Web Service Interaction Modeling and Verification Using Recursive Composition Algebra

Gopal N. Rai¹², G. R. Gangadharan¹, Vineet Padmanabhan², Rajkumar Buyya³  
¹University of Hyderabad, Hyderabad, India  
²IDRBT, Hyderabad, India  
³University of Melbourne, Melbourne, Australia

The design principle of composability among Web services is one of the most crucial reasons for the success and popularity of Web services. However, achieving error-free automatic Web service composition is still a challenge. In this paper, we propose a recursive composition based modeling and verification technique for Web service interaction. The application of recursive composition over a Web service with respect to a given set of Web services yields a recursive composition interaction graph (RCIG). In order to capture the requirement specifications of a Web service interaction scenario, we propose recursive composition specification language (RCSL) as a requirement specification language. Further, we employ the proposed RCIG as an interpretation model to interpret the semantics of a RCSL formula. Our verification technique is based on the generation and analysis of all possible interaction patterns. The key advantages of the proposed approach are: (i) it does not require explicit system modeling as in model checking based approaches, (ii) it captures primitive characteristics of Web service interaction patterns, such as recursive composition, sequential and parallel flow, etc, and (iii) it supports automatic composition of services.

Index Terms—Web service composition, Web service interaction, recursive composition, interaction modeling, interaction verification

I. INTRODUCTION

Web services are distributed and independent software modules which communicate with each other through the exchange of messages based on the XML standards [1]. Web services are categorized into basic and composite [2]. A basic Web service is self-contained and independent whereas a composite Web service is dependent on other Web services and based on the requirements, forms composition out of available services. In the context of Web services, two types of composition are possible: linear composition and recursive composition [3], [1]. In linear composition, the constituent Web services are only basic Web services, whereas in recursive composition, the constituent Web services could be basic as well as composite. The notion of recursive composition requires special attention [4] in the verification process as it is not easily tractable with classical modeling and verification schemes such as model checking and Petri net.

Modeling and verification of Web service interaction is a well-explored research area. Various solutions that are proposed for modeling and verification of Web service interactions can be classified as: aspect-based: modeling [5], [6], [7], verification [8], [9], modeling and verification [10], [11]; target-based: BPEL [12], [13], WSDL [11], [14]; and approach-based: model-based [15], [16], Petri net based [7], [17], process algebra based [18], artificial intelligence (AI) planning based [19], [6], logic based [20]. Although the existing solutions are promising, core techniques adopted in the solutions do not capture all the required characteristics of Web service interaction verification as they were proposed natively for different scenarios and applications. Key issues associated with Web service interaction modeling and verification, focused in this paper, are described as follows: If we model a Web service interaction scenario using a generic interaction model [21], [22], it may not be suitable because a Web service cannot interact with all other available Web services as invoking a service is conditional (based on input and output messages compatibility of caller and callee Web services). Further, unlike a generic interaction model, in a Web service interaction model, a participant may be dependent on other participants for its replies. A message sequence chart (MSC) is one of the popular and classical generic interaction modeling techniques. MSCs have also been used for the verification purpose [23], [16], [24]. However, MSCs do not capture the primitive characteristics of Web service interaction patterns such as parallel and sequential interaction flows initiated at a time by a service.

WS-BPEL defines a model for describing the behavior of a business process based on interactions between the process and its partners [25]. In order to realize an automatic and dynamic Web service composition, a feasible way is to generate a composite service or a BPEL file automatically and on the fly as per requirement. However, automatic generation of BPEL is not feasible unless we have a technique for automatic knowledge extraction about Web services. For a service, its WSDL document [26] is a source to extract the knowledge about the service, but a WSDL file does not provide underlying implementation details and logic. Due to the lack of this knowledge, the complete and comprehensive verification of Web service interaction is not possible [27]. WSDL and BPEL documents consist of several built-in features. If these built-in features are richer, the accompanying coding or logic effort for the verification process becomes less. If a verifier could know interaction patterns in advance, it would be easy to find out the possible undesired interaction patterns. In a composition
hierarchy, composite services may exist at several levels. In order to support the full automation, no composite service should be bound with the constituent services before run-time [2].

The Kripke model has been a prominent model to interpret formulas written in temporal logics, thus comprises fundamental part of model checking [28]. In a model checking based verification technique, a set of labelled transition rules governs the transitions from one world (Kripke node) to another. However, in the context of Web services, the underlying philosophy of a transition from one service to another is different than the model checking. For instance, in the context of Web service interaction, the communication messages are considered as propositions [10]. The truth value of a message infers whether the message is communicated or not. Once truth value for a message is set ‘true’, it cannot be altered to ‘false’ at a subsequent time instance. Moreover, model checking does not support modeling subtleties regarding Web service interaction such as automatic discovery of Web services [29].

In order to overcome the mentioned problems, in this paper, we employ our previously proposed algebraic model for Web services namely, the Recursive Composition Algebra (RCA) [14] (with several modifications). This paper proposes a complete framework for modeling and verification of the Web service interaction and makes the following key contributions:

- A recursive composition based modeling technique for the Web service interaction: This technique generates a recursive composition interaction graph (RCIG) that works as an interpretation model. We studied the feasibility for implementability of the RCIG and found that it is completely implementable in the real-time scenarios.
- A requirement specification language: We propose a specification language namely, recursive composition specification language (RCSL) that is expressive enough to capture requirements of Web service interaction scenarios and completely interpretable on the RCIG model.
- A verification technique - We propose a verification technique based on the possible trace phenomenon and outline the fundamental differences with possible world phenomenon. This verification technique employs RCIG as its interpretation model and RCSL as its specification language.

We implemented our proposed framework of Web service interaction modeling and verification using Java programming language. Given a set of WSDL documents of candidate services, the framework accepts the following inputs:

- An input message \(I_p\) or a service name \(w_i\) or an input-service tuple \((w_i, I_p)\)
- A specification formula \(\phi\) written in RCSL

Provision of an input \(I_p\) or \(w_i\) or \((w_i, I_p)\) generates a RCIG (say \(M\)) for interactive trace visualization and performance analysis, whereas provision of a RCSL formula (say \(\phi\)) triggers verification process \(\models M, \phi\) along with trace visualization. If model \(M\) does not satisfy \(\phi\) (\(M \nmodels \phi\)), counter trace \(T\) is also generated. In the implementation, a RCIG is generated automatically using GraphViz tool by invoking the system level commands internally.

The rest of the paper is organized as follows. Section II presents our proposed algebraic modeling of Web service interaction. Formation of recursive composition interaction graph and its implementation are discussed in Section III. Verification approach based on possible trace phenomenon is described in Section IV. Section V provides implementation details and feasibility analysis of the RCIG. Section VII investigates the relevant works followed by the advantages and limitations of our proposed approach with possible future works in Section VIII.

II. Algebraic Modeling of Web Service Composition

In this section, we present complete description of modified RCA with its algebraic properties and computability analysis. In comparison to the previous version of RCA [14], current version consists of two key modifications: (1) A single composition operation instead of previously defined two composition operations and (2) Introduction of a term service-input tuple and based on it, we redefine the operators: conditional successor, restrictive successor, and recursive composition.

Let \(W = \{w_1, w_2, w_3, \ldots, w_m, \epsilon\}\) be a finite set of available Web services, where \(\epsilon\) represents an empty Web service. An empty Web service does not invoke any service or perform any activity. On the basis of our proposition, we define a Web service \(w_i \in W\) as follows:

**Definition II.1** (Basic Web service). A Web service \(w_i \in W\) is a 3-tuple \((I, R, Rl)\), where \(I = \{I_1, \ldots, I_p\}\), \(p \in N\) is a finite set of input messages, that \(w_i\) accepts. \(R = \{R_1, \ldots, R_q\}\), \(q \in N\) is a finite set of response messages, that \(w_i\) produces. \(Rl\) is a service logic that maps an input message from \(I\) to the output messages in \(R (Rl \subseteq I \times R)\). \(w_i.I, w_i.R\), and \(w_i.Rl\) are referred as the set of input messages, the set of response messages, and the relation from \(w_i.I\) to \(w_i.R\) in \(w_i\).

For a Web service, the set of input messages, the set of output messages, and relation from input message set to output message set are static and available in the respective WSDL document. We define a composite service as follows:

**Definition II.2** (Composite Web service). A composite Web service \(w_i \in W\) is a 3-tuple \((I, F, Rl)\), where \(I = \{I_1, \ldots, I_p\}\), \(p \in N\) is a finite set of input messages, that \(w_i\) accepts. \(F = \{F_1, \ldots, F_q\}\), \(q \in N\) is a finite set of forward messages, that \(w_i\) produces. \(Rl\) is a service logic that maps an input message from \(I\) to a set of forward messages in \(F (Rl \subseteq I \times 2^F)\). \(w_i.I, w_i.F,\) and \(w_i.Rl\) are referred as the set of input messages, the set of forward messages, and the relation (called as service logic) from \(w_i.I\) to \(w_i.F\) in \(w_i\).

A. Operators (successor, composition, and recursive composition)

**Definition II.3** (Absolute successor). Let ‘\(\succ\)’ be a symbol to represent the successor operator. ‘\(\succ\)’ maps an element of the \(W\) to an element of the power set of the set \(W (\succ W \rightarrow 2^W)\). Given a composite Web service \(w_i \in W, S \subset W\) is a set of successor services for \(w_i\) if and only if \(\forall w_j \in S, \exists F_j \in w_i : w_i.F_j \cap w_j.I \neq \emptyset\).
The absolute successor (in short, successor) operator (≻) is an unary operator that provides services directly invokable by a composite service (we call them as successor services). The successor operator works only for a composite service as a basic service does not call other services for composition. If a service \( w_i \in W \) invokes a service \( w_j \in W \), then \( w_j \in (\succ w_i) \). If \( w_j \) is not known in advance, we write \( \succ w_i = \{ w_{i+1} \} \) unless stated otherwise. If the service \( w_i \) directly invokes a set of services (say \( \{ w_1, \ldots, w_l \} \subseteq W \) then \( \succ w_i = \{ w_1, \ldots, w_l \} \). If the service \( w_i \) does not invoke any service from the set \( W \), then \( \succ w_i = \emptyset \).

The composition of services (say, \( n \) no. of services) is the aggregation of facilities provided by the \( n \) services as a single service. Composability of a service with another service is decided by successor relation. Given a composite service \( w_i \) and its successor service \( w_j, w_j \) is always composable with \( w_i \). Let ‘⊕’ be a symbol that represents the service composition.

We define composition of services as follows:

**Definition II.4 (Service composition).** Given two Web services \( w_i, w_j \in W : w_j \in (\succ w_i) \), composition of \( w_i \) and \( w_j \) (represented as \( w_i \oplus w_j \)) yields a composite Web service \( w_k \in W \) such that

\[
\exists m \in w_i. I \left( (w_i.RI(m) = n) \land (n \in w_j.I) \right) \rightarrow \left( (m \in w_k.I) \land ((w_k.RI(m) \subseteq w_j.RI(n)) \right)
\]

A WSDL document of a composite service just provides the information on how it gets composed when required. The structural definition of a composite service, provided in its intermediate representation form (see Section III.B), decides whether a composition would be treated as parallel or sequential.

**Definition II.5 (Service-input tuple).** A tuple \( \langle w_i, I_p \rangle \) is called as a service-input tuple if and only if \( w_i \in W \) and \( I_p \in w_i.I \).

**Definition II.6 (Service-response tuple).** A tuple \( \langle w_i, R_q \rangle \) is called as a service-response tuple if and only if \( w_i \in W \) and \( R_q \in w_i.R \).

A service-input tuple is possible for basic and composite services whereas service-response tuple is possible only for basic services. A service-message tuple is a common name for both service-input and service-response tuples. \( \langle w_i, m \rangle \) is a representation for service-message tuple.

**Definition II.7 (Conditional successor).** A conditional successor (≻\(_C\)) accepts the input in the form of service-input tuple format and produces the output either in the form of service-input tuple \( \langle w_i, I_p \rangle \) or in the form of service-response tuple \( \langle w_i, R_q \rangle \). Given a tuple \( \langle w_i, I_p \rangle \), \( \langle w_j, m \rangle \) is a conditional successor of \( \langle w_i, I_p \rangle \) (written as \( \langle w_j, m \rangle \in (\succ\_C \langle w_i, I_p \rangle) \)) if and only if \( w_j \in (\succ w_i) \) and \( m \in w_i.RI(I_p) \).

Let \( \langle w_i, m_p \rangle \) and \( \langle w_j, m_q \rangle \) be two service-message tuples such that their composition \( \langle w_i, m_p \rangle \oplus \langle w_j, m_q \rangle \) is possible. Then, \( \langle w_i, m_p \rangle \oplus \langle w_j, m_q \rangle \) represents a composition chain that could participate in further composition processes as a single service. However, only the end elements of a composition chain participates in further composition process. If a service-message tuple \( \langle w_i, m_p \rangle \) composes with \( \langle w_j, m_q \rangle \) and \( \langle w_r, m_r \rangle \) in parallel, it is represented by the means of two separate composition chains: \( \langle w_i, m_p \rangle \oplus \langle w_j, m_q \rangle \) and \( \langle w_i, m_p \rangle \oplus \langle w_r, m_r \rangle \). A composition chain grows further with the attachment of other composable service-message tuples.

However, a composition with an empty service results as a tuple itself without any change \( \langle w_i, m_p \rangle \oplus e = \langle w_i, m_p \rangle \).

The conditional successor for a tuple with an empty second field (input message is not specified) behaves as an absolute successor, indicating that the service-message tuple can be replaced with the service name only. A conditional successor operator is a special case of restrictive successor operator (≻\(_R\)) representing \( \text{Domain}(\succ\_R) = \text{Domain}(\succ) \) and \( \text{Range}(\succ\_R) \subseteq \text{Range}(\succ\_C) \). We define a restrictive successor operator as follows.

**Definition II.8 (Restrictive successor).** Let \( \langle w_i, I_p \rangle \oplus \langle w_j, I_q \rangle \oplus \cdots \oplus \langle w_n, I_s \rangle \) be a composition chain and \( \langle w_x, I_r \rangle \) be a service-input tuple, then \( \langle w_x, I_r \rangle \) is a restrictive successor of \( \langle w_i, I_p \rangle \oplus \langle w_j, I_q \rangle \oplus \cdots \oplus \langle w_n, I_s \rangle \) if and only if the following two conditions hold.

1. \( \langle w_x, I_r \rangle \) is a conditional successor of the composition chain \( \langle w_i, I_p \rangle \oplus \langle w_j, I_q \rangle \oplus \cdots \oplus \langle w_n, I_s \rangle \) (written as \( \langle w_x, I_r \rangle \in (\succ\_C \langle w_i, I_p \rangle \oplus \langle w_j, I_q \rangle \oplus \cdots \oplus \langle w_n, I_s \rangle) \)).

2. \( \langle w_x, I_r \rangle \) is not a constituent member of the composition chain \( \langle w_i, I_p \rangle \oplus \langle w_j, I_q \rangle \oplus \cdots \oplus \langle w_n, I_s \rangle \) (written as \( \langle w_x, I_r \rangle \notin (\langle w_i, I_p \rangle, \langle w_j, I_q \rangle, \ldots, \langle w_n, I_s \rangle) \)).

The empty first field or second field in the input argument of a restrictive successor is a special case and is treated as follows:

\[
\succ\_R \langle w_i, - \rangle \equiv \succ\_R \{ \langle w_i, I_1 \rangle, \langle w_i, I_2 \rangle, \cdots, \langle w_i, I_p \rangle \} \quad (1)
\]

where \( \{ I_1, I_2, \cdots, I_p \} = w_i.I \).

\[
\succ\_R (-, I_p) \equiv \succ\_R \{ \langle w_i, I_p \rangle, \langle w_j, I_p \rangle, \cdots, \langle w_l, I_p \rangle \} \quad (2)
\]

where \( \{ w_i, w_j, \cdots, w_l \} \in W \) such that \( I_p \in w_i.I, w_j.I, \cdots, w_l.I \).

Let ‘⊕’ be a symbol to represent recursive composition. To define recursive composition, we use restrictive successor operator (≻\(_R\)) and composition operator (⊕) as supplementary operators (defined earlier in this section).

**Definition II.9 (Recursive composition).** Recursive composition for a given service-input tuple \( \langle w_i, I_p \rangle \), where \( I_p \in w_i.I \) is defined as follows:

\[
\odot \langle w_i, I_p \rangle \triangleq \begin{cases} 
\langle w_i, I_p \rangle 
& \text{if } \succ\_R \langle w_i, I_p \rangle = \emptyset \\
\odot \{ \langle w_i, I_p \rangle \oplus (\succ\_R \langle w_i, I_p \rangle) \} 
& \text{otherwise}
\end{cases}
\]

(3)

Successor operator and recursive composition operator are having equal precedence. They possess higher precedence over the composition operator.

Various flavors of Web service algebras [18], [17], [30], [31], [32] are available in the literature. The RCA differs from these algebras in consideration of recursive composition and its applicability to the well-known problem of Web service interaction verification.
III. Recursive Composition Interaction Graph (RCIG)

A. RCIG formation

Given a set of Web services $\mathcal{W}$ and an input argument such as a message $\langle I_p \rangle$ or a service $\langle w_i \rangle$ or a service-message tuple $\langle w_i, I_p \rangle$, the application of recursive composition forms a graph. We call it as a recursive composition interaction graph (RCIG) (see definition III.1). Algorithms for generating RCIG are given in Appendix A.

**Definition III.1** (Recursive composition interaction graph (RCIG)). A RCIG is a tuple $\langle V, E \rangle$ where $V$ is a set of nodes (either in service-input format or in service-response format) and $E$ is a set of directed edges. An edge connects a node with a set of nodes $E(v_i) = U$, where $v_i \in V$ and $U \subseteq V$ if $\forall v_j \in U : v_j \in R(v_i)$.

In the literature, interactions among services are defined and handled in many ways [33], [12], [10] based on their modeling approaches. In our context, we use the term Trace to name an interaction pattern from the RCIG, and we represent it using the letter $T$. We formally define a trace as follows:

**Definition III.2** (Trace). A trace $T$ is a RCIG such that a node in the graph can have only one child utmost.

Let $\mathcal{W}$ be a set of Web services, $w_i \in \mathcal{W}$, and $\mathcal{T}_{w_i} = \{T_0, T_1, \cdots, T_n\}$ represents a set which contains all the traces generated by applying the recursive composition on $w_i$. Similarly, $T_{I_p}$ represents a set that contains all the traces generated by applying the recursive composition on $I_p$. For the sake of convenience, we always extract traces from left to right in a RCIG. We follow the concept of trace, mainly, while studying behavioral equivalence of services.

**Subtrace.** Let $T_i$ and $T_j$ be two traces. Let $N_i$ and $N_j$ be the set of nodes in $T_i$ and $T_j$, respectively. Let $R_i$ and $R_j$ be the relations that map a node to another in $T_i$ and $T_j$. Then, $T_j$ is a subtrace of $T_i$ (represented as $T_j \subseteq T_i$) if and only if $N_j \subseteq N_i$ and $R_j \subseteq R_i$.

There are two types of traces based on the termination condition as follows:

**Definition III.3** (Open Trace). An open trace is a trace that ends with a service-input tuple.

**Definition III.4** (Closed Trace). A closed trace is a trace that ends with a service-response tuple.

For a given set of Web services $\mathcal{W}$ and an input message $I_p$, if $T_{I_p}$ consists an open trace, it implies that adequate candidate services are not available in $\mathcal{W}$ to compute all the possibilities. Since an open trace is a faulty trace, it is not desirable in service composition scenarios.

There are three types of RCIG based on its formation style: service-driven, message-driven, and service-message driven. In a service-driven RCIG, a service name is the generator of the graph. The root node consists of the service name and is preceded by service-message tuples. For instance, let $w_i$ be a service name that forms a root node. Then, all immediate nodes are of the form $\langle w_i, I_p \rangle$, where $I_p \in w_i$. In a message-driven RCIG, a message name (say, $I_p$) is the generator of the graph. The root node consists of the message name and is preceded by service-message tuples such that all immediate nodes (after root node) are restrictive successor of the $I_p$. In a service-message driven RCIG, a service-message tuple (say, $\langle w_i, I_p \rangle$) is the generator of the graph. The root node consists of the service-message tuple and is preceded by service-message tuples such that all immediate nodes (after root node) are restrictive successor of the previous node.

B. Implementation of the RCIG

A WSDL document is the description of a Web service, written in XML format. A WSDL document consists of the following elements: \langle definition \rangle, \langle types \rangle, \langle message \rangle, \langle operation \rangle, \langle portType \rangle, \langle binding \rangle, \langle port \rangle, and \langle service \rangle. Listing 1 depicts an abstract structural view of a WSDL document. The \langle portType \rangle element combines multiple message elements to form a complete one-way or round-trip operation. WSDL supports four basic patterns of operations as: one-way, request-response, solicit-response, and notification. In order to support verification of completely automated and dynamic Web service composition, we use an intermediate representation (see Listing 2) that is derived from an existing WSDL structure with the following minor modifications in the \langle operation \rangle element of the WSDL document. Except the \langle operation \rangle element, the remaining structure of WSDL is not altered.

Basic and composite services differ in their actions that they take upon reception of an input message to fulfill the request. A basic service computes the output itself for an input message whereas a composite service relies on others.

Listing 1. WSDL Document Structure

```xml
<portType name="">
  <operation name="">
    <input message="/"/>
    <output message="/"/>
  </operation>
</portType>
```

Listing 2. Intermediate Representation

```xml
<portType name="/"/>
<operation name="/"/>
<input message="/"/>
<branding name="/"/>
<port name="/"/>
<service name="/"/>
```

In the case of a basic service, we adopt the similar structure of a classical WSDL document. However, instead of input-output set of messages, we propose the input-response set of messages in the \langle operation \rangle element. In the case of a composite service, within the \langle operation \rangle element, an \langle input \rangle element is preceded by a number of (forward) elements and each \langle forward \rangle element consists \langle sequential \rangle elements. A \langle sequential \rangle element is a text element that consists a message that has to be forwarded to other services. All \langle forward \rangle elements corresponding to an input message, get triggered in parallel and all sequential messages within a
(forward) element get triggered successively in the order in which they appear. (forward) element is not a text element whereas (input) and (sequential) elements are text elements. The purpose of a forward message is to discriminate streams of parallel flows from each other. Throughout the paper, in our examples, wherever it is required to provide a WSDL document for a service, we provide a fraction (only (portType) element) of that WSDL document to avoid unnecessary complex details and to support better understandability.

Now, rest of this subsection presents an implementation (using Java) for formation of recursive composition interaction graph (RCIG) by means of a classical travel agency scenario. Let \( W = TA, HB, FB, CB, EQ, \) and \( Null \) be a finite set of Web services. \( TA, HB, FB, CB, EQ, \) and \( Null \) are the abbreviations for services: travel agency, hotel booking, flight booking, car booking, Enquiry and Null, respectively. \( TA \) is a composite service, \( FB, HB, CB, \) and \( EQ \) are basic services, and \( Null \) is an empty service. Simplified and fractional WSDL documents (in the proposed format) for these Web services are given in Appendix B.

Let \( W \) and \( \langle \text{Travel\_Booking} \rangle \) be the input arguments to construct a RCIG. The RCIG (the partial depiction of the RCIG is provided as Fig. 1 and the complete depiction of the RCIG is provided in Appendix C) is generated by using RCIGFORMATION algorithm (given in Appendix A) and shows all the possible interaction patterns in \( W \) triggered by the input message \text{Travel\_Booking}. The algorithm creates a root node \( (s_0) \) labeled with the input argument \text{Travel\_Booking}. Further, it searches existence of the input argument \text{Travel\_Booking} in the input set of available services and \text{Travel\_Booking} is found only in \( TA \). On reception of this input, \( TA \) initiates three parallel traces considering \( s_1 \) as the parent node. These three traces begin with forwarding three messages: \text{Flight\_Avail?}, \text{City\_Name}, and \text{Car\_Avail?}. These traces proceed further and stop when they map to a service-response tuple. Once a service-response tuple appears in a trace control goes back to \( TA \) and next sequential message get triggered. For instance, \( s_4 \) is a service-response tuple that comes in the path of trace \( T_0 \). Once \( s_4 \) is encountered, control goes back to \( TA \) and triggers next sequential message (\text{Flight\_Book}). In this way, it proceeds until all sequential messages from all the forward elements in \( TA \) get exhausted.

IV. WEB SERVICE INTERACTION SPECIFICATION AND VERIFICATION

The requirements of Web service interaction are classified into functional and non-functional [9]. A functional requirement describes the behavior of the system as it relates to the system’s functionality. A non-functional requirement elaborates a performance characteristic of the system such as efficiency, privacy, maintainability, etc. In this paper, we exclusively focus on functional requirements. Our proposed verification technique verifies both aspects of functional requirements: safety properties (describe what must not happen) and liveness properties (describe what must happen) [34], [35].

Throughout the paper, \( M \) represents an interpretation model and \( s_i \), where \( i \in \mathbb{N} \), represents \( i^{th} \) node or \( i^{th} \) state (depending on the context) in \( M \). The logical statement \( M, s_0 \models \phi \) infers that a state \( s_0 \) in the model \( M \) satisfies the requirement specification \( \phi \). Messages with similar name can exist in several Web services. While referring a message in particular, a service name is used as prefix and to refer a message in general, a message name itself appears without any prefix. Specifications are written using both the schemes as per requirement. For instance, let \( \phi = w_i,m_p \rightarrow m_q \) be a specification formula. In \( \phi \), \( w_i,m_p \) refers a message \( m_p \) in \( w_i \) and \( m_q \) is a message name in general. The specification formula \( \phi \) infers that if \( m_p \) is triggered from the service \( w_i \) then \( m_q \) will be triggered eventually.

A. Model for interpretation of semantics of specification formula

The proposed recursive composition interaction graph (RCIG) is employed as a model for interpreting requirement specifications formula. The interaction between two Web services can be anticipated very easily with the help of RCIG as it explores all possible interactions. A RCIG is a graph and each branch from the root to a terminal node is considered as a trace. An interaction pattern evolves with time. However, time ordering cannot be established between two nodes that belong to two different traces in a RCIG.

In the context of Web service interaction, the communication messages are considered as propositions [10]. The truth value of a message infers whether the message is communicated or not. For instance, if \( w_i,I_p = \top \), then the Web service \( w_i \) has communicated the message \( I_p \), otherwise not. In a trace, once truth value for a message is set to “true”, it cannot be altered to “false” at a subsequent time step.
B. Specification language

In order to specify the requirements regarding Web services interactions, we propose a specification language recursive composition specification language (RCSL).

**Definition IV.1 (Syntax of RCSL).** RCSL has the following syntax given in Backus-Naur form:

\[ \phi ::= \top \mid \bot \mid p \mid (\neg \phi) \mid (\phi \land \phi) \mid (\phi \lor \phi) \mid (\phi \rightarrow \phi) \mid (\phi \cup \phi) \mid A\phi \mid E\phi \]

where \( p \) is any propositional atom from some set of atoms and each occurrence of \( \phi \) to the right of ::= stands for any already constructed formula. \( \top \) and \( \bot \) are well formed formulas “the tautology” and “the falsum” respectively. \( \neg \), \( \land \), \( \lor \), and \( \rightarrow \) are sentential connectives and be used in their usual meaning. \( \cup \) is a temporal modality called until. \( A \) and \( E \) are path quantifiers. \( \phi \) stands for all paths and \( E \phi \) stands for all paths at least one path.

Negation symbol ‘\( \neg \)’ binds most tightly. Next in the order comes \( \cup \) that binds more tightly than \( \lor \) and \( \land \), and the latter two bind more tightly than \( \rightarrow \). Though RCSL consists of the constructs from both LTL and CTL, neither RCSL \( \subseteq \) LTL nor RCSL \( \subseteq \) CTL. Let \( M = (S,\rightarrow,L) \) be a RCSL model, \( T = s_0,\cdots,s_n \) be a trace in \( M \), and \( n(T) \) is a collection of all nodes in a trace \( T \). \( s_i \models p \) means that a node \( s_i \) consists of the proposition \( p \). The satisfaction relation \( \models \) (explaining whether \( T \) satisfies a RCSL formula) is defined as follows:

1. \( T \models \top \) (\( \top \) is always true).
2. \( T \models \bot \) (\( \bot \) is always false).
3. \( T \models p \) iff \( \exists s_i \in n(T) : s_i \models p \)
4. \( T \models \neg \phi \) iff \( T \not\models \phi \)
5. \( T \models \phi_1 \land \phi_2 \) iff \( T \models \phi_1 \) and \( T \models \phi_2 \)
6. \( T \models \phi_1 \lor \phi_2 \) iff \( T \models \phi_1 \) or \( T \models \phi_2 \)
7. \( T \models \phi_1 \rightarrow \phi_2 \) iff \( s_i, s_j \in n(T) : (s_i \models \phi_1 \land s_j \models \phi_2) \Rightarrow i > j \)
8. \( T \models \phi_1 \cup \phi_2 \) iff \( \phi_1 \) is a negative literal of the form \( \neg p \) and \( \left( (s_i, s_j \in n(T) : (s_i \models p) \land (s_j \models \phi_2) ) \Rightarrow i > j \right) \)
9. \( T \models E\phi_1 \) iff \( T_i \models \phi_1 \) for all \( i \geq 1 \)
10. \( T \models A\phi_1 \) iff \( T_i \models \phi_1 \) there exists \( i \geq 1 \)

**Difference between temporal logic and RCSL.** Temporal logic is a formal system for reasoning about time whereas RCSL reasons about possible Web service interaction patterns and verifies whether an interaction pattern is possible to be formed or not with the available services. There is a fundamental difference in motivation for utilizing any of them. The specification requirements are the key costs to opt a language. In linear temporal logic, there is an implicit universal quantification over the computations - the paths in state space. RCSL uses both universal and existential quantifiers explicitly, but does not use temporal operators \( X \) (next), \( F \) (finally), and \( G \) (globally). RCSL does not require \( X, F \), and \( G \) because its interpretation model RCIG is a finite and acyclic graph where no proposition can be false at later stage once it becomes true. In branching-time temporal logic, universal and existential quantifiers are used as explicit prefixes to the temporal operators and use combination of temporal operators with quantifiers such as \( AF, AG, \) etc., whereas RCSL does not require the combination of temporal operators with quantifiers.

C. Verification technique

Algorithm 1 initiates the verification process. It accepts a tuple as an input argument that consists of a set of Web services (say, \( \mathcal{W} \)) and a requirement specification formula (say, \( \phi \)).

Algorithm 1 (Interaction Verification) \( (\mathcal{W}, \phi) \)

| Input: \( \mathcal{W} \) (a set of Web services), \( \phi \) (a specification formula) |
| Output: \( \mathcal{W} \vdash \phi \) or \( \mathcal{W} \not\vdash \phi \) |

1. \( P_o \leftarrowREQSPECPARSING(\phi) \) \( \triangleright \) calling Algorithm 2
2. \( FLAG \leftarrow TRUE \)
3. Integer \( i, j, p, t \)
4. \( w_j \cdot i : \) set of all input messages in \( w_j \)
5. \( A_\phi : \) set of atoms in \( \phi \)
6. for all \( \alpha_i \in A_\phi \) do
7. for all \( \alpha_j \in \mathcal{W} \) do
8. if \( \alpha_i \in w_j \) then
9. \( I_p \leftarrow \alpha_i \)
10. \( M \leftarrow RCIGFORMATION(\mathcal{W}, I_p) \) \( \triangleright \) is a model formed by RCIGFORMATION algorithm
11. for all trace \( T_i \in M \) do
12. \( FLAG \leftarrow INTERPRETATION(P_o, T_i) \) \( \triangleright \) calling Algorithm 3
13. end if
14. end for
15. end for
16. end for
17. if \( FLAG = TRUE \) then
18. \( \mathcal{W} \vdash \phi \) \( \triangleright \) available services satisfy the specification formula
19. else
20. \( \mathcal{W} \not\vdash \phi \) \( \triangleright \) available services do not satisfy the specification formula
21. end if

Further, Algorithm 1 collects all the atoms from \( \phi \) in the set \( A_\phi \) and observes whether an atom (say, \( \alpha_i \)) belongs to an input set of a service from the set \( \mathcal{W} \). If \( \alpha_i \) is found in the input set of a service, Algorithm 1 invokes RCIGFORMATION algorithm by supplying arguments \( \mathcal{W} \) and \( I_p \). After completing the processing of the input arguments, RCIGFORMATION algorithm provides a RCIG model \( M \) rooted at \( I_p \). Then, Algorithm 1 extracts the traces \( T_i \) \( (t \in \mathcal{N}) \) from the model \( M \) one by one and calls Algorithm 3 for further processing by passing the arguments \( P_o \) and \( T_i \). Then, Algorithm 3 interprets \( P_o \) on the provided trace \( T_i \) and results as TRUE or FALSE, based on its computation. In case, if the result is TRUE, trace \( T_i \) is a witness example, otherwise trace \( T_i \) is a counter example. Algorithm 3 decomposes the AST \( P_o \) in subtrees recursively and divide also the trace \( T_i \) recursively corresponding to subtrees until unit-level-subtrees (smallest non-trivial subtrees) are achieved. Now, the function \( P\text{I}nterpretation(subtree, trace) \) in Algorithm 3 interprets the unit-level-subtrees over corresponding dividend of the trace. Once these subtrees are satisfied in the trace, satisfaction of the higher level subtrees will be investigated in bottom to top fashion.

**Example IV.1.** Let the RCIG depicted in Fig. 1 be an inter-
Algorithm 2 ReqSpecParsing(φ)

Input: φ: a requirement specification formula written in RCSL
Output: P_φ: an abstract syntax tree for φ

1: int i = 0
2: String Id_i ← NULL
3: String Token ← NULL
4: String nextToken ← NULL
5: String prevToken ← NULL
6: for all Token ∈ φ do
7:   TokenSet ← Token
8: end for
9: Token ← TokenSet(i)
10: while [TokenSet] ≠ 1 AND Token ≠ Id do
11:   if Token = “~” then
12:     if nextToken = “)” then
13:       PARENTHESIS(nextToken)
14:     else
15:       Id_i ← nextToken
16:       Replace nextToken with Id_i in φ
17:     end if
18:     Replace “~Id_i” with “Id_i+1” in φ
19:     i ← i + 1
20:   else if Token = “|” then
21:     while nextToken = “)” do
22:       Token ← nextToken
23:     end while
24:     PARENTHESIS(TOKEN)
25:   else if Token = “∪” then
26:     FUNC(TOKEN)
27: else if Token = “∨” then
28:     FUNC(TOKEN)
29: else if Token = “∧” then
30:     FUNC(TOKEN)
31: else if Token = “→” then
32:     FUNC(TOKEN)
33:   else if Token = “p” then
34:     Replace ‘p’ with ‘Id_i’ in φ
35:     i ← i + 1
36: else if Token = “)” then
37:     Skip move to next token
38:   end if
39: end while
40: end function
41: function PARENTHESIS(Value)
42:   String Token ← Value
43:   repeat
44:     φ_Temp ← Token
45:     Token ← nextToken
46:     until Token ≠ “)"
47:     φ_Temp ← Token
48:     Replace φ_Temp with Id_i in φ
49:     i ← i + 1
50: end function
51: function FUNC(TOKEN)
52:   Id_i ← prevToken Token
53:   Replace prevToken Token nextToken with “Id_i” in φ
54:   i ← i + 1
55: end function
56: AST(Id_i−1) → print the abstract syntax tree for φ

Fig. 2. AST for the specification formula φ₁ (Eqn. 4)

Now, a verifier has to verify the model M against the formula φ₁. According to the verification technique, traces in model M are considered one by one for verification. Let us consider that a trace T_i (shown in Fig. 3) from the model M (T_i = s_0, ⋯, s_24) has to be verified against φ₁. The AST (P_φ₁) for φ₁ is given in Fig. 2.

Fig. 3. A trace from the RCIG in Fig. 1

Fig. 4. Subtree decomposition of the AST given in Fig. 2

According to Algorithm 1, P_φ₁ gets recursively decomposed in its respective subtrees recursively until unit-level-subtrees are achieved (see Fig. 4) (further no decomposition is possible). Initially, P_φ₁ is decomposed into subtrees: ST₁ and ST₂. ST₁ is decomposed into ST₁₁ and ST₁₂ that are unit-level-subtrees. Therefore, they cannot be decomposed further. ST₂ is further decomposed into: ST₂₁ and ST₂₂ and so on. Once the given formula φ is completely decomposed in its constituent unit-level-subtrees, decomposition process stops.

Now, ST₁ → ST₂ will be interpreted on the trace T_i. The subtree ST₁ represents a subformula Flight_Yes ∧ Flight_Book that is satisfied in the subtrace s₀, ⋯, s₅. Then, we divide the trace T_i into subtraces: s₀, ⋯, s₅ and s₇, ⋯, s₂₄. After this, the subtree ST₂ ::= ST₂₁ ∨ ST₂₂ (Flight_Booked ∨ ¬ Hotel_Booked ∨ Flight_Booked) gets interpreted over subtrace s₇, ⋯, s₂₄. Since the subtrace satisfies ST₂₁, ST₂ ::= ST₂₁ ∨ ST₂₂ becomes satisfied. Consequently, the requirement specification φ₂ = (Flight_Yes ∧ Flight_Book) → (Flight_Booked ∨ ¬ Hotel_Booked ∨ Flight_Booked) is satisfied in the trace T_i. In the similar way, we check satisifiability of φ₁ over every trace in the model M and find satisfied. Hence, M |= φ₁

V. IMPLEMENTATION AND ANALYSIS

A. Implementation

In this subsection, we describe a prototype implementation of our proposed approach for verifying the specifications written in the RCSL against the set of available Web services. The implementation and experiments conducted have shown that the ideas proposed in this paper are realizable using existing technologies. Fig. 5 shows the high-level architecture of our prototype system, which has been implemented in Java and is based on technologies such as XML, SOAP, and WSDL. Fig. 5 consists of four modules namely, specification formula parsing, RCIG and trace generation, intermediate form conversion, and semantical interpretation. All modules are detailed as follows:
1) **Specification formula parsing**: This module receives a requirement specification formula from the verifier and processes it using Algorithm 2. Syntax checking is performed at first. Thereafter, it makes an abstract syntax tree (AST) out of the given formula. Generated AST is decomposed into its constituent subtrees until unit-level-subtrees are achieved. Finally, unit-level-subtrees are provided to the module semantical interpretation.

2) **Intermediate form conversion**: This module also receives the specification formula and discovers the set of relevant services from the available ones. Then, it retrieves their WSDL documents and makes duplicate (local) copy of WSDL documents and modifies them by adding two tags: sequential and parallel. Modified WSDL documents work as intermediate representation and are provided to the module RCIG and trace generator for further processing.

3) **RCIG and trace generation**: This module receives the set of modified WSDL documents along with an input (a service-input tuple or a message name or a service name). The input is provided by Algorithm 1. Once an input and the set of modified WSDL documents are available, it forms a RCIG using the algorithm mentioned in Appendix A. Further, it optimizes the generated RCIG by removing redundant subtraces and supplies the traces to the module semantical interpretation.

4) **Semantical interpretation**: This module verifies whether a given trace interprets the semantics of a given subtree (subformula). If the trace interprets the semantics, the module produces the witness example, otherwise, the module produces the counter example.

In addition to automation, our implementation also supports dynamic availability of services. A Web service verification framework where the verifier has to decide the participant services in advance (with or before specifying the requirement) does not support dynamic availability of services. However, in
Algorithm 3 \textsc{interpretation}(P_0, T)

\begin{algorithmic}
\STATE \textbf{Input}: $P_0$ (parse tree) and $T$ (trace)
\STATE \textbf{Output}: \textsc{true} or \textsc{false}
\STATE 1: \textbf{Root} $\leftarrow \text{Root}(P_0)$
\STATE 2: $\text{LST} \leftarrow \text{LeftSubTree}(P_0)$
\STATE 3: $\text{RST} \leftarrow \text{RightSubTree}(P_0)$
\STATE 4: \textbf{if} $\text{Root} \notin \{\text{and}, \text{or}, \text{not}, \text{for}\}$ \textbf{then}
\STATE 5: \textbf{if} $\text{interpretation}(\text{Root}, T) = \text{true}$ \textbf{then}
\STATE 6: \textbf{return} \text{true}
\STATE 7: \textbf{else}
\STATE 8: \textbf{return} \text{false}
\STATE 9: \textbf{end if}
\STATE 10: \textbf{else if} $\text{Root} = \text{\{for\}}$ \textbf{then}
\STATE 11: \textbf{if} $\text{interpretation}(\text{LST}, T) = \text{true}$ \textbf{then}
\STATE 12: \textbf{return} \text{true}
\STATE 13: \textbf{else}
\STATE 14: \textbf{return} \text{true}
\STATE 15: \textbf{end if}
\STATE 16: \textbf{else if} $\text{Root} = \text{\{not\}}$ \textbf{then}
\STATE 17: \textbf{if} $\text{interpretation}(\text{LST}, T) = \text{true}$ \textbf{and} $\text{interpretation}(\text{RST}, T) = \text{true}$ \textbf{then}
\STATE 18: \textbf{return} \text{true}
\STATE 19: \textbf{else}
\STATE 20: \textbf{return} \text{true}
\STATE 21: \textbf{end if}
\STATE 22: \textbf{else if} $\text{Root} = \text{\{or\}}$ \textbf{then}
\STATE 23: \textbf{if} $\text{interpretation}(\text{LST}, T) = \text{true}$ \textbf{or} $\text{interpretation}(\text{RST}, T) = \text{true}$ \textbf{then}
\STATE 24: \textbf{return} \text{true}
\STATE 25: \textbf{else}
\STATE 26: \textbf{return} \text{false}
\STATE 27: \textbf{end if}
\STATE 28: \textbf{else if} $\text{Root} = \text{\{and\}}$ \textbf{then}
\STATE 29: \textbf{if} $\text{interpretation}(\text{LST}) = \text{true}$ \textbf{then}
\STATE 30: \textbf{return} \text{true}
\STATE 31: \textbf{else}
\STATE 32: \textbf{for all} node $n \in T$ \textbf{do}
\STATE 33: \textbf{concatenating} $n$ to the existing sequence of nodes in Temp
\STATE 34: \textbf{if} $\text{interpretation}(\text{LST}, n) = \text{true}$ \textbf{then}
\STATE 35: \textbf{return} \text{true}
\STATE 36: \textbf{break}
\STATE 37: \textbf{end if}
\STATE 38: \textbf{end for}
\STATE 39: \textbf{return} \text{true}
\STATE 40: \textbf{else if} $\text{flag} = \text{false}$ \textbf{and} $\text{interpretation}(\text{RST}, T) = \text{false}$ \textbf{then}
\STATE 41: \textbf{return} \text{true}
\STATE 42: \textbf{else if} $\text{flag} = \text{false}$ \textbf{then}
\STATE 43: \textbf{return} \text{true}
\STATE 44: \textbf{else}
\STATE 45: \textbf{return} \text{false}
\STATE 46: \textbf{end if}
\STATE 47: \textbf{else if} $\text{Root} = \text{\{if\}}$ \textbf{then}
\STATE 48: \textbf{if} $\text{interpretation}(\text{RST}, T) = \text{true}$ \textbf{then}
\STATE 49: \textbf{return} \text{true}
\STATE 50: \textbf{for all} node $n \in T$ \textbf{do}
\STATE 51: $\text{Temp} \leftarrow n$
\STATE 52: \text{concatenating} $n$ to the existing sequence of nodes in Temp
\STATE 53: \textbf{if} $\text{interpretation}(\text{LST}', \text{Temp}) = \text{true}$ \textbf{then}
\STATE 54: \textbf{return} \text{true}
\STATE 55: \textbf{end if}
\STATE 56: \textbf{else}
\STATE 57: \textbf{return} \text{false}
\STATE 58: \textbf{end if}
\STATE 59: \textbf{else if} $\text{interpretation}(\text{RST}, T) = \text{true}$ \textbf{then}
\STATE 60: \textbf{return} \text{false}
\STATE 61: \textbf{else}
\STATE 62: \textbf{return} \text{true}
\STATE 63: \textbf{end if}
\STATE 64: \textbf{else}
\STATE 65: \textbf{function} $\text{interpretation}(\text{p}, \text{T})$
\STATE 66: \text{FLAG} $\leftarrow$ \text{false}
\STATE 67: \textbf{for all} node $n \in T$ \textbf{do}
\STATE 68: \text{if} $p \in L(n)$ \textbf{then}
\STATE 69: $\text{FLAG} \leftarrow \text{true}$
\STATE 70: \textbf{break}
\STATE 71: \textbf{end if}
\STATE 72: \textbf{else}
\STATE 73: \textbf{return} \text{false}
\STATE 74: \textbf{end for}
\STATE 75: \textbf{return} \text{false}
\STATE 76: \textbf{end if}
\STATE 77: \textbf{end function}
\end{algorithmic}
respectively. A number that precedes an element keyword is the count for that element in the respective service. For instance, ‘4Res 5Seq 6Fwd’ indicates that there are four response elements in $w_r$, five sequential elements in $w_s$, and six forward elements in $w_f$. Table I depicts the various observations taken for total number of nodes with respect to the increment in ⟨forward⟩, ⟨sequential⟩, and ⟨response⟩ elements.

**Effect of increasing response messages on the order of the RCIG:** From Table I, we extract the various observations taken for total number of nodes with respect to the increment in ⟨response⟩ elements while the count of ⟨forward⟩ and ⟨sequential⟩ elements are kept constant at the values 1, 2, 3, 4, 5, and 6. Based on the extracted values, Fig. 6 (split into two parts for better visibility) depicts six curves namely Seq1Fwd1, Seq2Fwd2, Seq3Fwd3, Seq4Fwd4, Seq5Fwd5, and Seq6Fwd6. Nature of the curves in Fig. 6 are linear and polynomial. Seq1Fwd1 is a line $(y = x + 4)$. Seq2Fwd2 is a polynomial of degree 2; Seq3Fwd3 and Seq4Fwd4 are polynomials of degree 3; Seq5Fwd5 and Seq6Fwd6 are polynomials of degree 4.

**Effect of increasing sequential messages on the order of the RCIG:** From Table I, we extract the various observations taken for total number of nodes with respect to the increment in ⟨sequential⟩ elements while forward and response elements are kept constant at the values 1, 2, 3, 4, 5, and 6. Based on the extracted values, Fig. 7 depicts six curves namely Res1Fwd1, Res2Fwd2, Res3Fwd3, Res4Fwd4, Res5Fwd5, and Res6Fwd6. Nature of the curves in Fig. 7 are linear and polynomial. Res1Fwd1 is a line $(y = x + 3)$. Seq2Fwd2 and Res3Fwd3 are three degree polynomials. Curves Res4Fwd4 and Res5Fwd5 are four degree polynomials. Res6Fwd6 is a five degree polynomial.

**Effect of increasing forward messages on the order of the RCIG:** From Table I, we extract the various observations taken for total number of nodes with respect to the increment in ⟨forward⟩ elements while sequential and response elements are kept constant at the values 1, 2, 3, 4, 5, and 6. Based on the extracted values, Fig. 8 depicts six curves namely Res1Seq1, Res2Seq2, Res3Seq3, Res4Seq4, Res5Seq5, and Res6Seq6. Nature of the curves in Fig. 8 is linear.

### Table I

<table>
<thead>
<tr>
<th>Messages</th>
<th># Nodes</th>
<th>Messages</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Res 1Seq 1Fwd</td>
<td>5</td>
<td>1Res 4Seq 4Fwd</td>
<td>30</td>
</tr>
<tr>
<td>2Res 1Seq 2Fwd</td>
<td>6</td>
<td>2Res 4Seq 4Fwd</td>
<td>212</td>
</tr>
<tr>
<td>3Res 1Seq 3Fwd</td>
<td>7</td>
<td>3Res 4Seq 4Fwd</td>
<td>802</td>
</tr>
<tr>
<td>4Res 1Seq 4Fwd</td>
<td>8</td>
<td>4Res 4Seq 4Fwd</td>
<td>2092</td>
</tr>
<tr>
<td>5Res 1Seq 5Fwd</td>
<td>9</td>
<td>5Res 4Seq 4Fwd</td>
<td>4370</td>
</tr>
<tr>
<td>6Res 1Seq 6Fwd</td>
<td>10</td>
<td>6Res 4Seq 4Fwd</td>
<td>8290</td>
</tr>
</tbody>
</table>

Fig. 6. Effect of Increasing Response Messages on the Order of the RCIG

In this best case (when count of forward elements are growing), the growth of the order of RCIG is linear. In the average case (when count of response elements are growing), the growth of the order of RCIG is lower degree polynomial, and in the worst case (when count of sequential elements are growing), the order of RCIG is a higher degree polynomial. However, in many practical cases, it is a lower degree polynomial.

### VI. Discussion

In this section, we present discussion on similarity of RCIG with a call graph of input/output WSDL, inclusion of same service more than once in a composition chain, and distance from the presented approach to a system useful in practice. Similar to RCIG, in literature, several graphical models based on input/output WSDL messages have been proposed for capturing recursive composition of Web services. There are mainly two categories of those graphical models: call graphs [36] and graph-based planners [37]. However, we do not use the said (existing recursive composition based) models because our requirements are different. We find many graphical models
that are suitable for discovery and composition planning [38], [19], however, our main interest is in the verification process. Our focus is on interaction verification and we are generating a RCIG based on the requirement specification given by a verifier/user. One more difficulty with the available models is that, by seeing WSDL file of a composite service, we are not able to find the sufficient details (i) how (sequentially or in parallel) a composite service is composed of its component/constituent services, and (ii) the messages a composite service is sending to its component/constituent services. Unavailability of the said details hinders the real-time implementation of automatic composition and verification process. Moreover, an abstract graphical representation is not suitable to verify the concrete requirement specification given by a user.

With our proposed modeling technique, it is possible to include a service in a composition chain more than once. However, it is not possible to include a service-input tuple more than once in a composition chain. While forming a composition chain, the proposed approach avoids composing a service-input tuple that has been already composed as a constituent service of the composition chain. If the same service-input tuple be included in a composition chain more than once, then it can lead into an indirect deadlock. For instance, let us consider a scenario where A invokes B for hotel booking, which invokes C for hotel booking, and C invokes A for hotel booking. This scenario may lead into an indirect deadlock. Explanation is given as follows: A invokes B for hotel booking can be written as ‘⟨A, Hotel_Book⟩ ⊕ ⟨B, Hotel_Book⟩’. Further, B in ‘⟨A, Hotel_Book⟩ ⊕ ⟨B, Hotel_Book⟩’ invokes C for hotel booking that can be written as ‘⟨A, Hotel_Book⟩ ⊕ ⟨B, Hotel_Book⟩ ⊕ ⟨C, Hotel_Book⟩’. Now, if C invokes A for hotel booking, it lead into an indirect deadlock written as ‘⟨A, Hotel_Book⟩ ⊕ ⟨B, Hotel_Book⟩ ⊕ ⟨C, Hotel_Book⟩ ⊕ ⟨A, Hotel_Book⟩’. That is why the proposed approach does not allow including a service-input tuple more than once in a composition chain.

Our proposed Web service interaction modeling and verification technique consists of three steps: (i) given a set of Web services, modeling of the Web service interaction, (ii) writing a requirement specification for verification, and (iii) verification of the requirement specification against the model. Though we have implemented and demonstrated that all the
steps are working correctly, there is space for sophistication of the technique from the practical perspective as follows. We perform I/O messages (mentioned in WSDL files) based matching to discover a composable service while creating a composite service. However, matching I/O messages of WSDL files is very syntactic in nature and does not capture the service logic. Due to this fact, small variation in message syntax will make look compatible services as incompatible. The similar kind of situation arise when one verifies whether a given trace interprets the semantics of a given requirement specification. In order to address the said problem, as our future work, we plan to incorporate a vocabulary in service discovery and specification formula interpretation process.

VII. RELATED WORK

Our proposition is completely focused on Web service interaction modeling and verification. We compare our work with most related works from the literature on modeling and verification of service interaction.

The problem of automatic Web service composition generation is closely related to the problem of Goal-Oriented Action Planning (GOAP) in artificial intelligence (AI) [37], [39]. In literature, several AI planning based techniques for automatic composition are available: STRIPS-based [39], PDDL-based [40], HTN-based [41], [42], etc. Though theoretically possible, they present a number of complexities in practical implementation, such as, generating and maintaining heavy amount of additional information (for instance, task library in [39]) hinders the automation process. We also use WSDL-based intermediate representation, however, in our approach, generation of the intermediate representation takes place automatically at the back-end and a verifier need not to be concerned about it. Recent AI planning based works, like [19] and [6], are better than previous proposals as they handle dynamic availability of services and domain-dependency of planning in more efficient way. Zou et al. [6] focused on search time reduction when finding a composite service from the Web service repository. In order to achieve their goal they converted the Web service repository into a planning domain. This transformation reduces the response time and improves the scalability of solving Web service composition problems. Kaldeli et al. [19] differed from other AI planning based techniques in that they used state variables rather than predicates as the basic elements for describing the worlds (in modeling phase). From the modeling perspective, we differ from [19] as, in our approach, the worlds need not to be defined or generated by the designer explicitly. Once the specification formula is available, worlds in the interpretation model get generated automatically. Moreover, unlike to our work, verification aspect of Web service interaction is not discussed in [6], [19].

Another line of work [4], [43] investigated into recursive composition of Web services. To form a cost-effective composite service, Jaiswal et al. [4] used a recursive composition based model. The model proposed in [4] is suitable for optimization, whereas our focus is interaction verification. Abrougui et al. [43] used recursive multi-agent systems to support dynamism in Web service composition. Like previous one [4], this work also supports in finding a better composition solution; their motive was not to verify an interaction specification.

Application of model checking based techniques for verification of Web service interaction is not entirely new, but still is in use because of its efficacy. Foster et al. [33] proposed a model-based technique to verify Web service compositions represented in the form of BPEL. They modeled specifications in the form of Message Sequence Charts (MSCs). Further, BPEL and MSCs were mechanically compiled into the finite state process notation (FSP). Then, verification process takes place between FSPs generated from BPEL and MSCs using trace equivalence phenomenon. Contrary to [33], Fu et al. [12] presented a Web service interaction verification scheme based on the centralized theme of conversation modeling. They specified desired conversations of a Web service as a guarded automaton. Their focus was on the asynchronous messaging and they made effort to relax the restrictions in the way of direct application of model checking. Walton [44] verified the interaction among agents participating in multi-agent Web service systems by proposing a Web service architecture and a lightweight protocol language. Further, he verified the specification properties written in the proposed language using model checking. Techniques presented in [33], [12], [44] were efficient, however, they did not deal with automation of verification process. Schlingloff et al. [45] presented an integrated technique for modeling and automated verification of Web service composition. Their modeling was based on Petri net and for correctness they employed model checking technique with alternating temporal logics. Zheng et al. [46] presented a test case generation framework for BPEL using model checkers SPIN and NuSMV. They modeled BPEL as Web service automata (WSA) and on the basis of WSA they presented their test case generation framework. Test cases were used to verify whether the implementation of Web services meet pre-specified BPEL behavior. Rossi [47] proposed a model checking algorithm for adaptive service compositions. She employed a logic-based technique for verification of security and correctness properties using modal μ-calculus. Collectively, we differ from all the said model checking based techniques [33], [12], [44], [45], [46], [47] in that our verification technique employ possible trace-based phenomenon for verification instead of classical possible-world phenomenon and explicit system modeling (specifications of the system provided by the designer) is not required.

Further, as an improvement over the previous ones, recent model-checking based verification techniques [10], [48], [49] support automation to a great extent. Bentahar et al. [49] proposed a modeling and verification technique for composite Web services. Their modeling aspect is based on separation of concerns between operational and control behaviors (interactions among Web services) of Web services. Their verification technique was model checking-based where they automatically generated Kripke model out of the given operational behavior. Similarly, Sheng et al. [48] also proposed an automated service verification approach based on the operational and control behaviors. The coordination of operational and control behaviors
at runtime was facilitated by conversational messages and their proposed automated verification technique was based on symbolic model checking. Like [49] and [48], our proposition also supports the operational and control behaviors. Operational behaviors can be captured using the RCIG model and control behavior can be specified using RCSL. In addition to that, in our approach, once control behaviors are provided by a verifier, related operational models are discovered automatically. Rai et al. [11] proposed a set partition and trace based technique for Web service composition and its verification. However, unlike to our proposition, their focus was on the control flow logic, not on the interaction between the services. El Kholy et al. [10] presented a framework to capture and verify the interactions among multi-agent based Web services. In order to capture the interactions, they proposed and use a specification language that use commitment modalities in the form of contractual obligations. Further, multi-agent commitment protocols regulated the interactions among services and engineered service compositions. Though their approach is efficient and incorporate commitment modalities, it can capture the conversation between two agents only if participant agents are known in advance (does not support automation). Moreover, it does not capture recursive composition scenarios that is done in our approach. Recently, a Petri net based formal model for verification of Web service composition [50] was proposed. However, their goal was to verify the compliance not the interaction.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we present a recursive composition based technique for modeling and verification of interactions among Web services. Given a set of Web services and an interaction specification, our goal is to verify whether the specification is being satisfied or not. We propose recursive composition interaction graph (RCIG) to model the interactions among Web services, and recursive composition specification language (RCSL) to capture the specifications about service interactions. Further, we propose a verification technique based on the interpretation of a requirement specification formula (written in RCSL) over a given interpretation model (represented as a RCIG). Recursive composition and the quest for automated composition are two important challenges that make Web service interaction verification process difficult and different from other classical verification problems. In this paper, we successfully addressed these two challenges.

Although our proposed approach is able to achieve its intended objectives, it still has two limitations: partially solved state explosion problem and non-consideration of Quality of Services (QoSs). As we have seen in Section V-B2, a RCIG grows polynomially if response messages and sequential messages grow higher. Trace merging [11] (merging of similar traces in a RCIG) is a technique that is applicable and working fine to reduce the order of the RCIG. However, more sophisticated solutions are required. Non-consideration of QoS parameters is a major limitation of our proposed approach. In a RCIG, a QoS parameter could be represented in two ways: either by labeling the edges or by providing the values in nodes. After forming a RCIG with QoS parameters, multi-objective optimization techniques could be used to compute the best possibility at runtime based on availability of services.

In our future work, apart from addressing the said limitations, we plan to make the proposed technique more designer interactive, so that a designer will have fine grained control over the modeling and verification aspects. We also plan to enhance the technique in such a way that it would be able to capture and verify more generic interaction scenarios. Further, we want to investigate the applicability of our proposition for multi-agent interaction verification, formalization of negotiation and bargaining, and modeling of enterprise mash-up.

REFERENCES
