, Web services are still largely perceived as simple, passive components that react upon request only [8,40]. In this section, we present a Web service behavior model using more richer description, which isolates a service from any orchestration scenario before it abstracts and separates its behavior into operational behavior and control behavior. We use statecharts [14] to model both behaviors. It should be noted that other formalisms such as Petri nets [21] also can be used. A magazine version of the content in this section appears in [31]. In this paper, we present a complete and more formal description of the service behavior model.

2.1. Operational and control behaviors

The *operational behavior* shows the business logic that underpins the functioning of a Web service. In contrast, the *control behavior* guides the execution progress of the business logic of a Web service. The control behavior relies on a number of states (`activated`, `not-activated`, `done`, `aborted`, `suspended`, and `compensated`) that are reported in the transactional Web services literature [22,23,38].

Definition 1 (*Web Service Behavior*). The behavior of a Web service is a 5-tuple $\mathcal{B} = \langle \mathcal{S}, \mathcal{L}, \mathcal{T}, s^0, \mathcal{F} \rangle$ where:

- $\mathcal{S}$ is a finite set of state names.
- $s^0 \in \mathcal{S}$ is the initial state.
- $\mathcal{F} \subseteq \mathcal{S}$ is a set of final states.
- $\mathcal{L}$ is a set of transition labels.
- $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$ is the transition relation. Each transition $t = (s^{src}, l, s^{tgt})$ is composed of a source state $s^{src} \in \mathcal{S}$, a target state $s^{tgt} \in \mathcal{S}$, and a transition label $l \in \mathcal{L}$. We qualify these transitions as *intra-behavior* (the rationale behind this qualification will be given later).

In statecharts, a label consists of three optional components (event E, condition C, and action A) and is written as $E[C]/A$.

A Web service's control and operational behaviors are instances of the Web service's behavior. These two behaviors are denoted by $\mathcal{B}_{co} = \langle \mathcal{S}_{co}, \mathcal{L}_{co}, \mathcal{T}_{co}, s^0_{co}, \mathcal{F}_{co} \rangle$ and $\mathcal{B}_{op} = \langle \mathcal{S}_{op}, \mathcal{L}_{op}, \mathcal{T}_{op}, s^0_{op}, \mathcal{F}_{op} \rangle$, respectively.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

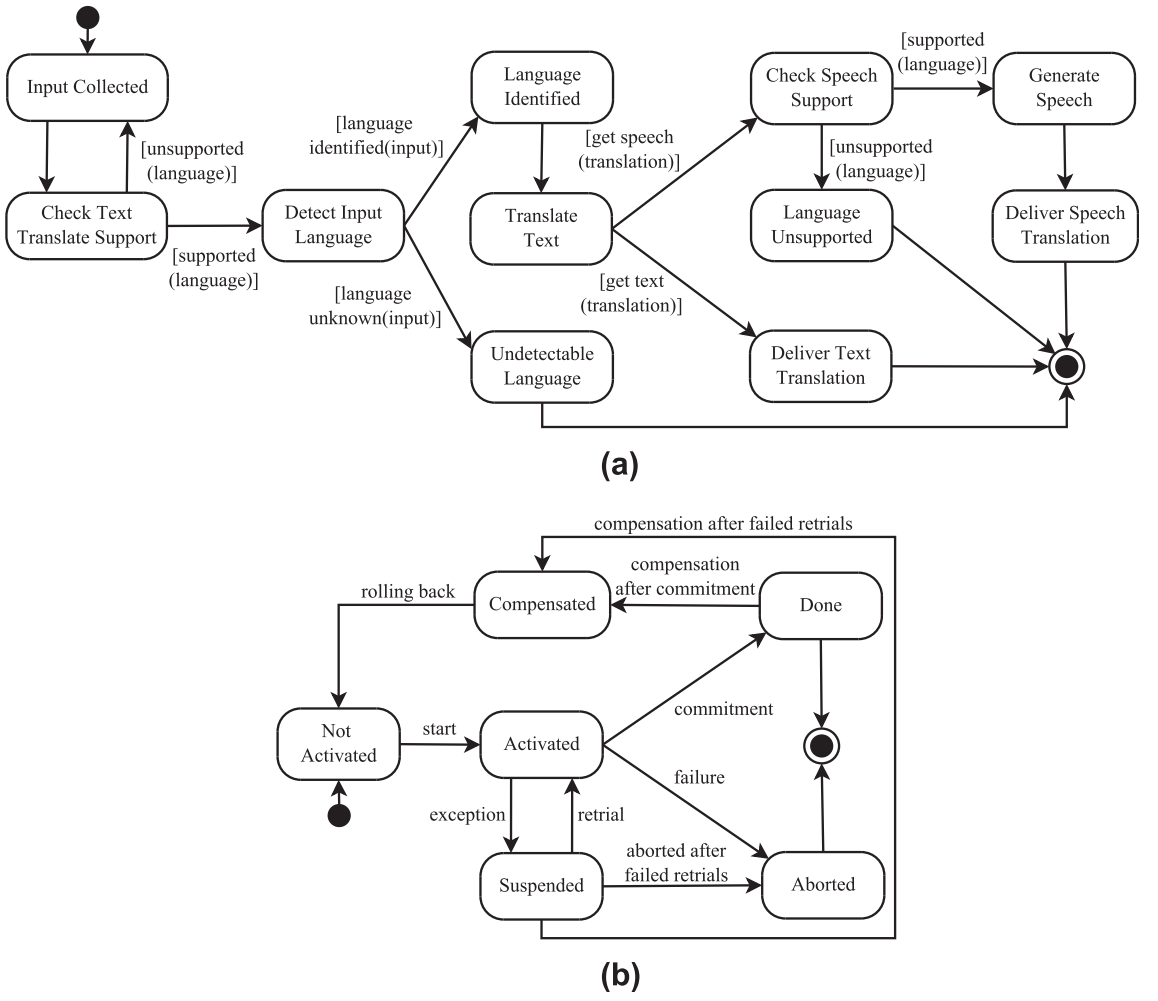

Fig. 1. The behaviors of the translation service: (a) operational behavior, and (b) control behavior.

Fig. 1a and b depict respectively the operational and control behaviors of a translation service using the `Google Translate`[1] and `Microsoft Translator`[2] APIs. Fig. 1a is the operational behavior represented in a simple statechart. The service utilizes functions of the `Google Translate` service to translate a text, and optionally provides audio speech using the `Microsoft Translator` service. Following an input of text to be translated, target language, and translation format (text or speech), the service first checks the support for the language. The base language of the input text is detected automatically, and the text translation is then produced. If an audio speech is requested, the target language is checked once more, against the languages supported by the speech function. Audio of the translation is then delivered, or the operation is canceled.

The control behavior of the translation service is illustrated in Fig. 1b, using a number of states extracted from the field of transactional Web services [6]. These states include *activated*, *not activated*, *done*, *aborted*, *suspended*, and *compensated*. Other types of states could be used if needed, without deviating from the original purpose of the control behavior of a Web service.

In Fig. 1, a Web service's control and operational behaviors rely on a set of finite sequences of states and transitions. We call these sequences *paths* and define them as follows:

Definition 2 (*Path in Web Service Behavior*). A path $p^{i\to j}$ in a Web service's behavior $\mathcal{B}$ is a finite sequence of states and transitions starting at state s^i and ending at state s^j, denoted as $p^{i\to j} = s^i \xrightarrow{l^i} s^{i+1} \xrightarrow{l^{i+1}} s^{i+2} \ldots s^{j-1} \xrightarrow{l^{j-1}} s^j$ such that $\forall k \in \{i, j-1\} : (s^k, l^k, s^{k+1}) \in \mathcal{T}$ (here the sequence of the exponents in the state names is only given for notational purposes).

For example, in Fig. 1b, $\texttt{not-activated} \xrightarrow{l^1} \texttt{activated} \xrightarrow{l^2} \texttt{done}$ is a path in the control behavior of the translation service.

[1] https://developers.google.com/translate/v2/getting_started#background-operations.
[2] http://msdn.microsoft.com/en-us/library/ff512419.aspx.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

2.2. Connecting control and operational behaviors

We pointed out that the control behavior guides the execution of a Web service represented by its control flow. We discuss now how this guidance is implemented by connecting both behaviors together.

Concretely, the process of taking on states and thus, connecting operational and control behaviors together occurs by establishing correspondences between the respective states of these two behaviors. These correspondences implement the *mapping* step and result in forming *conversation sessions*. The idea of this mapping is to associate a given state in the control behavior with a set of possible paths in the operational behavior.

Definition 3 (*Mapping Operation*). Let $\mathcal{P}_{co}$ be the set of paths in the control behavior of a Web service starting by any state in this behavior. The mapping operation is defined using the following function: $Map : \mathcal{S}_{co} \rightarrow 2^{\mathcal{P}_{op}}$.

The mapping function *Map* associates each state in the control behavior with a set (possibly empty) of possible paths in the operational behavior ($2^{\mathcal{P}_{op}}$ is the power set of $\mathcal{P}_{op}$). Fig. 2 is a mapping example in the translation service where `activated` state in the control behavior is associated with different paths in the operational behavior. One of these paths is: `input collected` → `check text translate support` → `detect input language` → `language identified` → `translate text`.

Interactions between operational and control behaviors are specified as part of the exercise of working out the conversation sessions. These sessions' role is to keep both behaviors synchronized. By specification, we mean how and when a state in the operational behavior communicates with other states in the control behavior and *vice-versa*. This communication is illustrated via the transitions between this state and the associated paths that the mapping function *Map* produces.

Definition 4 (*Specification Operation*). Let $\mathcal{P}_{op}$ be the set of paths in an operational behavior starting by any state in this behavior, and $\mathcal{L}_{\mathcal{S}}$ be the set of labels associated with the transitions between operational and control behaviors. The specification operation is defined through the following two functions:

$$Spec : \mathcal{S}_{co} \rightarrow 2^{\mathcal{L}_{\mathcal{S}} \times \mathcal{P}_{op} \times \mathcal{L}_{\mathcal{S}}} \text{ and } Next : \mathcal{S}_{co} \times \mathcal{P}_{op} \rightarrow \mathcal{L}_{co} \times \mathcal{S}_{co}.$$

The specification function *Spec* associates each state s_{co} in the control behavior with a (possibly empty) set of triples. Each triple contains i) the label of the transition from s_{co} to the first state in the operational behavior of a mapped path, ii) the mapped path itself, and iii) the label of the transition from the last state in the operational behavior of the mapped path back to s_{co}. Transitions that connect states in independent statecharts are called *inter-behaviors*.

The partial function *Next* associates a given state in the control behavior along with the mapped path in the operational behavior with the next state to take on in the control behavior along with the associated transition label.

Fig. 3 shows the specification and synchronization of our translation service's operational and control behaviors where two types of transitions exist: intra-behaviors (plain lines) and inter-behaviors (dashed lines). $Labels_{1,2,3}$ name the inter-behavior transitions and their structures are provided in Section 2.3. Fig. 3 contains $Spec$(`activated`) = {($label_1$, $path_1$, $label_2$), ($label_1$, $path_2$, $label_3$)}, where $path_1$ = `input collected` → `check text translate support` → `detect input language` → `language identified` → `translate text` → `check speech support` → `generate speech` → `deliver speech trans-lation` and $path_2$ = `input collected` → `check text translate support` → `detect input language` → `language identified` → `translate text` → `check speech support` → `language unsupported`.

Fig. 3 shows the initiation of the translation service in the control behavior with `activated` state, following receipt of user's input. Because of the (`activated`, `label`$_1$, `input collected`) inter-behavior transition, the execution of the translation service commences by checking the language support of the input, translating the input into the target language, and then checking the speech support. Afterwards, two cases exist as shown in Fig. 3. In Fig. 3a, the target language is supported,

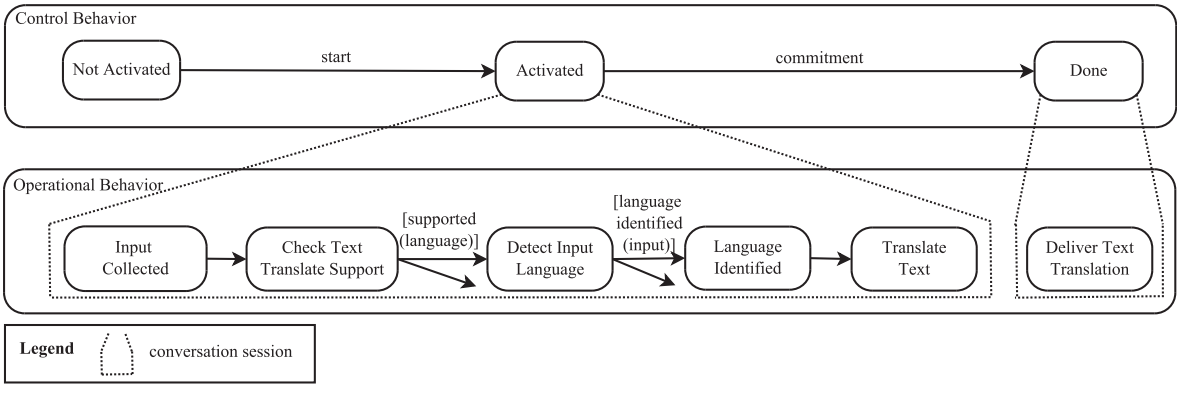

Fig. 2. Example of operational and control behaviors mapping in the translation service.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

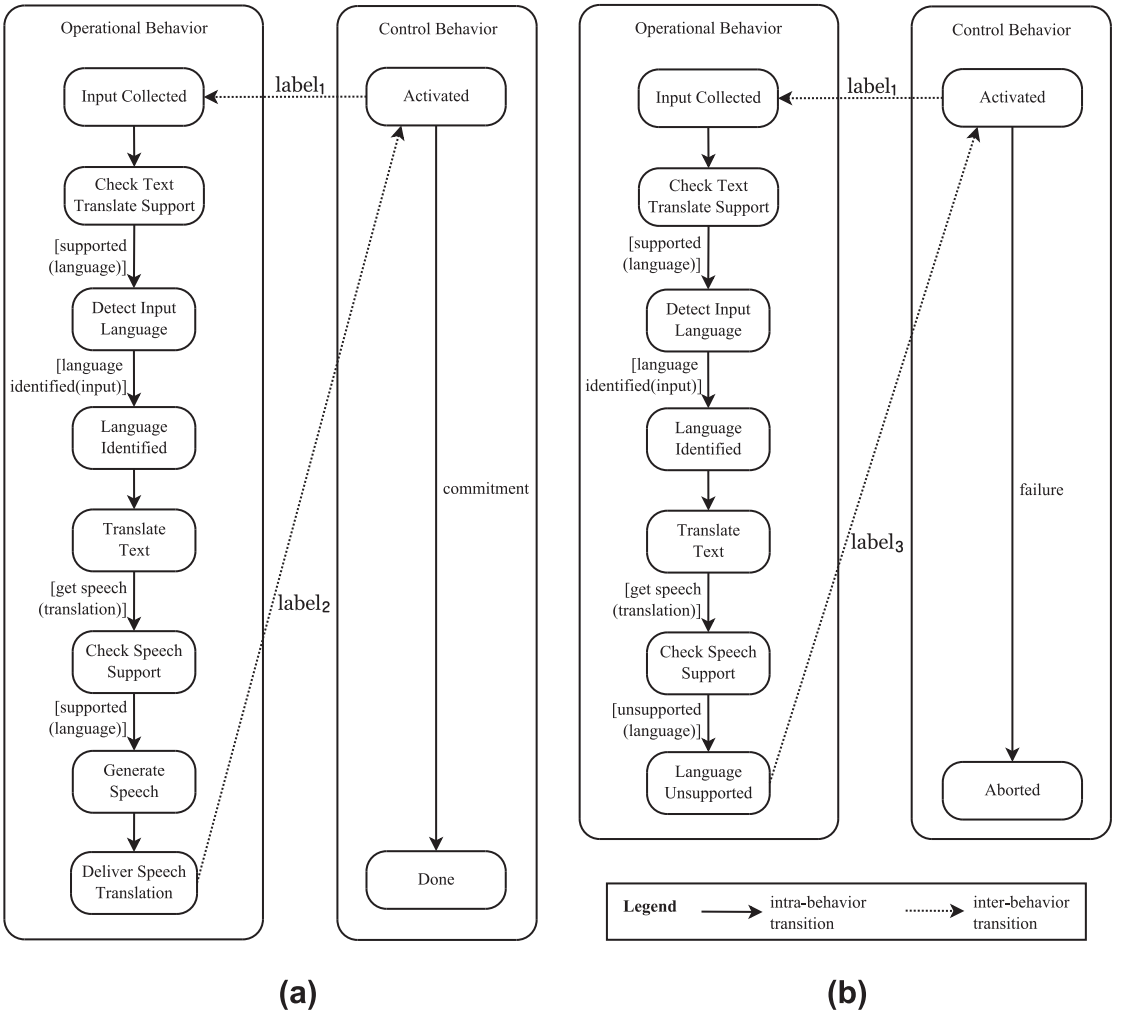

Fig. 3. Synchronization of the translation service's control and operational behaviors: (a) the success case, and (b) the failure case.

the audio speech of the text is generated and delivered to the user. Because of (`deliver speech translation`, $\texttt{label}_2$, `activated`) inter-behavior transition, the translation service completes its operation with success by transiting from `activated` to `done` states in the control behavior. However, in Fig. 3b, the target language is not supported by the translator, indicating with the `language unsupported` state. Because of (`language unsupported`, $\texttt{label}_3$, `activated`) inter-behavior transition, the whole translation service terminates its operation by transiting from `activated` to `aborted` states in the control behavior.

The formal definitions of inter-behavior and conversation session are as the following.

Definition 5 (*Inter-Behavior Transition*). The set of all inter-behavior transitions that connect the operational and control behaviors of a Web service is denoted by $\mathcal{IT}$ where $\mathcal{IT} = \mathcal{IT}_{op\to co} \cup \mathcal{IT}_{co\to op}$ such that:

- $\mathcal{IT}_{op\to co} \subseteq \mathcal{S}_{IT(op)} \times \mathcal{L}_{op\to co} \times \mathcal{S}_{IT(co)}$ is the inter-behavior transition relation starting from the operational behavior and ending at the control behavior.
- $\mathcal{IT}_{co\to op} \subseteq \mathcal{S}_{IT(co)} \times \mathcal{L}_{co\to op} \times \mathcal{S}_{IT(op)}$ is the inter-behavior transition relation starting from the control behavior and ending at the operational behavior.
- $\mathcal{S}_{IT(op)} \subseteq \mathcal{S}_{op}$ is a finite set of state names in the operational behavior that take part in inter-behavior transitions.
- $\mathcal{S}_{IT(co)} \subseteq \mathcal{S}_{co}$ is a finite set of state names in the control behavior that take part in inter-behavior transitions.
- $\mathcal{L}_{op\to co}$ is a set of inter-transitions' labels from the operational to the control behaviors, and $\mathcal{L}_{co\to op}$ is a set of inter-transitions' labels from the control to the operational behaviors ($\mathcal{L}_{co\to op} \cup \mathcal{L}_{op\to co} = \mathcal{L}_S$).

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

Definition 6 (*Web Service Conversation Session*). A conversation session between the operational and control behaviors of a Web service is a 4-tuple $\langle s_{co}, it_{co\rightarrow op}, p_{op}, it_{op\rightarrow co}\rangle$ such that:

- $s_{co} \in S_{co}, it_{co\rightarrow op} \in \mathcal{IT}_{co\rightarrow op}, it_{op\rightarrow co} \in \mathcal{IT}_{op\rightarrow co}, p_{op} \in \mathcal{P}_{op}$.
- $\mathcal{L}\dashv\lfloor(it_{co\rightarrow op}), p_{op}, \mathcal{L}\dashv\lfloor(it_{co\rightarrow op})) \in Spec(s_{co})$. The $\mathcal{L}\dashv\lfloor$ function returns the label of an inter-behavior transition, $Lab : \mathcal{IT} \rightarrow \mathcal{L}_{\mathcal{S}}$.

2.3. Types and structure of conversational messages

In Definition 5, we treated labels of inter-behavior transitions in $\mathcal{L}_{op\rightarrow co}$ and $\mathcal{L}_{co\rightarrow op}$ sets as messages. To determine appropriate messages, we draw some analogies between operational/control behaviors and networking protocols. These analogies result in the following seven types of messages: *sync*, *ping*, *success*, *ack*, *fail*, *delay*, and *syncreq*. The description of each message type is given in Table 1. Each message type is associated with a category of performative either *initiation* from the control to operational behaviors to start an execution or *outcome* from the operational to control behaviors to report on the execution status.

All conversational messages have input arguments denoted by *str*. While some arguments in *str* come out of states in the control behavior and arrive to states in the operational behavior, other arguments are the opposite. We decompose input arguments into $Part_1$ (common arguments to all messages), $Part_2$ (arguments from the control to operational behaviors), and $Part_3$ (arguments from the operational to control behaviors).

- Part_1 consists of *ID*, *Name*, *From*, *To*, and *Trigger* arguments.
- Part_2 consists of *Authorized activity-time*, *Authorized passivity-time*, and *Required participants* arguments.
- Part_3 consists of *Counter-part ID*, *Effective activity-time*, *Effective passivity-time*, and *Execution nature* arguments.

Because of the monitoring nature of *ping* and *ack* messages, their input arguments, i.e., *str*, are different from the aforementioned arguments. Hereafter we suggest examples on the use of some conversational messages. To this end, we consider messages implementing some inter-behavior transitions of Fig. 3.

- *sync*(*str*)] corresponds to (`activated`, `labell`, `input collected`) where *str* is populated with elements reported in Table 2a.
- *fail*(*str*)] corresponds to (`language unsupported`, label_3, `activated`) that is correlated to (`activated`, `labell`, `input collected`). *str* is populated with elements reported in Table 2b.
- *ping*(*str*)] is related to (`activated`, `labell`, `input collected`) where *str* is populated with elements reported in Table 2c.
- *ack*(*str*)] is related to (`activated`, `labell`, `input collected`) where *str* is populated with elements reported in Table 2d.

Table 1
Messages implementing inter-behavior transitions.

Message type	Performative category	Description
sync	Initiation	Originates from a control state and targets an operational state. The purpose is to trigger the execution of the operational states (including the targeted operational state) in a conversation session. *sync* is a blocking message, which makes the control state wait for a notification back from the last operational state to execute in this conversation session
delay	Initiation	Originates from a control state and targets an operational state that was the destination of *sync*. The purpose is to inform this operational state of the unacceptable delay in operational states execution so that corrective actions are taken. The targeted operational state reacts to the delay by submitting either *success*, *fail*, or *syncreq* to the control state depending on the current status of this execution
ping	Initiation	Originates from a control state and targets an operational state that was the destination of *sync*. The purpose is to check the liveness of the operational states in the conversation session of the targeted operational state
success	Outcome	Originates from an operational state and targets the control state that submitted *sync*. The purpose is to inform this control state of the successful execution of the operational states in a conversation session and to return the execution thread back to this control state as well. *success* is coupled with *sync*
fail	Outcome	Originates from an operational state and targets the control state that submitted *sync*. The purpose is to notify this control state of the failure execution of the operational states in a conversation session and to return the execution thread back to this control state as well. *fail* is the opposite of *success* and is coupled with *sync*
syncreq	Outcome	Originates from an operational state and targets the control state that submitted *sync*, and denotes request for another synchronization. *syncreq* is motivated by the possibility of unhandled exceptions in the conversation session of this operational state and thus, required actions are to be taken
ack	Outcome	Originates from an operational state and targets a control state. The purpose is to confirm the liveness of the operational states in a conversation session. *ack* is coupled with *ping*. If no *ack* is received within a time limit, the control state submits another *sync* to the operational state prior to declaring the failure of this conversation session

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

Table 2
Examples related to conversational messages.

Case a		Case b	
ID	ID1	*ID*	ID2
Name	Label1	*Name*	Label3
From	Activated	*From*	language unsupported
To	Input collected	*To*	Activated
Trigger	15 s [request receipt]	*Trigger*	Execution failure
		Counter-Part-Id	ID1
Authorized-Activity-Time	1 min	*Effective-Activity-Time*	45 s
Authorized-Passivity-Time	Null	*Effective-Passivity-Time*	Null
Required-Participants	Google Translate Service	*Execution-Nature*	failure
Case c		Case d	
ID	ID3	*ID*	ID4
From	Activated	*From*	input collected
To	Input collected	*To*	Activated
		Counter-Part-Id	ID3
		Status	OK

2.4. Sequence of conversational messages

A sequence of conversational messages is an ordered list of messages that are put together in a consistent way. Capturing such sequences generates the *execution traces* of a Web service and turns out to be very useful for post-analysis activities. An execution trace helps review all actions that a Web service performed from operational and control perspectives. For instance, *delay* messages can be examined so that similar and frequent cases in the operational behavior are addressed. In addition, *fail* messages might indicate a reliability problem in the control behavior requiring corrective actions. The following are some possible sequences of messages where stands for "next":

- *sync.success* (resp. *sync.fail*) refers to synchronization followed by success (resp. failure).
- *sync.delay.syncreq.sync.success* refers to synchronization followed by delay then by request of re-synchronization then by synchronization then by success.

All possible sequences of conversational messages can be represented using a combination of if-then rules. Formally:

Definition 7 (*Sequences of Conversational Messages*). Let n be the number of conversational messages exchanged during a session. Also, let $m_1, \ldots, m_7$ be conversational messages of Table 1, and $m_i(t)(i \in \{1, \ldots 7\})$ be a conversational message sent at time t from a state in the operational behavior (resp., the control behavior) to a state in the control behavior (resp., the operational behavior). All possible sequences of conversational messages can be represented using a combination of rules of the form:

$$\forall t \in [0, n-2], m_i(t) \Rightarrow \bigvee_{j \in \mathcal{J}} m_j(t+1) \text{with } \mathcal{J} \subseteq \{1, \ldots, 7\} \text{ and } i \neq j$$

where $m_i(0) = sync \vee m_i(0) = ping$ and $m_i(n-1) = success \vee m_i(n-1) = fail \vee m_i(n-1) = ack$.

Using this formula, it is possible to specify some conditions that help us determine that a conversation sequence of messages is *well-formed*. For instance, to avoid design errors such as deadlock situations of *sync.delay.delay*..., we can put some restrictions on the conversations to be developed. Examples of some restrictions are the rules as follows:

- Each *sync* message in a sequence should be followed by either a *success*, *fail*, or *delay* message for the sake of synchronization matching. Formally, $sync(t) \Rightarrow success\ (t+1) \vee fail(t+1) \vee delay(t+1)$.
- Each *ping* message in a sequence should be followed by an *ack* message for the sake of synchronization matching. Formally, $ping(t) \Rightarrow ack(t+1)$.

A well-formed sequence of conversational messages suggests some benefits that make the engineering exercise of Web services sound and complete. For instance, it would be possible to (i) determine if a Web service's operational and control states were properly executed (e.g., a Web service is not indefinitely waiting for an *ack*), (ii) to detect errors at design-time (e.g., having a *sync* without a corresponding synchronization matching transition), and (iii) to verify Web services' non-functional properties (e.g., *delay* messages submitted upon response-time verification). In [7], we propose a set of rules that guarantee a well-formed conversation between operational and control behaviors. Interested readers are referred to that paper for more details. In the next section, we will formally introduce our approach on automated verification of services design.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

3. Verification of control and operational behaviors

Our verification approach is based on the formal model checking of conversations of Web service behaviors. As shown in Fig. 4, we propose to verify that the operational behavior is well-designed and fits well with the control behavior in two ways. Firstly, we verify that the operational behavior and the control behavior are synchronized by checking the message sequences defined above. Secondly, we verify that the operational behavior fits well within the control behavior using a novel approach that extracts properties from the control behavior and verifies these properties on the operational behavior. We represent these properties as temporal logic properties [9].

To prevent state-space explosion, we convert the operational behavior into a Kripke structure [9] (step 2 in Fig. 4). We perform a semantic state abstraction algorithm to align states in the operational behavior to states in the control behavior using information provided by the service designer (step 3), reduce the number of states using our reduction algorithm (step 4) and verify the temporal logic properties using model checking tools such as NuSMV [9]. Our approach is fully automated and the details will be described below.

3.1. Extracting properties from control behaviors

As discussed above, the control behavior of a Web service is a domain-application independent behavior that guides the execution progress of the service through conversations with the operational behavior. By nature, the states in the control behavior will rarely be subject to change, and such several properties can be associated with them. Towards ensuring the correctness of Web Services at design time, we propose to extract logical properties from the control behavior and verify that the operational behavior conforms to them. This ensures that the operational behavior is well-designed and fits with its control behavior.

We propose to represent these logical properties using temporal logics such as Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [10] to facilitate the verification of a rich set of properties. The properties are associated with each state in the control behavior. We employ both LTL and CTL logics because they are not equivalent, in that there exists properties in LTL that cannot be expressed in CTL and vice versa. For example, $\mathcal{AF}(p \wedge \mathcal{X}_p$ in an LTL property that cannot be expressed in CTL, and $\mathcal{AGEF}p$ vice versa. This is because LTL cannot express that at some instants along the execution it would be possible to extend the execution to explore all possible paths. As such, the LTL property is checked for a particular run,

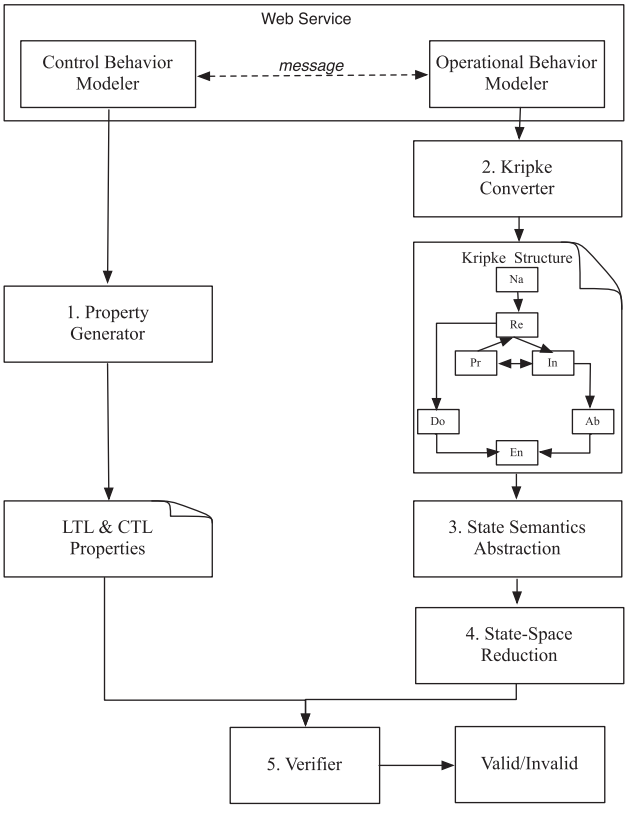

Fig. 4. Verifying well-designed control and operational behaviors.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

with no possibility of switching to another run during the checking. On the other hand, the CTL semantics checks a formula on all possible runs and will try either all possible runs ($\mathcal{A}$ operator) or only one run ($\mathcal{E}$ operator) when facing a branch. As such, with carefully chosen properties, both LTL and CTL can be used to complement rather than contradict each other [5].

In the following, we represent LTL and CTL formulae using known symbols and operators summarized below:

- $\mathcal{F}\varphi$: sometimes in the **F**uture, φ is true in some future moment.
- $\varphi\mathcal{U}\psi$: **U**ntil, φ is true until ψ is true.
- $\mathcal{G}\varphi$: **G**lobally in the future, φ is true in all future moment.
- $\mathcal{X}\varphi$: ne**X**t time, φ is true in the next moment in time.

As CTL permits only branching-time operators, each of the LTL operators defined above ($\mathcal{G}, \mathcal{F}, \mathcal{X}$, and $\mathcal{U}$) must be immediately preceded by a path quantifier (i.e., $\mathcal{A}$ for **A**ll paths, and $\mathcal{E}$ for **E**xisting a path). It is important to note that we are considering fair LTL and CTL [10], which means that in any computation, some states, called *fair states*, should be reached. For instance, in the control behavior, *Done* is a fair state and should always be reached.

We use the following initials for simplicity to specify the control behavior states when defining the logical properties to be checked: (1) *Na*: Not Activated, (2) *Re*: Received, (3) *In*: Invoked, (4) *Su*: Suspended, (5) *Ab*: Aborted, (6) *Pr*: Process, (7) *Co*: Compensated, (8) *Do*: Done, and (9) *En*: End.

Let $\rightarrow$ be the logical implication. Examples of fair LTL properties that can be defined and verified for the control behavior are:

- $\Phi = \mathcal{G}(Na \rightarrow \mathcal{X}\mathcal{R}])$ – always a `Received` state after a `Not Activated` state.
- $\Phi = \mathcal{G}(Re \rightarrow \mathcal{X}\mathcal{F}(In \vee Ab \vee Su \vee Do))$ – always an `Invoked`, `Aborted`, `Suspended`, or `Done` state in the future after a `Received` state.
- $\Phi = \mathcal{G}(Co \rightarrow \mathcal{X}\mathcal{F}Na)$ – always a `Not Activated` state in the future after a `Compensated` state.
- $\Phi = \mathcal{G}(Do \rightarrow \mathcal{X}\mathcal{F}(En \vee Co))$ – always an `End` or `Compensated` state in the future after a `Done` state.
- $\Phi = \mathcal{G}((Do \vee Ab) \rightarrow \mathcal{X}\mathcal{F}En)$ – always an `End` state in the future after a `Done` or `Aborted` state.
- $\Phi = \mathcal{G}(In \rightarrow \mathcal{X}\mathcal{F}(Ab \vee Pr \vee Re \vee Su))$ – always an `Aborted`, `Process`, `Received` or `Suspended` state in the future after an `Invoked` state.
- $\Phi = \mathcal{A}\mathcal{F}((Re \vee In) \rightarrow \mathcal{X}(Do \vee Ab))$ – always a `Done` or `Aborted` state on all paths after a `Received` or `Invoked` state.

It should also be noted that the last LTL property defined above cannot be expressed in CTL.

Similarly, examples of CTL properties that can be verified from the control behavior are:

- $\Phi = \mathcal{A}\mathcal{G}(Na \rightarrow \mathcal{A}\mathcal{X}Re)$ – there is always a path from state `Not Activated` to state `Received`.
- $\Phi = \mathcal{A}\mathcal{G}(Re \rightarrow \mathcal{A}\mathcal{X}\mathcal{A}\mathcal{F}(In \vee Ab \vee Su \vee Do))$ – there is always a path from state `Received` to states `Invoked`, `Aborted`, `Suspended`, or `Done`.
- $\Phi = \mathcal{A}\mathcal{G}(Co \rightarrow \mathcal{A}\mathcal{X}\mathcal{A}\mathcal{F}Na)$ – there is always a path from state `Compensated` to state `Not Activated`.
- $\Phi = \mathcal{A}\mathcal{G}(Do \rightarrow \mathcal{A}\mathcal{X}\mathcal{A}\mathcal{F}(En \vee Co))$ – there is always a path from state `Done` to state `End` or `Compensated`.
- $\Phi = \mathcal{A}\mathcal{G}((Do \vee Ab) \rightarrow \mathcal{A}\mathcal{X}\mathcal{A}\mathcal{F}En)$ – there is always a path from state `Done` or `Aborted`, to state `End`.
- $\Phi = \mathcal{A}\mathcal{G}(In \rightarrow \mathcal{A}\mathcal{X}\mathcal{A}\mathcal{F}(Ab \vee Pr \vee Re \vee Su))$ – there is always a path from state `Invoked` to state `Aborted`, `Process`, `Received` or `Suspended`.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}(En)$ – state `End` is always potentially reachable.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}(Do)$ – state `Done` is always potentially reachable.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}(Ab \vee Do)$ – states `Aborted` or `Done` are always potentially reachable.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}(Re \rightarrow In)$ – a path from state `Received` to state `Invoked` is always potentially reachable.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}((In \rightarrow EXPr) \vee (In \rightarrow EXRe))$ – a path from state `Invoked` to the next state `Process`, or from state `Invoked` to next state `Received` is always potentially reachable.
- $\Phi = \mathcal{A}\mathcal{G}\mathcal{E}\mathcal{F}((Pr \rightarrow EXRe) \vee (Pr \rightarrow EXAb))$ – a path from state `Process` to the next state `Received`, or from state `Process` to next state `Aborted` is always potentially reachable.

The last six CTL properties are examples of CTL properties that cannot be expressed in LTL, for reasons discussed above.

3.2. Kripke-structured operational behavior

The verification of the above properties may lead to state-space explosion, particularly for complex services. Towards reducing this problem, we propose to *transform* the operational behavior into a Kripke structure [19] that captures only the states in the operational model that are relevant to its behavior. This reduces complex behaviors to simplified representations without complex structures such as and-states. Next, we perform a *semantic state abstraction* as shown in step 3 in Fig. 4. This is because the CTL and LTL properties defined above refer to control behavior states such as `Invoked`, or `Done`, which may not be present in the operational behavior. Towards this, we match operational behavior states to the corresponding state in the control behavior. For example, if the text-to-speech Web service is invoked in the operational behavior,

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

this state is abstracted as `Invoked`. These mappings from operational behavior states to the symbols *Na*, *Re*, *In*, *Su*, *Ab*, *Pr*, *Co*, *Do*, and *En* are performed at design-time by the user, and are saved into a hash map to be later used in the abstraction algorithm. To further reduce the number of states, we also developed a *reduction algorithm*, detailed in Section 3.2.1.

A Kripke structure is a nondeterministic finite state machine that is used in model checking to represent system behavior. In a Kripke structure, the nodes represent reachable states and edges represent state transitions. Labels attached to each node represent the set of properties that hold in the corresponding state.

Formally, let $\mathcal{AP}$ be a set of atomic proposition (e.g., boolean expression over variables, constants and predicate symbols), a Kripke structure is a 4-tuple expressed as: $\mathcal{M}^k = \langle \mathcal{S}^k, \mathcal{I}^k, \mathcal{R}^k, \mathcal{L}^k \rangle$, where $\mathcal{S}^k$ is a finite set of states, $\mathcal{I}^k : \mathcal{I}^k \subseteq \mathcal{S}^k$, is the set of initial states, $\mathcal{R}^k : \mathcal{R}^k \subseteq \mathcal{S}^k \times \mathcal{S}^k$ ($\forall s \in \mathcal{S}^k, \exists s' \in \mathcal{S}^k$, such that $(s, s') \in \mathcal{R}^k$), is the transition relation, and $\mathcal{L}^k : \mathcal{S}^k \to 2^{\mathcal{AP}}$, is a labeling function that labels states with atomic propositions from a given language. It defines for each state $s \in \mathcal{S}^k$ the set $\mathcal{L}^k(s)$ of all atomic propositions that are valid in s. In a Kripke structure, a transition function must be complete, i.e., each state must have a transition from it. Deadlock states are those in which the state has a single outgoing edge to itself.

3.2.1. Transforming operational behavior to Kripke structures

We propose the following algorithm for automatically translating an operational behavior into a Kripke structure. In our algorithm, each state s_{op} in the operational behavior is translated to a set of atomic states and transitions in the Kripke structure $\mathcal{M}^k$, in which one or more of the logical properties extracted from the control behavior are true. Each transition is translated to one or many transitions. If s_{op} is an atomic state, it is translated into exactly one state in $\mathcal{M}^k$ with the same content. If s_{op} is a compound state, i.e., a statechart state, two cases exist: (i) s_{op} is a sequential state; (ii) s_{op} is a concurrent state (e.g., an and-state). In both cases, each atomic state in s_{op} is translated into one state with the same content and all the end states in s_{op} are translated to one end state in $\mathcal{M}^k$.

In the case of a sequential compound state s_{op}, we keep the sequences of states and the connector is replaced by the next state or by the first state of the next sequential state or concurrent state. In the case of a concurrent compound state s_{op}, the concurrent states are simply considered as sequential and the sequence order is selected randomly. This is because in a concurrent state, all the states must be considered but their order is not important, and only the last state in the selected order is related to the next state by a transition. The number of possible Kripke-like structures thus depends on the number of states in the concurrent states. However, all executions are equivalent and thus only one structure should be considered. The conditional selections are captured by deterministic transitions. Transitions between atomic states are translated to transitions between the corresponding states in the Kripke-like structure. Transitions between atomic and sequential states or concurrent states are translated into transitions between the atomic state and the first state of the sequential state or concurrent state.

Fig. 5a shows the Kripke-like model after translating the operational behavior of the translation service in Fig. 1a using the above translation procedure. The service does not contain any concurrent states and as such the translation is straightforward. We perform a semantic state abstraction as discussed above, resulting in the model shown in Fig. 5b.

We also introduce a *reduction algorithm* to reduce the number of states and transitions. The idea is that two states labeled with the same atomic propositions using the valuation function $\mathcal{L}^k$ are equivalent and they can be reduced to only one state. For all s_1 and s_2 in the Kripke structure, if s_2 is reduced to s_1, then:

- If (s_1, s_2) and (s_2, s_1) are two transitions, then they are replaced by the transition (s_1, s_1).
- If only one of the two transitions exists, then it is removed.
- For all x, if (s_x, s_2) is a transition, then it is removed and replaced by the transition (s_x, s_1) if such a transition does not exist.
- For all y, if (s_2, s_y) is a transition, then it is removed and replaced by the transition (s_1, s_y) if such a transition does not exist.

The reduction algorithm preserves the behavioral semantics of the Kripke structure and Fig. 5c shows the result after reducing the Kripke-like model in Fig. 5b.

3.3. Symbolic model checking of web service behaviors

In this paper, we exploit *symbolic model checking* as it reduces state space explosion by avoiding to build or explore the state space corresponding to the models explicitly. Instead, a symbolic representation is used, based on ordered binary decision diagrams (OBDDS) or propositional satisfiability (SAT) solvers [10]. This is further enhanced by our Kripke-structure transformation as discussed above. We employ the NuSMV [9] model checker, which is a software tool for the formal verification of finite state systems against specifications in the temporal logics LTL and CTL. It is aimed at reliable verification of industry-size designs, for use as a backend for other verification tools, and as a research tool for formal verification techniques.

We provide an automatic translation from the Kripe-like structure obtained from the operational behavior to the SMV (Symbolic Model Verifier) code. This is done by transforming each state in the reduced Kripke structure of the operational behavior into a `case` branch in a `switch` statement. For example, for the reduced Kripke structure in Fig. 5c, we define an initial state *Na*, and a switch statement as shown below:

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

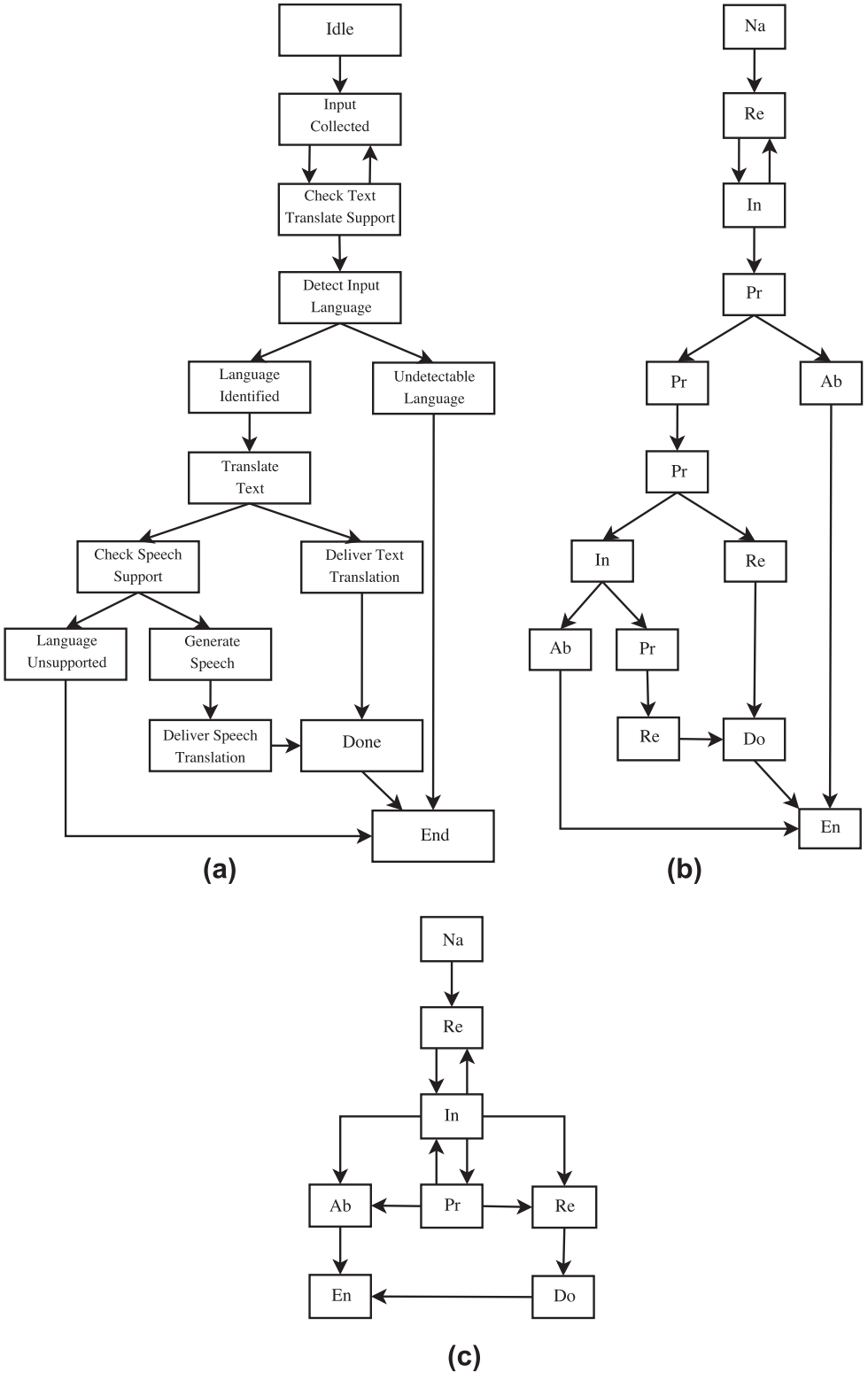

Fig. 5. (a) Model of the translation service, (b) Kripke model of the translation service, and (c) reduced Kripke model of the translation service.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

```
init(state) := Na;
next(state) := case
          {state = Re}: In
          {state = In}: Ab, Pr, Re
          ...
```

The code above describes an initial state Na, followed by a decision on the next states depending on the current state. The code follows closely the graph presented in Fig. 5c and includes all the states in the structure, as shown in Fig. 8. The properties to be checked are extracted from the control behavior based on the control behavior states and their associated LTL and CTL properties defined above, are translated into LTL and/or CTL, and then appended to the SMV code.

4. System implementation and experiments

In this section, we describe a prototype implementation of our proposed approach for developing dependable cloud services. The implementation and experiments conducted have shown that the ideas proposed in this paper are realizable using existing technologies.

4.1. Implementation

Fig. 6 shows the architecture of our prototype system, which has been implemented in Java and is based on state-of-the-art technologies like XML, SOAP, WSDL and symbolic model checking. Service engineers access our system's CASE-like tool for cloud services design via a dedicated user interface. Specifically, the `ControlBehaviorModeler` and the `OperationalBehaviorModeler` assist engineers in specifying the control and operational behaviors of a cloud service, respectively. In the implementation, we extended ArgoUML,[3] a leading open source UML modeling tool for behavior specification (see Fig. 7). Behaviors represented in ArgoUML statechart diagrams are exported as XMI files. The `ConversationModeler` takes a service's behavior specifications (i.e., the XMI files generated by the behavior modelers) as input and produces conversation specifications as output (that is, the inter-transitions and message sequences between the two behaviors). The respective modeler then translates all these specifications into XML documents for subsequent processing.

The `BehaviorConverter` is used to extract the properties from the control behaviors as the LTL and CTL representations (see Section 3.1), while the `BehaviorTranslator` takes the responsibility of abstracting the operational behavior into a Kripe-like structure system model and automatically translating the model into SMV code (see Section 3.2.1). The `ModelProcessor` then generates the model specifications using the outputs from the `BehaviorConverter` and the `BehaviorTranslator`, which will be used as input for the `ServiceVerifier` to do the model checking of cloud services. In our implementation, the `ServiceVerifier` has been implemented using NuSMV2.[4]

Finally, the `ConversationController` implements functions to support the conversations between operational and control behaviors. Specifically, it provides methods for managing conversation sessions, triggering transitions, and communicating with the `ServiceManager`, which is responsible for managing and coordinate service execution. Through the `ConversationController`, `ModelController`, and the `ServiceManager`, a cloud service engineer can conveniently track and analyze (if necessary) the service's execution according to its conversation definition (for example, whether messages are received and sent in an appropriate order).

It should be noted that we integrated all these features in a single interface. In the interface shown in Fig. 7, we offer a number of buttons that a service designer can simply click to perform the checking of services design. For example, after the designer completes her design of a cloud service, she can click the `ConversationChecking` button and the analysis result of the conversation messages between control and operational behaviors will be displayed in the bottom left panel. If she clicks on the `ModelChecking` button, the model checking process will be triggered and a pop-up window will appear displaying the result (see Fig. 8). The designer also can simulate the execution of the service (see Fig. 9). When the designer is happy about the service design and clicks on the `ServiceDelpoying` button, the service specification will be transformed into executable Web service, represented in BPEL process, using RubyGems 1.0.1.[5] The generated BPEL process is then deployed to GlassFishESB 2.1,[6] an open source application server, and exposed as a Web service.

4.2. System validation

To evaluate the proposed approach, we conducted experiments using the implemented prototype system. Our experiments focus on studying the performance on detecting design problems in Web services.

[3] http://argouml.tigris.org/.
[4] http://nusmv.fbk.eu/.
[5] http://rubyforge.org/projects/rubygems.
[6] http://openesb-dev.org/.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

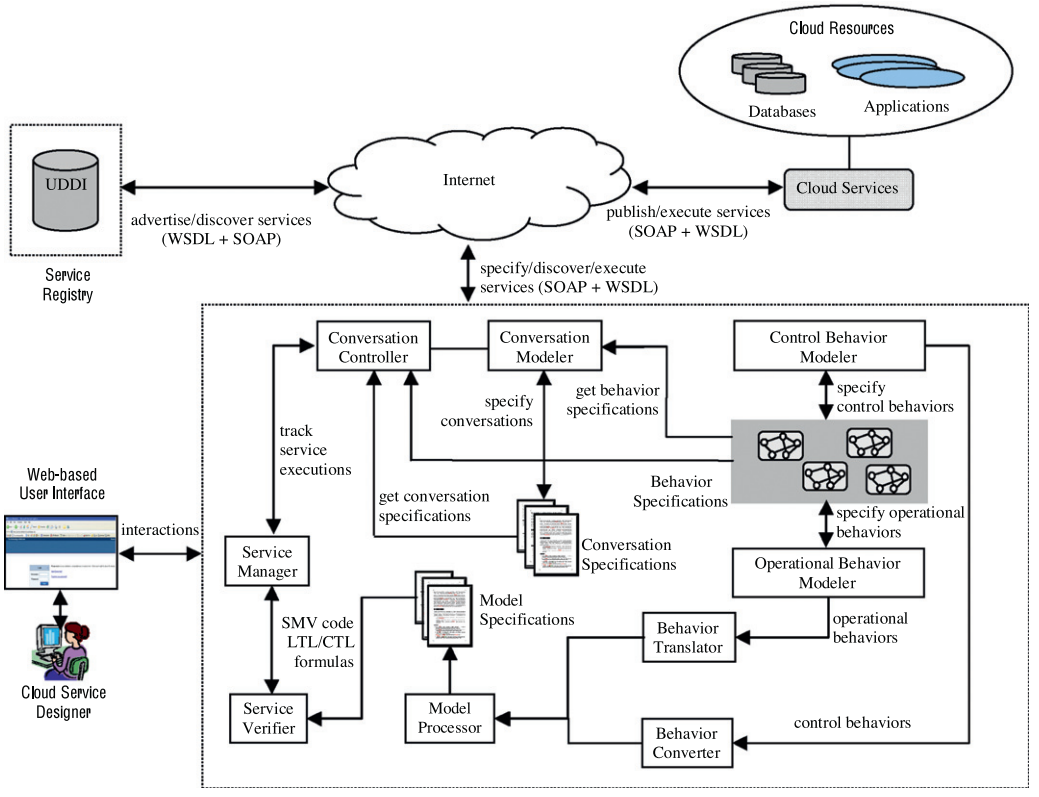

Fig. 6. Prototype architecture.

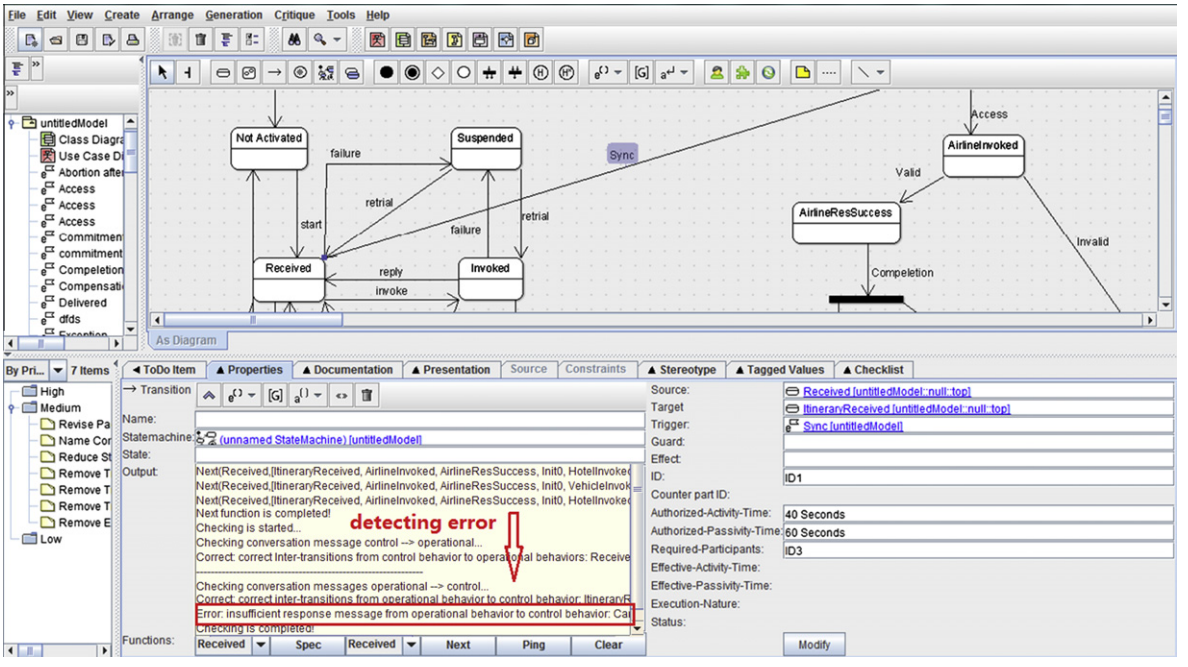

Fig. 7. Specifying service behaviors.

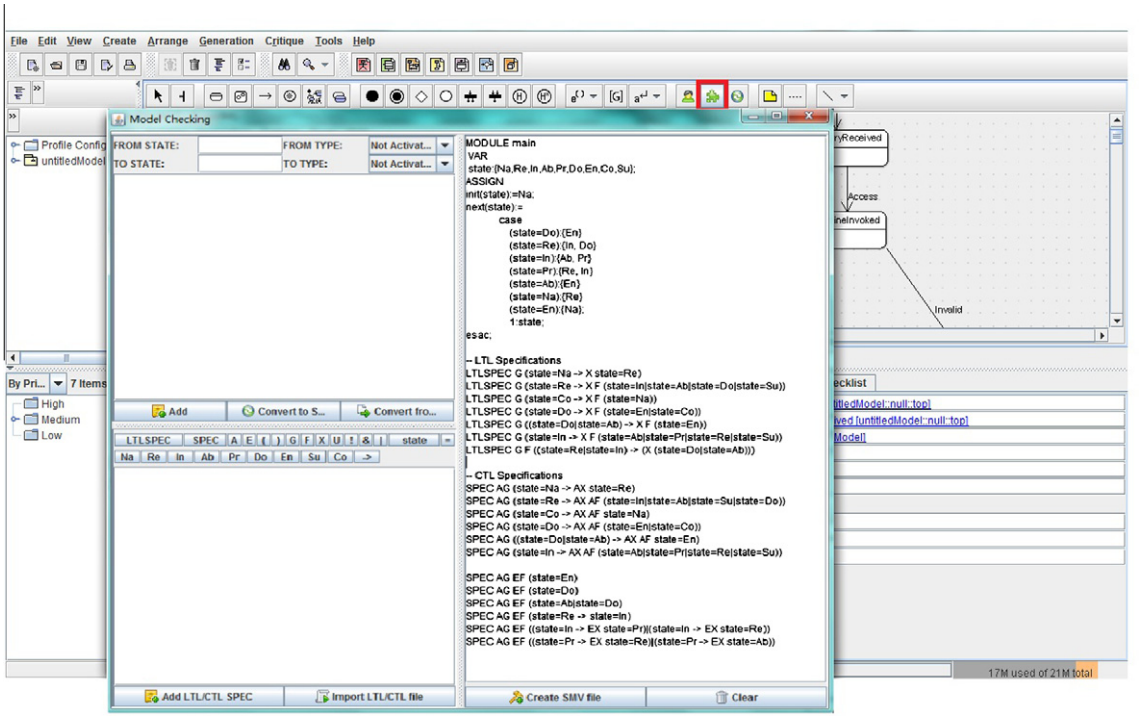

Fig. 8. Model checking of service behaviors.

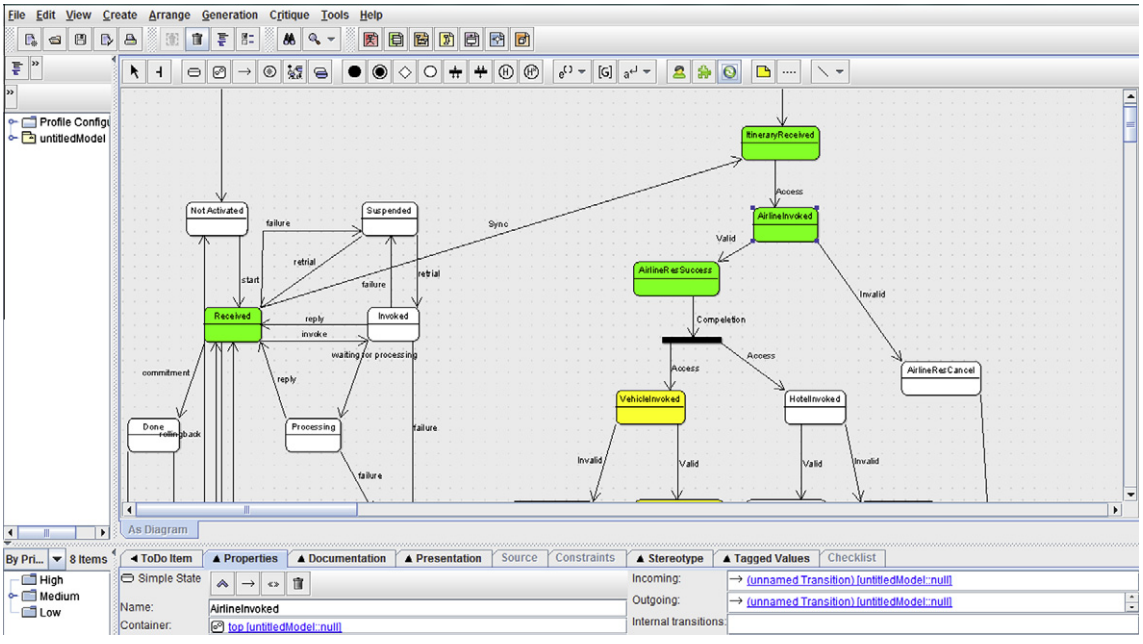

Fig. 9. Simulation of service execution step by step.

We validated the design of ten Web services including the translation service described in Fig. 1, as well as other well-known Web service examples such as `WeatherWS`, a weather information collection service, and `TravelWS`, a travel planning Web service. For each Web service, we designed 20 test cases and each test case was designed in a way of intentionally modified with wrong designs (e.g., wrong state in control behaviors, incorrect conversation sequences, properties not satisfied by specifications). We then ran the system (particularly `ServiceVerifier`) using these test cases to check if the design problems are correctly detected. Fig. 10 shows partial results of the conversation checking and model checking

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

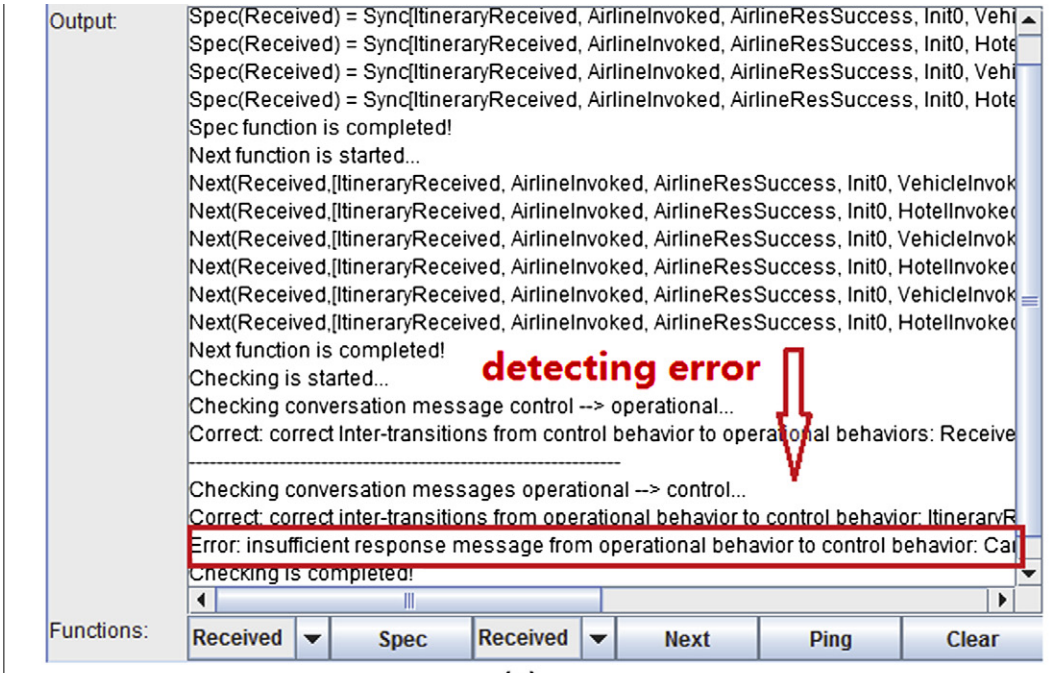

Fig. 10. Verification results on (a) the conversation checking, and (b) fragment of the model checking.

of a particular test case of `TravelWS` service. In the experiments, our system detected successfully the design problems of all test cases. Interested readers are referred to [39] for more results on detecting design problems of additional Web services and test cases.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

The experimental results show that our system supports the specification of Web services and can perform automated checking to detect service design problems. In the future, we plan to conduct more experiments to study further the performance (e.g., scalability when services become very complicated) and the usability of the system.

5. Related work

Over the last few years, Web services has been a very active area of research and development [36,40,8,31,4,12,18,26]. Our work proposed in this paper is at the crossing point of several initiatives that examine Web services from different perspectives such as *conversation*, *behavior modeling*, *management*, and *verification*.

From the conversational perspective, two specifications back the added value of conversations to Web services: the Web Services Conversation Language [3], which describes the external behavior of a Web service in terms of acceptable sequences to invoke a Web service, and the Web Services Choreography Interface [1], which supports message correlation, message choreography, and service operation compensation. These specifications focus on examining the surrounding environment of a Web service, not its internal structure.

From a behavioral modeling perspective, representative specifications include the Semantic Markup for Web Service (OWL-S,[7] formerly DAML-S) and the Web Service Semantics (WSDL-S[8]) [35]. The former organizes the description of a Web service along three categories, namely *profile* (what it does), *process model* (how it operates internally), and *grounding* (how it accepts requests), while the latter provides a lightweight approach for creating semantic descriptions of Web services. Unfortunately, verifying the design of Web services is not discussed in these specifications. From a management perspective, there exist a good number of specifications. For instance, the W3C Web Services Architecture Working Group suggests the life cycle of a Web service, and how this Web service processes requests.[9]

Our approach stresses the (intra-) conversation sessions that Web services initiate and thus, goes beyond the simple use of conversations as an interaction means. In particular, we (i) separate the operational and control behaviors of Web services, (ii) specify conversation-based mechanisms that synchronize these behaviors, (iii) develop performatives to implement these conversations, and (iv) develop an automated symbolic model checking approach for verifying service design. Compared to the aforementioned conversation specifications and existing research initiatives on Web service conversations [4,24,12,16], this is the first work that the operational and control behaviors are combined and made accessible (interactible) to each other through conversations. It is noted that some existing initiatives focus on either the operational behavior (e.g., [4]) or the control behavior (e.g., [24]) of Web services, while other works do not even acknowledge the existence of these behaviors.

Verification of Web services has become an active research topic recently. In [29], the authors propose to verify the Atomic Transaction Protocol (WS-AT) using UPPAAL. The verification aims at ensuring that coordinations of Web services reach an agreement on the outcome of a distributed transaction. In [12,13], the authors proposed an approach to verify Web services specified in a Petri net model. Their tool can be used to check that Web services satisfy LTL properties. In their approach, BPEL specifications are converted into guarded automata, which are then translated into Promela language and checked in the SPIN model checker [15]. Hull and Su present several composition models in [16] including the Roman model, and the Mealy conversation model. They also give techniques for analyzing Web services by translating services into formalisms that are suitable for analysis (e.g., state machines are extended to mealy machines and process algebra). In [32], the authors propose a framework to check behavioral correctness of the Web services by monitoring runtime conversations between partners. They exploit a subset of UML 2.0 Sequence Diagrams as a property specification language to capture safety and liveness properties. Although these verification approaches are similar to ours, our work has several advantages. The verification in our approach is based on separating behaviors and symbolic model checking (NuSMV). Compared to automata-based techniques, our symbolic model checking approach does not suffer from the state explosion problem. In addition, our approach can check not only LTL specifications, but also CTL specifications.

Finally, in [34], the authors show the importance of asynchronous messaging in sharing information and resources in the form of Web processes. Web service interaction models are formalized into a conversation concept with ordering constraints on messages. FIFO queues are considered in the design of message passing between services. In terms of verification, only some abstract strategies of model checking service composition for both bottom-up and top-down design approaches are outlined. However, the detailed analysis or implementation of these strategies are not provided.

6. Conclusion

With the increasingly rapid adoption of cloud computing during the past few years, more and more cloud services will be available in the near future. Unfortunately, techniques on developing dependable Web services, which underpin the development of flexible and reliable cloud services, are still not fully mature yet. In this paper, we have presented a novel approach that supports dependable development of Web services. In particular, we introduced a new Web service model

[7] http://www.w3.org/Submission/OWL-S.
[8] http://www.w3.org/Submission/WSPL-S.
[9] http://www.w3.org/TR/2004/NOTE-wslc-20040211.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

that separates service behaviors into operational and control behaviors. The coordination of operational and control behaviors at runtime is facilitated by conversational messages. This richer description of Web services offers a number of advantages such as easy maintenance and better testing, analyzing, and debugging. We also proposed an automated service verification approach based on symbolic model checking. Our approach extracts the checking properties, in the form of LTL and/or CTL formulas, from control behaviors, and automatically verifies the properties in operational behaviors. The approach presented in this paper has been implemented using a number of state-of-the-art technologies and is fully functional. We conducted experimental studies to validate the feasibility of our approach. Experimental results show that our approach can correctly detect design problems in Web services. Given that testing and debugging tasks are still difficult and early detection can help address problems at design time, the techniques proposed in this paper can be extremely valuable in practice.

The encouraging results from our work so far are stimulating a number of further researches to extend the current prototype. Firstly, we will investigate the restrictions that transactional properties, e.g., pivot and retriable, would put on the behaviors of a Web service. For instance, if the translation service is declared as pivot, then the number of *sync* messages that the operational behavior could submit to the control behavior needs to be limited to one. As a result, this will make the use of other *sync* messages or *syncreq* messages prohibited. This prohibition has to be reflected on the conversation sessions along with their respective sequences of conversational messages. Secondly, we also plan to extend our approach to the composition of Web services, an important technology for integrating complex cloud applications. Finally, we will conduct more experiments to further study the performance (e.g., scalability and usability) of our system for services verification.

Acknowledgment

Quan Z. Sheng's work has been partially supported by Australian Research Council (ARC) Discovery Grant DP0878367. The authors thank the anonymous reviewers for their valuable feedback on this work.

References

[1] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, et al., Web Service Choreography Interface (WSCI) 1.0. <http://www.w3.org/TR/wsci/>.
[2] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Communication of the ACM 53 (4) (2010) 50–58.
[3] A. Banerji, C. Bartolini, D. Beringer, V. Chopella, K. Govindarajan, A. Karp, H. Kuno, M. Lemon, G. Pogossiants, S. Sharma, et al., Web Services Conversation Language (WSCL) 1.0. <http://www.w3.org/TR/wscl10/>.
[4] B. Benatallah, F. Casati, F. Toumani, Web service conversation modeling: a cornerstone for e-business automation, IEEE Internet Computing 8 (1) (2004) 46–54.
[5] Beatrice Berard, Michel Bidoit, Alain Finkel, Francois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, Systems and Software Verification. Springer Link, 2001.
[6] S. Bhiri, O. Perrin, C. Godart, Ensuring required failure atomicity of composite web services, in: Proc. of the 14th International World Wide Web Conference (WWW 2005), Chiba, Japan, 2005.
[7] S. Bourne, C. Szabo, Q.Z. Sheng, Ensuring well-formed conversations between control and operational behaviors of web services, in: Proc. of the 10th International Conference on Service-Oriented Computing (ICSOC 2012), Shanghai, China, November 2012.
[8] G. Castagna, N. Gesbert, L. Padovani, A theory of contracts for web services, ACM Transactions on Programming Languages and Systems 31 (5) (2009) 19.
[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, Nusmv 2: an opensource tool for symbolic model checking, in: Proc. of International Conference on Computer-Aided Verification (CAV 2002), Copenhagen, Denmark, July 2002.
[10] E.M. Clarke, O. Grumberg, D. Peled, Model Checking, MIT Press, 1999.
[11] J. Domingue, D. Fensel, Toward a service web: integrating the semantic web and service orientation, IEEE Intelligent Systems 23 (1) (2009) 86–88.
[12] X. Fu, T. Bultan, J. Su, Analysis of interacting BPEL web services, in: Proc. of the 13th International World Wide Web Conference (WWW 2004), New York, NY, USA, May 2004, pp. 621–630.
[13] X. Fu, T. Bultan, J. Su, Synchronizability of conversations among web services, IEEE Transactions on Software Engineering 31 (12) (2005) 1042–1055.
[14] D. Harel, A. Naamad, The STATEMATE semantics of statecharts, ACM Transactions on Software Engineering and Methodology 5 (4) (1996) 293–333.
[15] G.J. Holzmann, The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, 2003.
[16] R. Hull, J. Su, Tools for composite web services: a short overview, ACM SIGMOD Record 34 (2) (2005) 86–95.
[17] Y. Kambayashi, H.F. Ledgard, The separation principle: a programming paradigm, IEEE Software 21 (2) (2004) 78–87.
[18] M. Kovas, J. Bentahar, Z. Maamar, H. Yahyaoui, Formal verification of conversations in composite web services, in: Proc. of the 8th International Conference on Software Methodologies, Tools and Techniques (SoMeT 2009), Prague, Czech Republic, 2009.
[19] S. Kripke, Semantical considerations on modal logic, Acta Philosophica Fennica 16 (1963) 83–94.
[20] J. Li, J. Huai, C. Hu, Y. Zhu, A secure collaboration service for dynamic virtual organizations, Information Sciences 180 (17) (2010) 3086–3107.
[21] X. Li, Y. Fan, Q.Z. Sheng, Z. Maamar, H. Zhu, A petri net approach to analyzing behavioral compatibility and similarity of web services, IEEE Transactions on Systems, Man, and Cybernetics, Part A 41 (2) (2011) 1–12.
[22] M. Little, Transactions and web services, Communications of the ACM 46 (10) (2003) 49–54.
[23] Z. Maamar, N. Narendra, D. Benslimane, S. Subramanian, Policies for context-driven transactional web services, in: Proc. of International Conference on Advanced Information Systems Engineering (CAiSE 2007), Trondheim, Norway, June 2007.
[24] Z. Maamar, Q.Z. Sheng, B. Benatallah, Towards a conversation-driven composition of web services, Web Intelligence and Agent Systems 2 (2) (2004) 145–150.
[25] A.A. Nyre, M.G. Jaatun, Privacy in a semantic cloud: what's trust got to do with it? in: Proc. of the First International Conference on Cloud Computing (CloudCom 2009), Beijing, China, December 2009.
[26] E.K. Ozorhan, E.K. Kuban, N.K. Cicekli, Automated composition of web services with the abductive event calculus, Information Sciences 180 (19) (2010) 3589–3613.
[27] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: state of the art and research challenges, IEEE Computer 40 (11) (2007) 38–45.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016

[28] S. Paquette, P.T. Jaeger, S.C. Wilson, Identifying the security risks associated with governmental use of cloud computing, Government Information Quarterly 27 (3) (2010) 245–253.
[29] Anders Ravn, Jiri Srba, Saleem Vighio, A formal analysis of the web services atomic transaction protocol with UPPAAL, in: Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation, 2010, pp. 579–593.
[30] B. Rochwerger et al, Reservoir-when one cloud is not enough, IEEE Computer 44 (3) (2011) 44–51.
[31] Q.Z. Sheng, Z. Maamar, H. Yahyaoui, J. Bentahar, K. Boukadi, Separating operational and control behaviors: a new approach to web services modeling, IEEE Internet Computing 14 (3) (2010) 68–76.
[32] J. Simmonds et al, Runtime monitoring of web service conversations, IEEE Transactions on Services Computing 2 (3) (2009) 223–244.
[33] S. Srinivasan, V. Getov, Navigating the cloud computing landscape-technologies, services, and adopters, IEEE Computer 44 (3) (2011) 22–23.
[34] J. Su, T. Bultan, X. Fu, Web service interactions: analysis and design, in: Proc. of the Fifth International Conference on Computer and Information Technology (CIT 2005), Shanghai, China, September 2005.
[35] K. Verma, A.P. Sheth, Semantically annotating a web service, IEEE Internet Computing 11 (2) (2007) 83–85.
[36] M. Vieria, N. Laranjeiro, H. Madeira, Benchmarking the robustness of web services, in: Proc. of the 13th International Symposium on Pacific Rim Dependable Computing (PRDC 2007), Melbourne, Australia, December 2007.
[37] Y. Wei, M.B. Blake, Service-oriented computing and cloud computing: challenges and opportunities, IEEE Internet Computing 14 (6) (2010) 72–75.
[38] W. Yang, S. Tang, A solution for web services transaction, in: Proc. of the 1st International Conference on Hybrid Information Technology (ICHIT 2006), Jeju Island, Korea, November 2006.
[39] L. Yao, A novel approach to automatic verification of web service design, Masters Thesis, School of Computer Science, The University of Adelaide, 2010.
[40] Q. Yu, X. Liu, A. Bouguettaya, B. Medjahed, Deploying and managing web services: issues, solutions, and directions, The VLDB Journal 17 (3) (2008) 537–572.
[41] Q. Zhang, L. Cheng, R. Boutaba, Cloud computing: state-of-the-art and research challenges, Journal of Internet Services and Applications 1 (1) (2010) 7–18.

Please cite this article in press as: Q.Z. Sheng et al., Behavior modeling and automated verification of Web services, Inform. Sci. (2012), http://dx.doi.org/10.1016/j.ins.2012.09.016