Web Service Interaction Modeling and Verification Using Recursive Composition Algebra

Gopal N. Rai, G. R. Gangadharan, Senior Member, IEEE, Vineet Padmanabhan, and Rajkumar Buyya, Fellow, IEEE

Abstract—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. Performance evaluation results, provided in this paper, show that our proposition is implementable for the real world applications. 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

1 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 [1], [3]. 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 [6], [19], 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 [16], [23], [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 \( (I_p, w) \) or a service name \( (w) \) or an input-service tuple \( ((w), I_p) \)
- A specification formula \( (\phi) \) written in RCSL

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

The rest of the paper is organized as follows. Section 2 presents our proposed algebraic modeling of Web service interaction. Formation of recursive composition interaction graph and its implementation are discussed in Section 3. Verification approach based on possible trace phenomenon is described in Section 4. Section 5 provides implementation details and feasibility analysis of the RCIG. Section 7 investigates the relevant works followed by the advantages and limitations of our proposed approach with possible future works in Section 8.

2 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 2.1 (Basic Web service).** A Web service \( w_i \in W \) is a 3-tuple \( (I, R, RL) \), where \( I = \{ I_1, \ldots, I_p \} \), \( p \in \mathbb{N} \) is a finite set of input messages, that \( w_i \) accepts. \( R = \{ R_1, \ldots, R_q \} \), \( q \in \mathbb{N} \) 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 2.2 (Composite Web service).** A composite Web service \( w_j \in W \) is a 3-tuple \( (I, F, RI) \), where \( I = \{ I_1, \ldots, I_p \} \), \( p \in \mathbb{N} \) is a finite set of input messages, that \( w_j \) accepts. \( F = \{ F_1, \ldots, F_q \} \), \( q \in \mathbb{N} \) is a finite set of forward messages, that
produces. \( Rl \) is a service logic that maps an input message from \( I \) to a set of forward messages in \( F (RI \subseteq I \times 2^F) \). \( w_i, I, w_i, F \), and \( w_i, RI \) 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 \( w_i \).

### 2.1 Operators (Successor, Composition, and Recursive Composition)

**Definition 2.3 (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 \( w_i \in W, S \subseteq W \) 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 \( \succ \) 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 \( \{w_{i,1}, \ldots, w_i\} \subseteq W \) then \( \succ w_i = \{w_{i,1}, \ldots, w_i\} \). 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 \( \langle \rangle \) be a symbol that represents the service composition. We define composition of services as follows:

**Definition 2.4 (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 \( w_i \triangleq w_j \)) yields a composite service \( w_i \_w_j \subseteq W \) such that \( \exists m \in w_i, I ((w_i, Rl(m) = n) \land (n \in w_j, I)) \rightarrow ((m \in w_i, I) \land ((w_i, Rl(m) \subseteq w_j, Rl(n)))) \).

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 3.2), decides whether a composition would be treated as parallel or sequential.

**Definition 2.5 (Service-input tuple).** A tuple \( \langle w_i, I \rangle \) is called as a service-input tuple if and only if \( w_i \in W \) and \( I \subseteq w_i, I \).

**Definition 2.6 (Service-response tuple).** A tuple \( \langle w_i, R_p \rangle \) is called as a service-response tuple if and only if \( w_i \in W \) and \( R_p \subseteq w_i, R \).

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 2.7 (Conditional successor).** A conditional successor \( \langle \succ \rangle \) 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 \( \langle w_i, R_p \rangle \). 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: \( \langle w_i, m \rangle \in (\succ \langle w_i, I_p \rangle) \)) if and only if \( w_i \in (\succ w_i) \) and \( m \in w_i, Rl(I_p) \).

Let \( \langle w_i, m_{i,0} \rangle \) and \( \langle w_j, m_{j,0} \rangle \) be two service-message tuples such that their composition \( \langle w_i, m_{i,0} \rangle \circ \langle w_j, m_{j,0} \rangle \) is possible. Then, \( \langle w_i, m_{i,0} \rangle \circ \langle w_j, m_{j,0} \rangle \) 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 processes. If a service-message tuple \( \langle w_i, m_p \rangle \) composes with \( \langle w_j, m_q \rangle \) and \( \langle w_k, m_r \rangle \) in parallel, it is represented by the means of two separate composition chains: \( \langle w_i, m_p \rangle \circ \langle w_j, m_q \rangle \) and \( \langle w_j, m_q \rangle \circ \langle w_k, m_r \rangle \). 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 \( \langle \langle w_i, m_p \rangle \circ \langle w_j, m_q \rangle \rangle \). 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 \( \langle \succ \rangle \) providing \( \langle \succ \rangle \) representing that \( \text{Domain}(\succ) = \text{Domain}(\succ) \) and \( \text{Range}(\succ) \subseteq \text{Range}(\succ) \). We define a restrictive successor operator as follows.

**Definition 2.8 (Restrictive successor).** Let \( \langle w_i, I_p \rangle \circ \langle w_j, I_q \rangle \circ \cdots \circ \langle w_n, I_s \rangle \) be a composition chain and \( \langle w_i, I_p \rangle \) be a service-input tuple, then \( \langle w_i, I_p \rangle \) is a restrictive successor of \( \langle w_i, I_p \rangle \circ \langle w_j, I_q \rangle \circ \cdots \circ \langle w_n, I_s \rangle \) if and only if the following two conditions hold.

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

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

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

\[ \succ_R \langle w_i, \_ \rangle = \succ_R \{\langle w_i, I_1\rangle, \langle w_i, I_2\rangle, \ldots, \langle w_i, I_p\rangle\}, \]

where \( I_1, I_2, \ldots, I_p \) = \( w_i, I \).

\[ \succ_R \langle \_ , I_p \rangle = \succ_R \{\langle w_i, I_p\rangle, \langle w_j, I_p\rangle, \ldots, \langle w_i, I_p\rangle\}, \]

where \( w_i, I_1, w_j, I_1, \ldots, w_i, I \).

Let \( \langle \_ \rangle \) be a symbol to represent recursive composition. To define recursive composition, we use restrictive successor operator \( \langle \succ \rangle \) and composition operator \( \circ \) as supplementary operators (defined earlier in this section).

**Definition 2.9 (Recursive composition).** Recursive composition for a given service-input tuple \( \langle w_i, I_p \rangle \), where \( I_p \in w_i \) is defined as follows:

\[ \langle \_ \rangle (\langle w_i, I_p \rangle) \triangleq \{\langle w_i, I_p\rangle; \forall \langle \_ \rangle (\langle w_i, I_p \rangle) \} \text{ if } \succ_R \langle w_i, I_p \rangle = \emptyset \]

(3) otherwise.
Successor operator and recursive composition operator are having equal precedence. They possess higher precedence over the composition operator.

Various flavors of Web service algebras [17], [18], [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.

3 Recursive Composition Interaction Graph

3.1 RCIG Formation

Given a set of Web services \( W \) 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 (see Definition 3.1).

Definition 3.1 (Recursive composition interaction graph). A RCIG is a tuple \( (V, E) \) 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 \( (E : V \rightarrow 2^V) \) such that following condition holds \( E(v_i) = U \), where \( v_i \in V \) and \( U \subseteq V \) iff \( \forall v_j \in U : v_j \in > (v_i) \).

In the literature, interactions among services are defined and handled in many ways [10], [12], [33] 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 3.2 (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 \( T_{w_i} = \{T_0, T_1, \ldots, T_n\} \) 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, \( T_i \) is a subtrace of \( T_j \) (represented as \( T_j \subset T_i \)) if and only if \( N_j \subset N_i \) and \( R_j \subset R_i \).

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

Definition 3.3 (Open Trace). An open trace is a trace that ends with a service-input tuple.

Definition 3.4 (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 \( I_p \in w_i.I \). 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.

3.2 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. WSDL document structure

```xml
1 <portType name . . . >*
2 <operation name . . . >
3   <input message . . . />*
4:   <output message . . . />*
5 </operation>
6 </portType>
```

Listing 2. Intermediate representation

```xml
1 <portType name . . . >*
2 <operation name . . . >
3   <input message . . . />*
4:   <forward . . . >*
5:     <sequential . . . />*
6 </forward>
7 <response message . . . />*
8 </operation>
9 </portType>
```

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 \( \langle \text{operation} \rangle \) element, an \( \langle \text{input} \rangle \) element is preceded by a number of \( \langle \text{forward} \rangle \) elements and each \( \langle \text{forward} \rangle \) element consists of \( \langle \text{sequential} \rangle \) elements. A \( \langle \text{sequential} \rangle \) element is a text element that consists of a message that has to be forwarded to other services. All \( \langle \text{forward} \rangle \) elements corresponding to an input message, get triggered in parallel and all sequential messages within a \( \langle \text{forward} \rangle \) element get triggered successively in the order in which they appear. \( \langle \text{forward} \rangle \) element is not a text element whereas \( \langle \text{input} \rangle \) and \( \langle \text{sequential} \rangle \) 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 \( \langle \text{portType} \rangle \) element) of that WSDL document to avoid unnecessary complex details and to support better understandability.

Now, rest of this section presents an implementation (using Java) for formation of recursive composition interaction graph 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.

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) 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.

4 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 \( i \in N \), represents ith node or ith 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 \( \phi = w, m_p \rightarrow m_q \) be a specification formula. In \( \phi, w, m_p \) refers a message \( m_p \) in \( w \) and \( m_q \) is a message name in general. The specification formula \( \phi \) infers that if \( m_p \) is triggered from the service \( w \), then \( m_q \) will be triggered eventually.

4.1 Model for Interpretation of Semantics of Specification Formula

The proposed recursive composition interaction graph 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 \( w_i, Ip = \top \), 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.

4.2 Specification Language

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

Definition 4.1 (Syntax of RCSL). RCSL has the following syntax given in Backus-Naur form:

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

\( p \) 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. \( \rightarrow, \land, \lor, \neg \) are sentential connectives and be used in their usual meaning. \( \cup \) is a temporal modality called until. \( A \) and \( E \) are path quantifiers. \( A \) stands for for all paths and \( E \) stands for 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, \( T = s_0, \ldots, s_n \) be a trace in \( M \), and \( n(T) \) is a collection of all nodes in a trace \( T \). \( s_i \models \varphi \) means that a node \( s_i \) consists of the proposition \( \varphi \). The satisfaction relation \( \models \) (explaining whether \( T \) satisfies a RCSL formula) is defined as follows:

1. \( T \models \top \) ( \( T \) is always true).
2. \( T \not\models \bot \) ( \( T \) is always false).
3. \( T \models \psi \rightarrow \varphi \) if \( T \models \varphi \).
4. \( T \models \neg \psi \) if \( T \not\models \psi \).
5. \( T \models \phi_1 \land \phi_2 \) then \( T \models \phi_1 \land T \models \phi_2 \).
6. \( T \models \phi \lor \psi \) if \( T \models \phi \) or \( T \models \psi \).
7. \( T \models \phi_1 \rightarrow \phi_2 \) iff \( s_i, s_j \in n(T) : (s_i \models \phi_1 \land s_j \models \phi_2) \land (i < j) \).
8. \( T \models \phi_1 \lor \phi_2 \) iff \( \phi \) is a negative literal of the form \( \neg \psi \) and \( (s_i, s_j \in n(T) : s_i \models \psi \land (s_i \models \phi_2)) \rightarrow i > j \).
9. \( T \models A \phi_1 \) if \( T \models \phi_1 \) for all \( i \geq 1 \).
10. \( T \models E \phi_1 \) if \( T \models \phi_1 \) 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 factors 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.

### 4.3 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, \( W \)) and a requirement specification formula (say, \( \phi \)) written in RCSL. \( W \) is an online Web service repository available on a specific url address and \( \phi \) is provided by a verifier. Once \( \phi \) is available, Algorithm 1 calls Algorithm 2 by passing \( \phi \) as an argument. Algorithm 2 parses \( \phi \), and correspondingly it generates an abstract syntax tree (written as \( P_\phi \)) if given formula is free from syntax errors. Word token in Algorithm 2 represents a sequence of characters that can be treated as a single logical entity. Typical tokens are: 1) identifiers 2) keywords 3) operators 4) special symbols, and 5) constants.

### Algorithm 1. INTERACTION verification (\( W, \phi \))

**Input:** \( W \) (a set of Web services), \( \phi \) (a specification formula) 
**Output:** \( W \models \phi \) or \( W \not\models \phi \)

1. \( P_\phi \leftarrow \text{REOSPACING} (\phi) \) \( \triangleright \) calling Algorithm 2
2. \( \text{FLAG} \leftarrow \text{FALSE} \)
3. Integer \( i, j, p, t \)
4. \( w_j, I \leftarrow \text{set of all input messages in } w_j \)
5. \( A_\phi \leftarrow \text{set of atoms in } \phi \)
6. for all \( \alpha_i \in A_\phi \) do
7. for all \( w_j \in W \) do
8. if \( \alpha_i \in w_j, I \) then
9. \( I_p \leftarrow \alpha_i \)
10. \( M \leftarrow \text{RCIGFORMATION} (W, I_p) \) \( \triangleright M \) is a model formed by RCIGFORMATION algorithm
11. for all trace \( T_i \in M \) do
12. \( \text{FLAG} \leftarrow \text{INTERPRETATION} (P_\phi, T_i) \) \( \triangleright \) calling Algorithm 3
13. end for
14. end if
15. end for
16. end for
17. if \( \text{FLAG} \leftarrow \text{TRUE} \) then
18. \( W \models \phi \) \( \triangleright \) available services satisfy the specification formula
19. else
20. \( W \not\models \phi \) \( \triangleright \) 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 \( W \). If \( \alpha_i \) is found in the input set of a service, Algorithm 1 invokes RCIGFORMATION algorithm by supplying arguments \( W \) 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 extract the traces \( T_i (t \in N) \) from the model \( M \) one by one and calls Algorithm 3 for further processing by passing the arguments \( P_\phi \) and \( T_i \). Then, Algorithm 3 interprets \( P_\phi \) 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 \( P_\phi \) into 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 \( \text{PINTERPRETATION(subtree, trace)} \) 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 4.1

Let the RCIG depicted in Fig. 1 be an interpretation model \( M \) and \( \phi_1 \) (see Eq. (4)) be a requirement specification formula which, formally, states that in all the traces, if flight is available and booking is requested, then either flight must be booked or hotel must not be booked until flight is booked

\[
\phi_1 = A((\text{Flight\_Yes} \land \text{Flight\_Booked}) \rightarrow (\text{Flight\_Booked} \lor (\neg \text{Hotel\_Booked} \lor (\text{Flight\_Booked}))).
\]
Now, a verifier has to verify the model $M$ against the formula $\phi_1$. 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 $M$ ($T_i = s_0, \ldots, s_{21}$) has to be verified against $\phi_1$. The AST ($P_{\phi_1}$) for $\phi_1$ is given in Fig. 2.

According to Algorithm 1, $P_{\phi_1}$ gets recursively decomposed in its respective subtrees recursively until unit-level-subtrees are achieved (see Fig. 4) (further no decomposition is possible). Initially, $P_{\phi_1}$ 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 $\phi_1$ is completely decomposed in its constituent unit-level-subtrees, decomposition process stops.

Now, $ST1 \rightarrow ST2$ will be interpreted on the trace $T_i$. The subtree $ST1$ represents a subformula $Flight\_Yes \land Flight\_Book$ that is satisfied in the subtrace $s_0, \ldots, s_5$. Then, we divide the trace $T_i$ into subtraces: $s_0, \ldots, s_6$ and $s_7, \ldots, s_{21}$. After this, the subtree $ST2 ::= ST21 \lor ST22$ ($Flight\_Booked \lor (\neg Hotel\_Booked \lor Flight\_Booked)$) gets interpreted over subtrace $s_0, \ldots, s_5$. Since the subtrace satisfies $ST21$, $ST2 ::= ST21 \lor ST22$ becomes satisfied. Consequently, the requirement specification $\phi_1 = (Flight\_Yes \land Flight\_Book) \rightarrow (Flight\_Booked \lor (\neg Hotel\_Booked \lor Flight\_Booked))$ is satisfied in the trace $T_i$. In the similar way, we check satisfiability of $\phi_1$ over every trace in the model $M$ and find satisfied. Hence, $M \models \phi_1$.

### Algorithm 2: ReqSpecParsing($\phi$)

**Input:** $\phi$: a requirement specification formula written in RCLSL
**Output:** $P_{\phi}$: an abstract syntax tree for $\phi$

1: int $i = 0$
2: String $Id_i \leftarrow$ NULL
3: String $Token \leftarrow$ NULL
4: String $nextToken \leftarrow$ NULL
5: String $prevToken \leftarrow$ NULL
6: for all $Token \in \phi$ do
7:   $TokenSet \leftarrow Token$
8: end for
9: $Token \leftarrow TokenSet(i)$
10: while $[TokenSet] \neq 1$ AND $Token \neq Id$ do
11:   if $Token = \lnot'$ then
12:     if $nextToken = \lnot(' then
13:       Parenthesis($nextToken$)
14:     else
15:       $Id_i \leftarrow nextToken$
16:       Replace $nextToken$ with $Id_i$ in $\phi$
17:     end if
18:     Replace $\lnot Id_i$ with $Id_{i+1}$ in $\phi$
19:     $i \leftarrow i + 1$
20: else if $Token = \lnot'$ then
21:    while $nextToken = \lnot'$ do
22:      $Token \leftarrow nextToken$
23:    end while
24:    Parenthesis($Token$)
25: else if $Token = \lor$ then
26:    Func($Token$)
27: else if $Token = \lor$ then
28:    Func($Token$)
29: else if $Token = \rightarrow$ then
30:    Func($Token$)
31: else if $Token = \lnot'$ then
32:    Func($Token$)
33: else if $Token = \lnot'$ then
34:    $Id_i \leftarrow \lnot'$
35:    Replace $\lnot'$ with $Id_i$ in $\phi$
36:    $i \leftarrow i + 1$
37: else if $Token = Id$ then
38:    Skip
39: end if
40: end while
41: function Parenthesis($Value$)
42:   String $Token \leftarrow Value$
43: repeat
44:   $\phi_{Temp} \leftarrow Token$
45:   $Token \leftarrow nextToken$
46: until $Token \neq \lor$
47: $\phi_{Temp} \leftarrow Token$
48: Replace $\phi_{Temp}$ with $Id_i$ in $\phi$
49: $i \leftarrow i + 1$
50: end function
51: function Func($Token$)
52:   $Id_i \leftarrow prevToken Token nextToken$
53: Replace $prevToken Token nextToken$ with $Id_i$ in $\phi$
54: $i \leftarrow i + 1$
55: end function
56: $AST(Id_{i-1}) \triangleright$ print the abstract syntax tree for $\phi$
Algorithm 3. INTERPRETATION($P_\phi$, T)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Input: $P_\phi$ (parse tree) and T (trace)</td>
</tr>
<tr>
<td>2.</td>
<td>Output: TRUE or FALSE</td>
</tr>
<tr>
<td>3.</td>
<td>Root ← Root($P_\phi$)</td>
</tr>
<tr>
<td>4.</td>
<td>LST ← LeftSubTree($P_\phi$)</td>
</tr>
<tr>
<td>5.</td>
<td>RST ← RightSubTree($P_\phi$)</td>
</tr>
<tr>
<td>6.</td>
<td>if Root $\notin {\lor, \land, \rightarrow, \Rightarrow}$ then</td>
</tr>
<tr>
<td>7.</td>
<td>else</td>
</tr>
<tr>
<td>8.</td>
<td>if INTERPRETATION(Root, T) = TRUE then</td>
</tr>
<tr>
<td>9.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>10.</td>
<td>else</td>
</tr>
<tr>
<td>11.</td>
<td>Return FALSE</td>
</tr>
<tr>
<td>12.</td>
<td>end if</td>
</tr>
<tr>
<td>13.</td>
<td>else</td>
</tr>
<tr>
<td>14.</td>
<td>if INTERPRETATION(LST, T) = TRUE then</td>
</tr>
<tr>
<td>15.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>16.</td>
<td>else</td>
</tr>
<tr>
<td>17.</td>
<td>if INTERPRETATION(LST, T) = TRUE AND</td>
</tr>
<tr>
<td>18.</td>
<td>INTERPRETATION(RST, T) = TRUE then</td>
</tr>
<tr>
<td>19.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>20.</td>
<td>else</td>
</tr>
<tr>
<td>21.</td>
<td>Return FALSE</td>
</tr>
<tr>
<td>22.</td>
<td>end if</td>
</tr>
<tr>
<td>23.</td>
<td>else if Root = ‘¬’ then</td>
</tr>
<tr>
<td>24.</td>
<td>if INTERPRETATION(LST, T) = TRUE OR INTERPRETATION</td>
</tr>
<tr>
<td>25.</td>
<td>(RST, T) = TRUE then</td>
</tr>
<tr>
<td>26.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>27.</td>
<td>else</td>
</tr>
<tr>
<td>28.</td>
<td>Return FALSE</td>
</tr>
<tr>
<td>29.</td>
<td>end if</td>
</tr>
<tr>
<td>30.</td>
<td>else if Root = ‘∀’ then</td>
</tr>
<tr>
<td>31.</td>
<td>Temp ← ∅</td>
</tr>
<tr>
<td>32.</td>
<td>TTemp ← ∅</td>
</tr>
<tr>
<td>33.</td>
<td>for all node n $\in$ T do</td>
</tr>
<tr>
<td>34.</td>
<td>Temp $\cdot$ n $\triangleright$ Concatenating n to the existing sequence</td>
</tr>
<tr>
<td>35.</td>
<td>if INTERPRETATION(LST, n) = TRUE then</td>
</tr>
<tr>
<td>36.</td>
<td>FLAG ← TRUE</td>
</tr>
<tr>
<td>37.</td>
<td>BREAK</td>
</tr>
<tr>
<td>38.</td>
<td>end if</td>
</tr>
<tr>
<td>39.</td>
<td>end for</td>
</tr>
<tr>
<td>40.</td>
<td>TTemp ← {T − TTemp}</td>
</tr>
<tr>
<td>41.</td>
<td>else if INTERPRETATION(RST, TTemp) = TRUE then</td>
</tr>
<tr>
<td>42.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>43.</td>
<td>else if FLAG = FALSE AND INTERPRETATION</td>
</tr>
<tr>
<td>44.</td>
<td>(RST, TTemp) = FALSE then</td>
</tr>
<tr>
<td>45.</td>
<td>Return TRUE</td>
</tr>
<tr>
<td>46.</td>
<td>else</td>
</tr>
<tr>
<td>47.</td>
<td>if Root = ‘$\exists$’ then</td>
</tr>
<tr>
<td>48.</td>
<td>FLAG ← FALSE</td>
</tr>
<tr>
<td>49.</td>
<td>Temp ← ∅</td>
</tr>
<tr>
<td>50.</td>
<td>for all node n $\in$ T do</td>
</tr>
<tr>
<td>51.</td>
<td>Temp $\cdot$ n</td>
</tr>
<tr>
<td>52.</td>
<td>if INTERPRETATION(LST, Temp) = FALSE then</td>
</tr>
<tr>
<td>53.</td>
<td>FLAG ← FALSE</td>
</tr>
<tr>
<td>54.</td>
<td>BREAK</td>
</tr>
<tr>
<td>55.</td>
<td>end if</td>
</tr>
<tr>
<td>56.</td>
<td>end if</td>
</tr>
</tbody>
</table>

5 Implementation and Analysis

5.1 Implementation

In this section, 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 RCIG and trace generation.

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. 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 our approach, a specification formula can be written with or without explicitly mentioning the participant service names. In the case, if service names are not explicit, relevant services are discovered based on the propositions in the formula. Initially, if required, the verifier has to specify only the addresses of local or global service repositories. Later, while verification, our implementation discovers relevant services and checks current availability of services each time verification query is submitted. In other words, our implementation periodically checks the availability of the services in the given online Web service
repositories. If any change occurs in availability at a later stage (for instance, during realization of a composition plan), the plan will be updated with the change and updated plan will be shown to the user.

Also, our implementation features a user interface that supports a wide range of features such as editing and tracking of the modeled system, writing requirements specifications in RCSL, adding new services in repository by specifying their addresses, checking syntax, and starting verification. Furthermore, our proposed verification framework is fully capable to verify other systems that behave similarly to the Web service system (such as multi-agent system), provided that their system description is in desired XML format and requirement specifications are written in the RCSL.

5.2 Performance Analysis of RCIG

The order of a graph (the total number of nodes in a graph \((|G(V)|)\)) generated by a technique is an important criteria to determine the feasibility for the real world implementability of a technique. The computational resources such as time and space are directly proportional to the number of nodes. In this section, we analyze how the cardinality of forward, sequential, and response messages in services affect the order of a RCIG.

5.2.1 Experimental Setup

Let us consider a set of three services \((W := w_f, w_s, w_r)\), where \(w_f\) and \(w_s\) are composite services and \(w_r\) is a basic service. All three services accept only one input message: \(Hotel\_Available\). The composite services \((w_f, w_s)\) forward the input message to other services, whereas, the basic service \((w_r)\), upon reception of this message, replies with the available hotel booking options. Initially, each service consists of only one output message (“output message” represents to \(\{\text{forward}\}, \{\text{sequential}\}, \text{and} \{\text{response}\}\) elements). As per the requirement of experiment, we gradually increase the number of forward, sequential, and response messages in the services. At any stage of experiment, all output messages in \(w_f\) are parallel to each other, all output messages in \(w_s\) are sequential to each other, and all output messages in \(w_r\) are only response messages. We assume that \(w_f\) can invoke only \(w_s\) and \(w_r\) can invoke only \(w_f\). Since \(w_r\) is a basic service, it cannot invoke any service. This assumption facilitates us with a hierarchical invocation system of services that prevents redundancy while performing recursive composition out of \(w_f, w_s, \text{and} w_r\). All observations are taken by providing the input \((w_f, Hotel\_Available)\) to the RCIG construction process. The number of nodes are counted for the unfolded form of the RCIG without applying any heuristic to reduce the number of nodes.

5.2.2 Experimental Evaluation

There are three types of elements in the candidate services: \(\{\text{forward}\}, \{\text{sequential}\}, \text{and} \{\text{response}\}\). Observations are taken in the total number of nodes by increasing an element type. In order to support symmetrical growth, we assume that a \(\text{forward}\) element can be increased only in \(w_f\), a \(\text{sequential}\) element can be increased only in \(w_s\), and a \(\text{response}\) element can be increased only in \(w_r\). The minimum threshold for the count of all element types is one and maximum threshold is six. The maximum threshold is set to six, that is sufficiently large to capture and observe the patterns of the growth of the order of the RCIG. For taking the observations, we increase the count for an element type while keeping the count for other element types as constant at a pre-specified value.

Following are the three sets of observations corresponding to the increment of response elements, sequential elements, and forward elements respectively. Three keywords are used in the observation tables: \(\text{Res}, \text{Seq}, \text{and} \text{Fwd}\). These keywords stand for \(\{\text{response}\}\), \(\{\text{sequential}\}\), and \(\{\text{forward}\}\) elements 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 1 depicts the various observations taken for total number of nodes with respect to the increment in \(\{\text{forward}\}, \{\text{sequential}\}, \text{and} \{\text{response}\}\) elements.

<table>
<thead>
<tr>
<th>Messages</th>
<th># Nodes</th>
<th>Messages</th>
<th># Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1Res 1Seq 1Fwd 5</td>
<td>1Res 4Seq 4Fwd 50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2Res 1Seq 1Fwd 6</td>
<td>2Res 4Seq 4Fwd 242</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3Res 1Seq 1Fwd 7</td>
<td>3Res 4Seq 4Fwd 802</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4Res 1Seq 1Fwd 8</td>
<td>4Res 4Seq 4Fwd 2,042</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5Res 1Seq 1Fwd 9</td>
<td>5Res 4Seq 4Fwd 4,370</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6Res 1Seq 1Fwd 10</td>
<td>6Res 4Seq 4Fwd 8,290</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1Res 2Seq 2Fwd 14</td>
<td>1Res 5Seq 5Fwd 77</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2Res 2Seq 2Fwd 26</td>
<td>2Res 5Seq 5Fwd 622</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3Res 2Seq 2Fwd 42</td>
<td>3Res 5Seq 5Fwd 3,027</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4Res 2Seq 2Fwd 62</td>
<td>4Res 5Seq 5Fwd 10,231</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5Res 2Seq 2Fwd 86</td>
<td>5Res 5Seq 5Fwd 24,524</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6Res 2Seq 2Fwd 114</td>
<td>6Res 5Seq 5Fwd 62,202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1Res 3Seq 3Fwd 29</td>
<td>1Res 6Seq 6Fwd 110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2Res 3Seq 3Fwd 86</td>
<td>2Res 6Seq 6Fwd 1,514</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3Res 3Seq 3Fwd 197</td>
<td>3Res 6Seq 6Fwd 10,922</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4Res 3Seq 3Fwd 380</td>
<td>4Res 6Seq 6Fwd 49,142</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5Res 3Seq 3Fwd 653</td>
<td>5Res 6Seq 6Fwd 164,054</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6Res 3Seq 3Fwd 1,034</td>
<td>6Res 6Seq 6Fwd 447,890</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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 ($y = 3x + 2$). Res2Fwd2 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.**

In the best case (when count of forward 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.

### 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 [19], [38], 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 ‘\((A, \text{Hotel Book}) \oplus (B, \text{Hotel Book})\)’. Further, B in ‘\((A, \text{Hotel Book}) \oplus (B, \text{Hotel Book})\)’ invokes C for hotel booking that can be written as ‘\((A, \text{Hotel Book}) \oplus (B, \text{Hotel Book}) \oplus (C, \text{Hotel Book})\)’. Now, if C invokes A for hotel booking, it lead into an indirect deadlock written as ‘\((A, \text{Hotel Book}) \oplus (B, \text{Hotel Book}) \oplus (C, \text{Hotel Book}) \oplus (A, \text{Hotel Book})\)’. 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.

7 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 [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 [12], [33], [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 $\mu$-calculus. Collectively, we differ from all the said model checking based techniques [12], [33], [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.

8 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 to model the interactions among Web services, and recursive composition specification language 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 5.2.2, 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 solution 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


---

**Gopal N. Rai** is currently working as a Senior Assistant Professor at the Madanapalle Institute of Technology and Science, Andhra Pradesh, India. He holds the PhD degree from the University of Hyderabad and the Institute for Development and Research in Banking Technology (IDRBT), Hyderabad.

**G. R. Gangadharan** received the PhD degree from Griffith University, Australia. He is an associate professor in the School of Computer and Information Sciences, University of Hyderabad, India. His research interests focus on the interface between technological and business perspectives. He is a senior member of the IEEE and ACM.

**Vineet Padmanabhan** received the PhD degree from Griffith University, Australia. He is a professor in the School of Computer and Information Sciences, University of Hyderabad, India. His areas of interest include knowledge representation and reasoning, multi-agent systems, and logics in artificial intelligence.

**Rajkumar Buyya** received the PhD degree from the University of Melbourne, Australia. He is a professor of computer science and software engineering, future fellow of the Australian Research Council, and the director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, University of Melbourne, Australia. His areas of interest include grid computing, cloud computing, and high-performance computing. He is a fellow of the IEEE and life member of the ACM.