Web Service Interaction Modeling and Verification Using Recursive Composition Algebra

Gopal N. Rai[1,2], G. R. Gangadharan[1], Vineet Padmanabhan[2], Rajkumar Buyya[3]
[1]IDRBT, Hyderabad, India
[2]University of Hyderabad, Hyderabad, India
[3]University 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. 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**

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.

Corresponding author: G.R.Gangadharan (email:geeyaar@gmail.com)

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 ($\langle w_i, I_p\rangle$)
- A specification formula (ϕ) written in RCSL

Provision of an input (I_p or w_i or $\langle w_i, I_p\rangle$) generates a RCIG (say M) for interactive trace visualization and performance analysis, whereas provision of a RCSL formula (say ϕ) triggers verification process ($M \overset{?}{\models} \phi$) along with trace visualization. If model M does not satisfy ϕ ($M \not\models \phi$), 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 $\mathcal{W} = \{w_1, w_2, w_3, \cdots, w_m, \epsilon\}$ be a finite set of available Web services, where ϵ 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 $w_i \in \mathcal{W}$ as follows:

Definition II.1 (Basic Web service)**.** A Web service $w_i \in \mathcal{W}$ is a 3-tuple $\langle I, R, Rl\rangle$, where $I = \{I_1, \cdots I_p\}$, $p \in \mathbb{N}$ is a finite set of input messages, that w_i accepts. $R = \{R_1, \cdots, R_q\}$, $q \in \mathbb{N}$ 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 II.2 (Composite Web service)**.** A composite Web service $w_i \in \mathcal{W}$ is a 3-tuple $\langle I, F, Rl\rangle$, where $I = \{I_1, \cdots I_p\}$, $p \in \mathbb{N}$ is a finite set of input messages, that w_i accepts. $F = \{F_1, \cdots, F_q\}$, $q \in \mathbb{N}$ 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 $\mathcal{W}$ to an element of the power set of the set $\mathcal{W}$ ($\succ: \mathcal{W} \to 2^{\mathcal{W}}$). Given a composite Web service $w_i \in \mathcal{W}$, $S \subset \mathcal{W}$ is a set of successor services for w_i if and only if $\forall w_j \in S, \exists F_f \in w_i : w_i.F_f \cap w_j.I \neq \emptyset$.

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 $w_i \in \mathcal{W}$ invokes a service $w_j \in \mathcal{W}$, 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 $\{w_1, \cdots, w_l\} \subset \mathcal{W}$) then $\succ w_i = \{w_1, \cdots, w_l\}$. If the service w_i does not invoke any service from the set $\mathcal{W}$, 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 '$\oplus$' be a symbol that represents the service composition. We define composition of services as follows:

Definition II.4 (Service composition)**.** Given two Web services $w_i, w_j \in \mathcal{W} : w_j \in (\succ w_i)$, composition of w_i and w_j (represented as $w_i \oplus w_j$) yields a composite Web service $w_k \in \mathcal{W}$ such that
$\exists m \in w_i.I \Big((w_i.Rl(m) = n) \;\wedge\; (n \in w_j.I)\Big) \rightarrow \Big((m \in w_k.I) \;\wedge\; ((w_k.Rl(m) \subseteq w_j.Rl(n))\Big)$

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 $w_i \in \mathcal{W}$ 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 $w_i \in \mathcal{W}$ 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 ($\succ_C$) 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: $\langle w_j, m\rangle \in \big(\succ_C \langle w_i, I_p\rangle\big)$) if and only if $w_j \in \big(\succ w_i\big)$ and $m \in w_i.Rl(I_p)$.

Let $\langle w_i, m_p\rangle$ and $\langle w_j, m_q\rangle$ be two service-message tuples such that their composition ($\langle w_i, m_p\rangle \oplus \langle w_i, m_q\rangle$) 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 $\langle w_i, m_p\rangle \oplus \langle w_k, m_r\rangle$. 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 ($\langle w_i, m_p\rangle \oplus \epsilon = \langle w_i, m_p\rangle$). 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 ($\succ_R$) representing that $Domain(\succ_R) = Domain(\succ)$ and $Range(\succ_R) \subseteq Range(\succ_C)$. 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 $\langle w_x, I_r\rangle \in \big(\succ_C \langle w_i, I_p\rangle \oplus \langle w_j, I_q\rangle \oplus \cdots \oplus \langle w_n, I_s\rangle\big)$).
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 $\langle w_x, I_r\rangle \notin \{\langle w_i, I_p\rangle, \langle w_j, I_q\rangle, \cdots, \langle w_n, I_s\rangle\}$).

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

$$\succ_R \langle w_i, -\rangle \equiv \succ_R \{\langle w_i, I_1\rangle, \langle w_i, I_2\rangle, \cdots, \langle w_i, I_p\rangle\} \tag{1}$$

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

$$\succ_R \langle -, I_p\rangle \equiv \succ_R \{\langle w_i, I_p\rangle, \langle w_j, I_p\rangle, \cdots, \langle w_l, I_p\rangle\} \tag{2}$$

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

Let '$\circledast$' be a symbol to represent recursive composition. To define recursive composition, we use restrictive successor operator ($\succ_R$) and composition operator ($\oplus$) 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 $I_p \in w_i$ is defined as follows:

$$\circledast\langle w_i, I_p\rangle \triangleq \begin{cases} \langle w_i, I_p\rangle & ; \text{if } \succ_R \langle w_i, I_p\rangle = \emptyset \\ \circledast\big\{\langle w_i, I_p\rangle \oplus (\succ_R \langle w_i, I_p\rangle)\big\} & ; \text{otherwise} \end{cases} \tag{3}$$

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 ($E : V \rightarrow 2^V$) such that following condition holds
$E(v_i) = U,$ where $v_i \in V$ and $U \subseteq V$ iff $\forall v_j \in U : v_j \in \succ_R (v_i)$

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, $\mathcal{T}_{I_p}$ 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 $T_j \subset T_i$) 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 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 $I_p \in w_i.I$. 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: ⟨definition⟩, ⟨types⟩, ⟨message⟩, ⟨operation⟩, ⟨portType⟩, ⟨binding⟩, ⟨port⟩, and ⟨service⟩. Listing 1 depicts an abstract structural view of a WSDL document. The ⟨portType⟩ 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 ⟨operation⟩ element of the WSDL document. Except the ⟨operation⟩ 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

```
<portType name...>*
    <operation name...>
        <input message.../>
        <output message.../>
    </operation>
</portType>
```

Listing 2. Intermediate Representation

```
<portType name...>*
    <operation name...>
        <input message.../>
            <forward...>*
                <sequential.../>*
            </forward>
        <response message.../>
    </operation>
</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 ⟨operation⟩ element. In the case of a composite service, within the ⟨operation⟩ element, an ⟨ input⟩ element is preceded by a number of ⟨forward⟩ elements and each ⟨forward⟩ element consists ⟨sequential⟩ elements. A ⟨sequential⟩ element is a text element that consists a message that has to be forwarded to other services. All ⟨forward⟩ 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 $\mathcal{W} = TA,\ HB,\ FB,\ CB,\ EQ$, 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 $\mathcal{W}$ and $\langle Travel\_Booking \rangle$ 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 $\mathcal{W}$ triggered by the input message $Travel\_Booking$. The algorithm creates a root node (s_0) labeled with the input argument $Travel\_Booking$. Further, it searches existence of the input argument $Travel\_Booking$ in the input set of available services and $Travel\_Booking$ 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: $Flight\_Avail?$, $City\_Name$, and $Car\_Avail?$. 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 ($Flight\_Book$). 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].

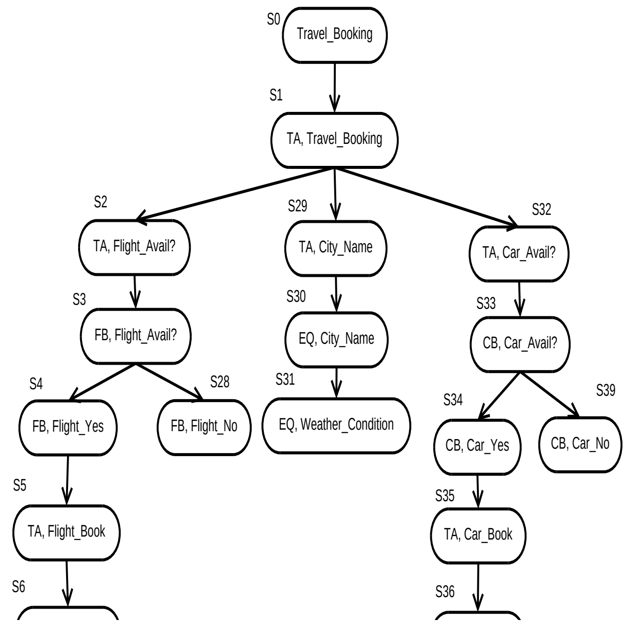

Fig. 1. A partial RCIG generated by the input $Travel\_Booking$

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 ϕ. 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 ϕ, $w_i.m_p$ refers a message m_p in w_i and m_q is a message name in general. The specification formula ϕ 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 $w_i.Ip = \top$, 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.

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:
$\phi ::= \top \mid \bot \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\phi \cup \phi) \mid A\phi \mid E\phi$

where p is any propositional atom from some set of atoms and each occurrence of ϕ to the right of ::= stands for any already constructed formula. $\top$ and $\bot$ are well formed formulas "the tautology" and "the falsum" respectively. $\neg$, $\wedge$, $\vee$, 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. A stands 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 $\vee$ and $\wedge$, 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) $T \nvDash \bot$ ($\bot$ is always false).
3) $T \models p$ iff $\exists s_i \in n(T) : s_i \models p$
4) $T \models \neg\phi$ iff $T \nvDash \phi$
5) $T \models \phi_1 \wedge \phi_2$ iff $T \models \phi_1$ and $T \models \phi_2$
6) $T \models \phi_1 \vee \phi_2$ iff $T \models \phi_1$ or $T \models \phi_2$
7) $T \models \phi_1 \rightarrow \phi_2$ iff $s_i, s_j \in n(T) : (s_i \models \phi_1 \;\wedge\; s_j \models \phi_2) \wedge (i < j)$
8) $T \models \phi_1 \cup \phi_2$ iff ϕ_1 is a negative literal of the form $\neg p$ and $\big((s_i, s_j \in n(T) : s_i \models p) \wedge (s_j \models \phi_2)\big) \implies i > j$
9) $T \models A\phi_1$ iff $T_i \models \phi_1$ for all $i \geq 1$
10) $T \models E\phi_1$ 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 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.

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, ϕ) written in RCSL. $\mathcal{W}$ is an online Web service repository available on a specific url address and ϕ is provided by a verifier. Once ϕ is available, Algorithm 1 calls Algorithm 2 by passing ϕ as an argument. Algorithm 2 parses ϕ, and correspondingly it generates an abstract syntax tree (written as P_ϕ) 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 INTERACTIONVERIFICATION($\mathcal{W}, \phi$)

Input: $\mathcal{W}$ (a set of Web services), ϕ (a specification formula)
Output: $\mathcal{W} \vDash \phi$ or $\mathcal{W} \nvDash \phi$

$P_\phi \leftarrow$ REQSPECPARSING(ϕ) ▷ calling Algorithm 2
$FLAG \leftarrow TRUE$
Integer i, j, p, t
$w_j.I$: set of all input messages in w_j
A_ϕ : set of atoms in ϕ
for all $\alpha_i \in A_\phi$ **do**
  for all $w_j \in \mathcal{W}$ **do**
    if $\alpha_i \in w_j.I$ **then**
      $I_p \leftarrow \alpha_i$
      $M \leftarrow$RCIGFORMATION ($\mathcal{W}, I_p$) ▷ M is a model formed by RCIGFORMATION algorithm
      for all trace $T_t \in M$ **do**
        $FLAG \leftarrow$ INTERPRETATION(P_ϕ, T_t) ▷ calling Algorithm 3
      end for
    end if
  end for
end for
if $FLAG = TRUE$ **then**
  $\mathcal{W} \vDash \phi$ ▷ available services satisfy the specification formula
else
  $\mathcal{W} \nvDash \phi$ ▷ available services do not satisfy the specification formula
end if

Further, Algorithm 1 collects all the atoms from ϕ in the set A_ϕ and observes whether an atom (say, α_i) belongs to an input set of a service from the set $\mathcal{W}$. If α_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 extract the traces T_t ($t \in \mathbb{N}$) from the model M one by one and calls Algorithm 3 for further processing by passing the arguments P_ϕ and T_t. Then, Algorithm 3 interprets P_ϕ on the provided trace T_t and results as TRUE or FALSE, based on its computation. In case, if the result is TRUE, trace T_t is a witness example, otherwise trace T_t is a counter example. Algorithm 3 decomposes the AST P_ϕ in subtrees recursively and divide also the trace T_t recursively corresponding to subtrees until unit-level-subtrees (smallest non-trivial subtrees) are achieved. Now, the function $PInterpretation(subtree, trace)$ 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 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

int i = 0
String Id_i ← NULL
String Token ← NULL
String nextToken ← NULL
String prevToken ← NULL
for all Token ∈ φ do
    TokenSet ← Token
end for
Token ← TokenSet(i)
while |TokenSet| ≠ 1 AND Token ≠ Id do
    if Token =‘¬’ then
        if nextToken =‘(’ then
            Parenthesis(nextToken)
        else
            Id_i ← nextToken
            Replace nextToken with Id_i in φ
        end if
        Replace ‘¬Id_i’ with ‘Id_{i+1}’ in φ
        i ← i + 1
    else if Token =‘(’ then
        while nextToken =‘(’ do
            Token ← nextToken
        end while
        Parenthesis(Token)
    else if Token =‘∪’ then
        Func(Token)
    else if Token =‘∨’ then
        Func(Token)
    else if Token =‘∧’ then
        Func(Token)
    else if Token =‘→’ then
        Func(Token)
    else if Token =‘p’ then    ▷ p is a proposition
        Id_i ← ‘p’
        Replace ‘p’ with ‘Id_i’ in φ
        i ← i + 1
    else if Token = Id then
        Skip    ▷ move to next token
    end if
end while
function Parenthesis(Value)
    String Token ← Value
    repeat
        φ_Temp ← Token
        Token ← nextToken
    until Token ≠‘)’
    φ_Temp ← Token
    Replace φ_Temp with Id_i in φ
    i ← i + 1
end function
function Func(Token)
    Id_i ← prevToken Token nextToken
    Replace prevToken Token nextToken with ‘Id_i’ in φ
    i ← i + 1
end function
AST(Id_{i−1})    ▷ print the abstract syntax tree for φ
```

pretation model M and ϕ_1 (see Equation 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.

$$\phi_1 = A((Flight\_Yes \wedge Flight\_Book) \rightarrow (Flight\_Booked \vee (\neg Hotel\_Booked \cup (Flight\_Booked)))) \quad (4)$$

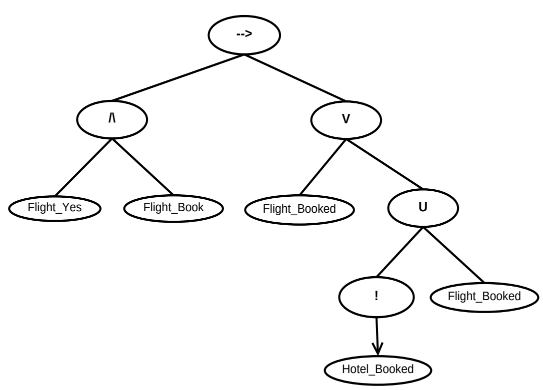

Fig. 2. AST for the specification formula ϕ_1 (Eqn. 4)

Now, a verifier has to verify the model M against the formula ϕ_1. According to the verification technique, traces in model M are considered one by one for verification. Let us consider that a trace T_t (shown in Fig. 3) from the model M ($T_t = s_0, \cdots, s_{21}$) has to be verified against ϕ_1. The AST (P_{ϕ_1}) for ϕ_1 is given in Fig. 2.

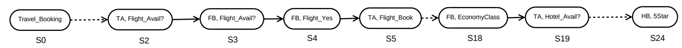

Fig. 3. A trace from the RCIG in Fig. 1

According to Algorithm 1, P_{ϕ_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_{ϕ_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 ϕ is completely decomposed in its constituent unit-level-subtrees, decomposition process stops.

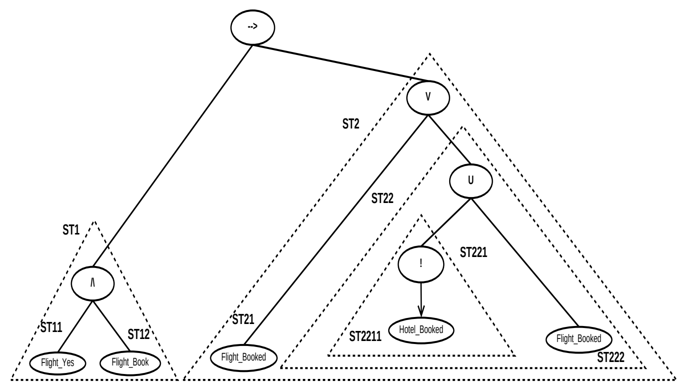

Fig. 4. Subtree decomposition of the AST given in Fig. 2

Now, $ST1 \rightarrow ST2$ will be interpreted on the trace T_t. The subtree $ST1$ represents a subformula $Flight\_Yes \wedge Flight\_Book$ that is satisfied in the subtrace $s_0, \cdots, s_5$. Then, we divide the trace T_t into subtraces: $s_0, \cdots, s_6$ and $s_7, \cdots, s_{24}$. After this, the subtree $ST2 ::= ST21 \vee ST22$ $(Flight\_Booked \vee (\neg Hotel\_Booked \cup Flight\_Booked))$ gets interpreted over subtrace $s_7, \cdots, s_{24}$. Since the subtrace satisfies $ST21$, $ST2 ::= ST21 \vee ST22$ becomes satisfied. Consequently, the requirement specification $\phi_1 = (Flight\_Yes \wedge Flight\_Book) \rightarrow (Flight\_Booked \vee (\neg Hotel\_Booked \cup Flight\_Booked))$ is satisfied in the trace T_t. In the similar way, we check satisfiability of ϕ_1 over every trace in the model M and find satisfied. Hence, $M \models \phi_1$

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:

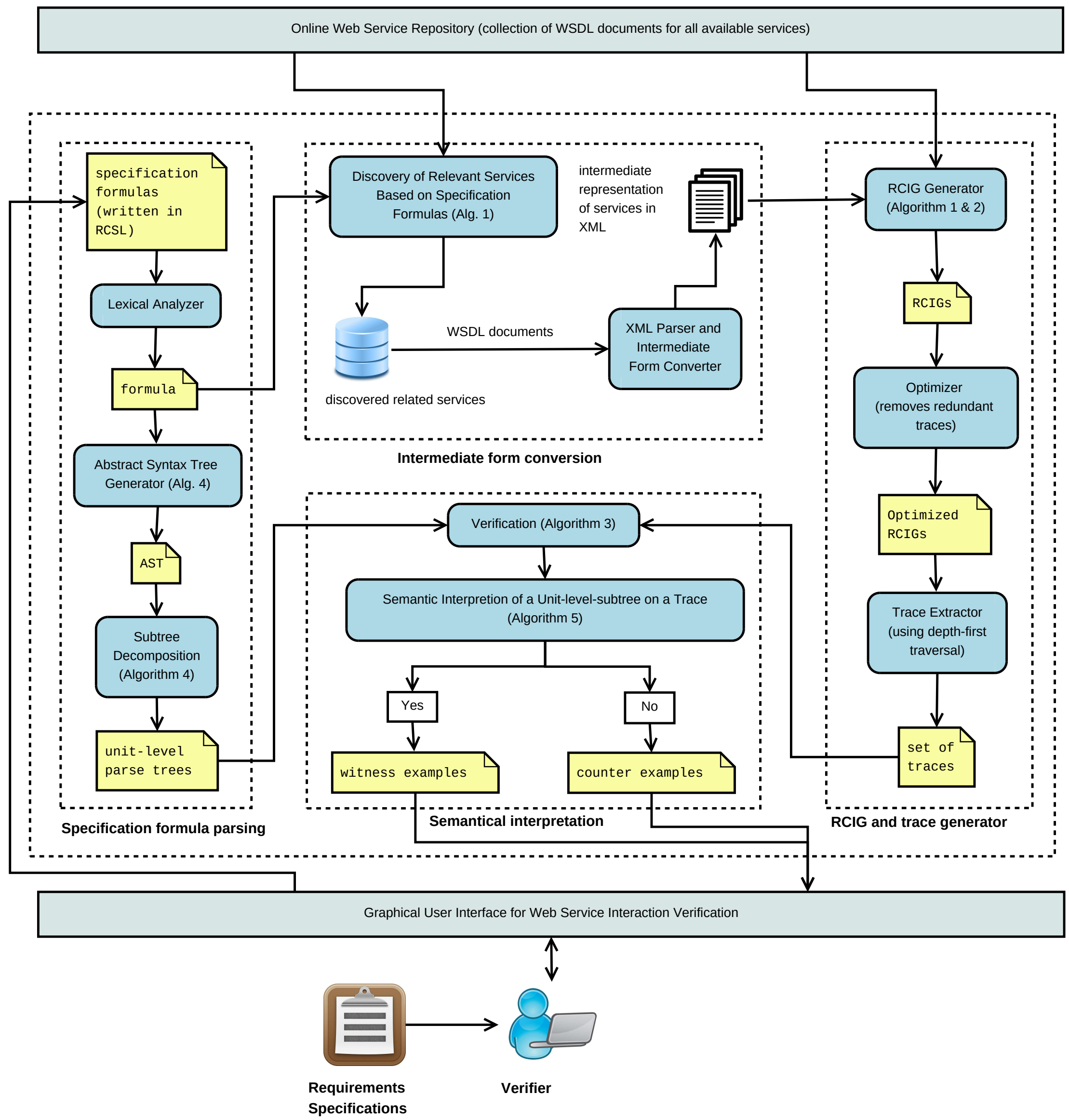

Fig. 5. High-level architecture of implementation

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 INTERPRETATION(P_ϕ, T)

```
Input: P_φ (parse tree) and T (trace)
Output: TRUE or FALSE

Root ← Root(P_φ)
LST ← LeftSubTree(P_φ)
RST ← RightSubTree(P_φ)
if Root ∉ {¬, ∧, ∨, →, ∪} then
    if PINTERPRETATION(Root, T) = TRUE then
        Return TRUE
    else
        Return FALSE
    end if
else if Root = '¬' then
    if PINTERPRETATION(LST, T) = TRUE then
        Return FALSE
    else
        Return TRUE
    end if
else if Root = '∧' then
    if INTERPRETATION (LST, T) = TRUE AND INTERPRETATION (RST, T) = TRUE then
        Return TRUE
    else
        Return FALSE
    end if
else if Root = '∨' then
    if INTERPRETATION (LST, T) = TRUE OR INTERPRETATION (RST, T) = TRUE then
        Return TRUE
    else
        Return FALSE
    end if
else if Root = '→' then
    FLAG ← FALSE
    Temp ← ∅
    T_Temp ← ∅
    for all node n ∈ T do
        Temp · n    ▷ Concatenating n to the existing sequence of nodes in Temp
        if INTERPRETATION (LST, n) = TRUE then
            FLAG ← TRUE
            BREAK
        end if
    end for
    T_Temp ← {T − Temp}
    if INTERPRETATION (RST, T_Temp) = TRUE then
        Return TRUE
    else if FLAG = FALSE AND INTERPRETATION (RST, T_Temp) = FALSE then
        Return TRUE
    else
        Return FALSE
    end if
else if Root = '∪' then
    FLAG ← FALSE
    Temp ← ∅
    for all node n ∈ T do
        Temp · n
        if INTERPRETATION (LST′, Temp) = FALSE then
            FLAG ← FALSE
            BREAK
        end if
    end for
    if FLAG ≠ FALSE then
        Return FALSE
    else if INTERPRETATION (RST, Temp) = TRUE then
        Return FALSE
    else
        Return TRUE
    end if
end if
function PINTERPRETATION(p,T)
    FLAG ← FALSE
    for all node n ∈ T do
        if p ∈ L(n) then    ▷ L(n) means label of node n
            FLAG ← TRUE
            BREAK
        end if
    end for
    if FLAG = TRUE then
        Return TRUE
    else
        Return FALSE
    end if
end function
```

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.

B. 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 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.

1) Experimental Setup

Let us consider a set of three services ($\mathcal{W} := w_f, w_s, w_r$), where w_f and w_s are composite services and w_r is a basic service. All three services accept only one input message: $Hotel\_Avail$. The composite services (w_f and 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 $\langle forward \rangle$, $\langle sequential \rangle$, and $\langle response \rangle$ 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_s can invoke only w_r. 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, and w_r. All observations are taken by providing the input $\langle w_f, Hotel\_Avail \rangle$ 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.

2) Experimental Evaluation

There are three types of elements in the candidate services: ⟨forward⟩, ⟨sequential⟩, and ⟨response⟩. Observations are taken for the total number of nodes by increasing an element type. In order to support symmetrical growth, we assume that a ⟨forward⟩ element can be increased only in w_f, a ⟨sequential⟩ element can be increased only in w_s, and a ⟨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 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: Res, Seq, and Fwd. These keywords stand for $\langle response \rangle$, $\langle sequential \rangle$, and $\langle forward \rangle$ elements

Messages	# Nodes	Messages	# Nodes
1Res 1Seq 1Fwd	5	1Res 4Seq 4Fwd	50
2Res 1Seq 1Fwd	6	2Res 4Seq 4Fwd	242
3Res 1Seq 1Fwd	7	3Res 4Seq 4Fwd	802
4Res 1Seq 1Fwd	8	4Res 4Seq 4Fwd	2042
5Res 1Seq 1Fwd	9	5Res 4Seq 4Fwd	4370
6Res 1Seq 1Fwd	10	6Res 4Seq 4Fwd	8290
1Res 2Seq 2Fwd	14	1Res 5Seq 5Fwd	77
2Res 2Seq 2Fwd	26	2Res 5Seq 5Fwd	622
3Res 2Seq 2Fwd	42	3Res 5Seq 5Fwd	3027
4Res 2Seq 2Fwd	62	4Res 5Seq 5Fwd	10231
5Res 2Seq 2Fwd	86	5Res 5Seq 5Fwd	24524
6Res 2Seq 2Fwd	114	6Res 5Seq 5Fwd	62202
1Res 3Seq 3Fwd	29	1Res 6Seq 6Fwd	110
2Res 3Seq 3Fwd	86	2Res 6Seq 6Fwd	1514
3Res 3Seq 3Fwd	197	3Res 6Seq 6Fwd	10922
4Res 3Seq 3Fwd	380	4Res 6Seq 6Fwd	49142
5Res 3Seq 3Fwd	653	5Res 6Seq 6Fwd	164054
6Res 3Seq 3Fwd	1034	6Res 6Seq 6Fwd	447890

TABLE I
EFFECT OF INCREASING RESPONSE MESSAGES ON THE ORDER OF THE RCIG

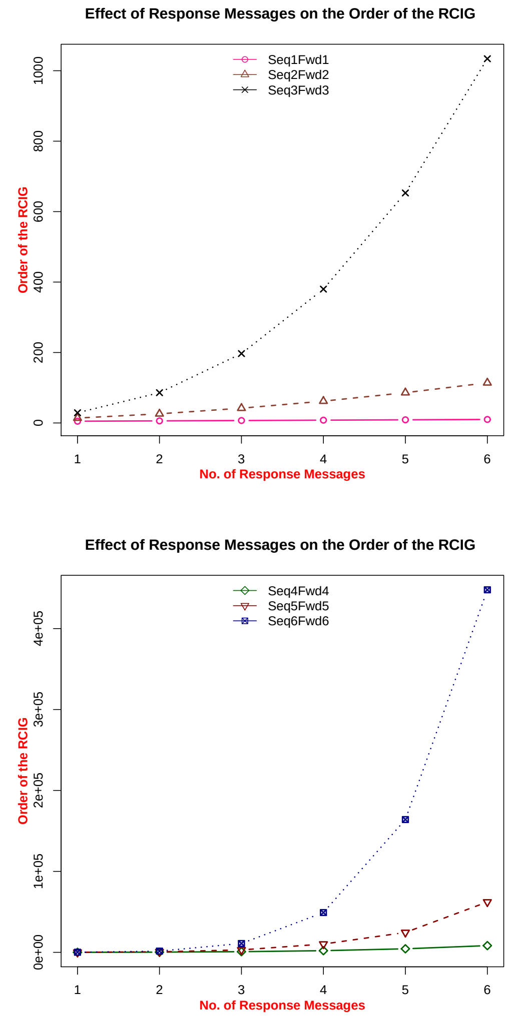

Fig. 6. Effect of Increasing Response Messages on the Order of the RCIG

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 ($y = 3x + 2$). 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: 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.

In the 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

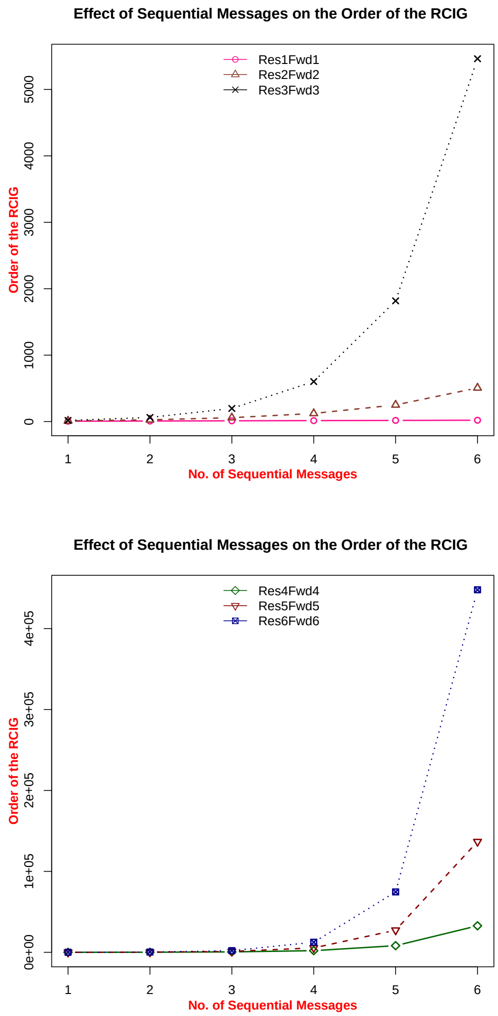

Fig. 7. Effect of Increasing Sequential Messages on the Order of the RCIG

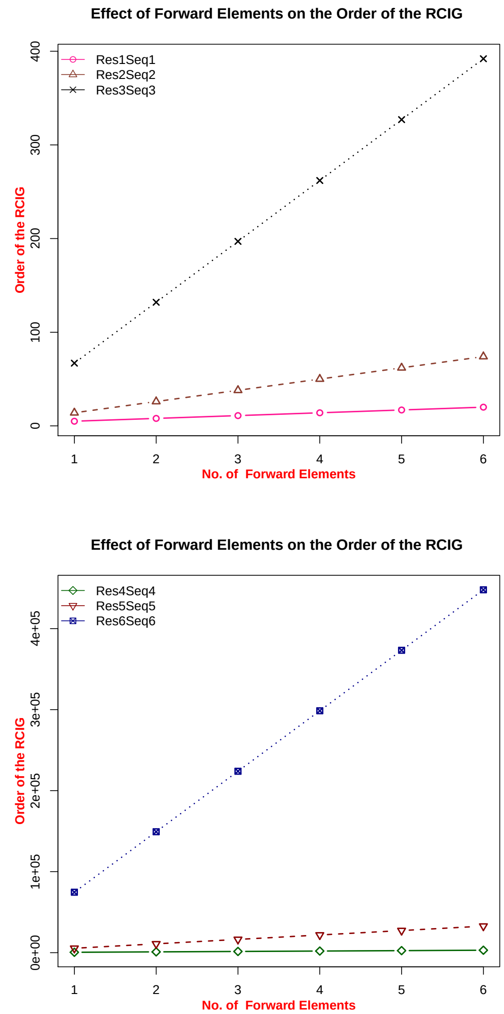

Fig. 8. Effect of Increasing Forward Elements on the Order of the RCIG

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 '$\langle A, Hotel\_Book\rangle \oplus \langle B, Hotel\_Book\rangle$'. Further, B in '$\langle A, Hotel\_Book\rangle \oplus \langle B, Hotel\_Book\rangle$' invokes C for hotel booking that can be written as '$\langle A, Hotel\_Book\rangle \oplus \langle B, Hotel\_Book\rangle \oplus \langle C, Hotel\_Book\rangle$'. Now, if C invokes A for hotel booking, it lead into an indirect deadlock written as '$\langle A, Hotel\_Book\rangle \oplus \langle B, Hotel\_Book\rangle \oplus \langle C, Hotel\_Book\rangle, \rangle \oplus \langle A, Hotel\_Book\rangle$'. 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

[1] G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju, *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[2] Q. Z. Sheng, X. Qiao, A. V. Vasilakos, C. Szabo, S. Bourne, and X. Xu, “Web services composition: A decade’s overview,” *Inform. Sci.*, vol. 280, pp. 218–238, 2014.

[3] C. Granell, M. Gould, R. Grønmo, and D. Skogan, “Improving reuse of web service compositions,” in *E-Commerce and Web Technologies*. Springer, 2005, pp. 358–367.

[4] V. Jaiswal, A. Sharma, and A. Verma, “Recomp: Qos-aware recursive service composition at minimum cost,” in *Proc. of the 12th IFIP/IEEE Int. Symp. on Integrated Network Management*. IEEE, 2011, pp. 225–232.

[5] C.-A. Sun, X. Zhang, Y. Shang, and M. Aiello, “Integrating transactions into bpel service compositions: An aspect-based approach,” *ACM Trans. on the Web*, vol. 9, no. 2, p. 9, 2015.

[6] G. Zou, Y. Gan, Y. Chen, and B. Zhang, “Dynamic composition of web services using efficient planners in large-scale service repository,” *Knowledge-Based Systems*, vol. 62, pp. 98–112, 2014.

[7] Q. Hu, Y. Du, and S. Yu, “Service net algebra based on logic petri nets,” *Information Sci.*, vol. 268, pp. 271–289, 2014.

[8] F. Belli, A. T. Endo, M. Linschulte, and A. Simao, “A holistic approach to model-based testing of web service compositions,” *Software: Practice and Experience*, vol. 44, no. 2, pp. 201–234, 2014.

[9] M. Chen, T. H. Tan, J. Sun, Y. Liu, J. Pang, and X. Li, “Verification of functional and non-functional requirements of web service composition,” in *Proc. of the Int. Conf. on Formal Engineering Methods*. Springer, 2013, pp. 313–328.

[10] W. El Kholy, J. Bentahar, M. El Menshawy, H. Qu, and R. Dssouli, “Modeling and verifying choreographed multi-agent-based web service compositions regulated by commitment protocols,” *Expert Systems with Applications*, vol. 41, no. 16, pp. 7478–7494, 2014.

[11] G. N. Rai and G. R. Gangadharan, “Set partition and trace based verification of web service composition,” in *Proc. of the 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT)*, 2015, pp. 278–285.

[12] X. Fu, T. Bultan, and J. Su, “Analysis of interacting BPEL web services,” in *Proc. of the 13th Int. Conf. on World Wide Web*. ACM, 2004, pp. 621–630.

[13] R. Kazhamiakin, M. Pistore, and M. Roveri, “Formal verification of requirements using SPIN: A case study on web services,” in *Proc. of the 2nd Int. Conf. on Software Eng. and Formal Methods*. IEEE, 2004, pp. 406–415.

[14] G. N. Rai, G. R. Gangadharan, and V. Padmanabhan, “Algebraic modeling and verification of web service composition,” in *Proc. of the 6th Int. Conf. on Ambient Systems, Networks and Technologies (ANT)*, 2015, pp. 675–679.

[15] G. M. Kapitsaki, D. A. Kateros, C. A. Pappas, N. D. Tselikas, and I. S. Venieris, “Model-driven development of composite web applications,” in *Proc. of the 10th Int. Conf. on Inform. Integration and Web-based Applications & Services*. ACM, 2008, pp. 399–402.

[16] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Compatibility verification for web service choreography,” in *Proc. of the Int. Conf. on Web Services (ICWS)*. IEEE, 2004, pp. 738–741.

[17] R. Hamadi and B. Benatallah, “A petri net-based model for web service composition,” in *Proc. of the 14th Australian Database Conf.* Australian Computer Society, Inc., 2003, pp. 191–200.

[18] A. Ferrara, “Web services: a process algebra approach,” in *Proc. of the 2nd Int. Conf. on Service Oriented Computing*. ACM, 2004, pp. 242–251.

[19] E. Kaldeli, A. Lazovik, and M. Aiello, “Domain-independent planning for services in uncertain and dynamic environments,” *Artificial Intelligence*, vol. 236, pp. 30–64, 2016.

[20] P. Papapanagiotou and J. D. Fleuriot, “A theorem proving framework for the formal verification of web services composition,” in *Proc. of the 7th Int. Workshop on Automated Specification and Verification of Web Systems, WWV*, 2011, pp. 1–16.

[21] D. Skogan, R. Grønmo, and I. Solheim, “Web service composition in uml,” in *Proc. of the* 8^{th} *IEEE Int. Conf. on Enterprise Distributed Object Computing*. IEEE, 2004, pp. 47–57.

[22] B. Benatallah, F. Casati, and F. Toumani, “Web service conversation modeling: A cornerstone for e-business automation,” *Internet Computing, IEEE*, vol. 8, no. 1, pp. 46–54, 2004.

[23] B. Finkbeiner and I. Krüger, “Using message sequence charts for component-based formal verification,” in *In: Proc. of the Int. Conf. on Specification and Verification of Component-Based Systems*, 2001, pp. 32–45.

[24] A. Letichevsky, J. Kapitonova, V. Volkov, S. Baranov, and T. Weigert, “Basic protocols, message sequence charts, and the verification of requirements specifications,” *Computer Networks*, vol. 49, no. 5, pp. 661–675, 2005.

[25] D. Jordan, J. Evdemon, A. Alves, A. Arkin, S. Askary, C. Barreto, B. Bloch, F. Curbera, M. Ford, Y. Goland *et al.*, “Web services business process execution language version 2.0,” *OASIS standard*, vol. 11, p. 11, 2007.

[26] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, World Wide Web Consortium Std., 2007.

[27] N. Mehandjiev, F. Lécué, M. Carpenter, and F. A. Rabhi, “Cooperative service composition,” in *Proc. of the Int. Conf. on Advanced Inform. Systems Engineering*. Springer, 2012, pp. 111–126.

[28] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press Cambridge, 2008.

[29] H. Giese and B. Becker, *Modeling and verifying dynamic evolving service-oriented architectures*. Universitätsverlag Potsdam, 2013.

[30] S. Hashemian and F. Mavaddat, “Composition algebra,” in *Proc. of the Formal Aspects of Component Software (FACS06)*, 2006, pp. 247–264.

[31] P. Höfner and F. Lautenbacher, “Algebraic structure of web services,” *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 3, pp. 171–187, 2008.

[32] Q. Yu and A. Bouguettaya, “Framework for web service query algebra and optimization,” *ACM Trans. on the Web*, vol. 2, no. 1, p. 6, 2008.

[33] H. Foster, S. Uchitel, J. Magee, and J. Kramer, “Model-based verification of web service compositions,” in *Proc. of the 18th IEEE Int. Conf. on Automated Software Engineering*. IEEE, 2003, pp. 152–161.

[34] L. E. Moser and P. M. Melliar-Smith, “Formal verification of safety-critical systems,” *Software: Practice and Experience*, vol. 20, no. 8, pp. 799–821, 1990.

[35] A. P. Sistla, “Safety, liveness and fairness in temporal logic,” *Formal Aspects of Computing*, vol. 6, no. 5, pp. 495–511, 1994.

[36] S. Kona, A. Bansal, G. Gupta, and D. Hite, “Automatic composition of semantic web services.” in *Proc. of the Int. Conf. on Web Services*, vol. 7, no. 1, 2007, pp. 150–158.

[37] J. Peer, “Description and automated processing of web services,” Ph.D. dissertation, University of St. Gallen, 2006.

[38] P. Bertoli, M. Pistore, and P. Traverso, “Automated composition of web services via planning in asynchronous domains,” *Artificial Intelligence*, vol. 174, no. 3, pp. 316–361, 2010.

[39] S. Lu, A. Bernstein, and P. Lewis, “Automatic workflow verification and generation,” *Theoretical Computer Science*, vol. 353, no. 1, pp. 71–92, 2006.

[40] O. Hatzi, D. Vrakas, M. Nikolaidou, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, “An integrated approach to automated semantic web service composition through planning,” *IEEE Trans. on Services Computing*, vol. 5, no. 3, pp. 319–332, 2012.

[41] I. Georgievski and M. Aiello, “HTN planning: Overview, comparison, and beyond,” *Artificial Intelligence*, vol. 222, pp. 124–156, 2015.

[42] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau, “Automating daml-s web services composition using shop2,” in *Proc. of the Int. Semantic Web Conf.* Springer, 2003, pp. 195–210.

[43] A. Abrougui, A. Mercier, M. Occello, and M.-P. Huget, “Recursive multi-agent system for dynamic and adaptative web services composition,” in *Proc. of the Int. Conf. on Management of Emergent Digital EcoSystems*. ACM, 2009, pp. 44:295–44:299.

[44] C. Walton, “Model checking multi-agent web services,” in *Proc. of the AAAI Symp. on Semantic Web Services*, 2004.

[45] H. Schlingloff, A. Martens, and K. Schmidt, “Modeling and model checking web services,” *Electronic Notes in Theoretical Computer Sci.*, vol. 126, pp. 3–26, 2005.

[46] Y. Zheng, J. Zhou, and P. Krause, “A model checking based test case generation framework for web services,” in *Proc. of the 4th Int. Conf. on Inform. Technology (ITNG ’07)*. IEEE, 2007, pp. 715–722.

[47] S. Rossi, “Model checking adaptive multilevel service compositions,” in *Proc. of the 7th Int. Workshop on Formal Aspects of Component Software, FACS*, 2010, pp. 106–124.

[48] Q. Z. Sheng, Z. Maamar, L. Yao, C. Szabo, and S. Bourne, “Behavior modeling and automated verification of web services,” *Inform. Sci.*, vol. 258, pp. 416–433, 2014.

[49] J. Bentahar, H. Yahyaoui, M. Kova, and Z. Maamar, “Symbolic model checking composite web services using operational and control behaviors,” *Expert Systems with Applications*, vol. 40, no. 2, pp. 508–522, 2013.

[50] H. Groefsema, N. van Beest, and M. Aiello, “A formal model for compliance verification of service compositions,” *IEEE Trans. on Services Computing*, vol. 10.1109/TSC.2016.2579621, 2016.

Gopal N. Rai received his B.Sc. and M.C.A. degree from University of Allahabad, Allahabad, India, in 2007 and 2011, respectively. He is currently working towards the Ph.D. degree at the University of Hyderabad and the Institute for Development and Research in Banking Technology (IDRBT), Hyderabad. His research interests include Software Verification, Web services, and formal methods.

G R Gangadharan is an associate professor at the Institute for Development and Research in Banking Technology, Hyderabad, India. His research interests focus on the interface between technological and business perspectives. Gangadharan received his PhD in information and communication technology from the University of Trento, Italy, and the European University Association. He is a senior member of IEEE and ACM.

Vineet Padmanabhan is a professor in the School of Computer and Information Sciences in the University of Hyderabad, India. His areas of interest include Knowledge Representation and Reasoning, Multi-Agent Systems, and Logics in Artificial Intelligence. Vineet received his Ph.D. from Griffith University, Australia.

Rajkumar Buyya 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 at the University of Melbourne, Australia. His areas of interest include grid computing, cloud computing, and high-performance computing. Buyya received his Ph.D. from University of Melbourne, Australia. He is a Fellow of IEEE and Life Member of ACM.