1-2013

Verification of functional and non-functional requirements of web service composition

Manman CHEN

Tian Huat TAN

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Yang LIU

Jun PANG

See next page for additional authors

Citation

CHEN, Manman; TAN, Tian Huat; SUN, Jun; LIU, Yang; PANG, Jun; and LI, Xiaohong. Verification of functional and non-functional requirements of web service composition. (2013). *Proceedings of the 15th International Conference on Formal Engineering Methods, ICFEM 2013, Queenstown, New Zealand, October 29 - November 1*. 313-328.
Available at: https://ink.library.smu.edu.sg/sis_research/5000

Verification of Functional and Non-functional Requirements of Web Service Composition*

Manman Chen[1], Tian Huat Tan[1], Jun Sun[2], Yang Liu[3], Jun Pang[4], and Xiaohong Li[5]

[1] School of Computing, National University of Singapore
{chenman,tianhuat}@comp.nus.edu.sg
[2] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[3] Nanyang Technological University
yangliu@ntu.edu.sg
[4] Université du Luxembourg
jun.pang@uni.lu
[5] Tianjin University
xiaohongli@tju.edu.cn

Abstract. Web services have emerged as an important technology nowadays. There are two kinds of requirements that are crucial to web service composition, which are functional and non-functional requirements. Functional requirements focus on functionality of the composed service, e.g., given a booking service, an example of functional requirements is that a flight ticket with price higher than $2000 will never be purchased. Non-functional requirements are concerned with the quality of service (QoS), e.g., an example of the booking service's non-functional requirements is that the service will respond to the user within 5 seconds. Non-functional requirements are important to web service composition, and are often an important clause in service-level agreements (SLAs). Even though the functional requirements are satisfied, a slow or unreliable service may still not be adopted. In our paper, we propose an automated approach to verify combined functional and non-functional requirements directly based on the semantics of web service composition. Our approach has been implemented and evaluated on the real-world case studies, which demonstrate the effectiveness of our method.

1 Introduction

Based on Service Oriented Architecture (SOA), Web services make use of open standards, such as WSDL [1] and SOAP [2], that enable the interaction among heterogeneous applications. A real-world business process may contain a set of services. A web service is a single autonomous software system with its own thread of control. A fundamental goal of web services is to have a collection of network-resident software services, so that it can be accessed by standardized protocols and integrated into applications or composed to form complex services which are called *composite services*. A

* This research is supported in part by Research Grant IDD11100102 of Singapore University of Technology and Design, IDC and MOE2009-T2-1-072 (Advanced Model Checking Systems).

L. Groves and J. Sun (Eds.): ICFEM 2013, LNCS 8144, pp. 313–328, 2013.

composite service is constructed from a set of *component services*. Component services have their interfaces and functionalities defined based on their internal structures. While the technology for creating services and interconnecting them with a point-to-point basis has achieved a certain degree of maturity, there is a challenge to integrate multiple services for complex interactions. Web service composition standards have been proposed in order to address this challenge. The $de\ facto$ standard for Web service composition is Web Services Business Process Execution Language (WS-BPEL) [3]. WS-BPEL is an XML-based orchestration business process language. It provides basic activities such as service invocation, and compositional activities such as sequential and parallel composition to describe composition of web services. BPEL is inevitably rich in concurrency and it is not a simple task for programmers to utilize concurrency as they have to deal with multi-threads and critical regions. It is reported that among the common bug types concurrency bugs are the most difficult to fix correctly, the statistic shows that 39% of concurrency bugs are fixed incorrectly [4]. Therefore, it is desirable to verify web services with automated verification techniques, such as model checking [5].

There are two kinds of requirements of web service composition, i.e., functional and non-functional requirements. Functional requirements focus on the functionalities of the web service composition. Given a booking service, an example of functional requirement is that a flight ticket with price higher than $2000 will never be purchased. The non-functional requirements are concerned with the Quality of Service (QoS). These requirements are often recorded in service-level agreements (SLAs), which is a contract specified between service providers and customers. Given a booking service, an example of non-functional requirements is that the service will respond to the user within 5 ms. Typical non-functional requirements include response time, availability, cost and so on. However, it is difficult for service designers to take the full consideration of both functional and non-functional requirements when writing BPEL programs.

Model checking is an automatic technique for verifying software systems [5], which helps find counterexamples based on the specification at the design time so that it could detect errors and increase the reliability of the system at the early stage. Currently, increasing number of complex service processes and concurrency are developed on web service composition. Hence, model checking is a promising approach to solve this problem. Given functional and non-functional requirements, existing works [6,7,8,9] only focus on verification of one aspect, and disregard the other, even though these two aspects are inseparable. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework.

In this work, we propose a method to verify BPEL programs against combined functional and non-functional requirements. A dedicated model checker is developed to support the verification. We make use of the labeled transition systems (LTSs) directly from the semantics of BPEL programs for functional verification. For non-functional properties, we propose different strategies to integrate different non-functional properties into the functional verification framework. We focus on three important non-functional properties in this work, i.e., availability, cost and response time. To verify availability and cost, we calculate them on-the-fly during the generation of LTS, and associate

calculated values to each state in the LTS. Verification of response time requires an additional preprocessing stage, before the generation of LTS. In the preprocessing stage, response time tag is assigned to each activity that is participated in the service composition. With such integration, we are able to support combined functional and non-functional requirements.

The contributions of our work are summarized as follows.

1. We support integrated verification of functional and non-functional properties for Web service composition. To the best of our knowledge, we are the first work on such integration.
2. We capture the semantics of web service composition using labeled transition systems (LTSs) and verify the web service composition directly without building intermediate or abstract models before applying verification approaches, which makes our approach more suitable for general web service composition verification.
3. Our approach has been implemented and evaluated on the real-world case studies, and this demonstrates the effectiveness of our method.

Paper Outline. The rest of paper is structured as follows. Section 2 describes the BPEL running example. Section 3 introduces QoS compositional model. Section 4 shows how to verify functional and non-functional propeties. Section 5 provides the evaluation of our work. Section 6 reviews the related work. Finally, Section 7 concludes the paper and outlines our future work.

2 Motivating Example

In our work, we assume that composite services are specified in the BPEL language. BPEL is the *de facto* standard for implementing composition of existing services by specifying an executable workflow using predefined activities. BPEL is an XML-based orchestration business process language for the specification of executable and abstract business processes. It supports control flow structures such as sequential and concurrency execution. In the following, we introduce the basic BPEL notations. <receive>, <invoke>, and <reply> are the basic communication activities which are defined to receive messages, execute component services and return messages respectively for communicating with component services. There are two kinds of <invoke> activities, i.e., synchronous and asynchronous invocation. Synchronous invocation activities are invoked and the process waits for the reply from the component service before moving on to the next activity. Asynchronous invocation activities are invoked and moving on to the next activity directly without waiting for the reply. The control flow of composite services is specified using the activities like <sequence>, <while>, <if> and <flow>. <sequence> is used to define the sequential ordering structure, <while> is used to define the loop structure, <if> is used to define the conditional choice structure, and <flow> is used to implement concurrency structure.

2.1 Computer Purchasing Services (CPS)

In this section, we introduce the computer purchasing service (CPS), which is designed to allow users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 1.

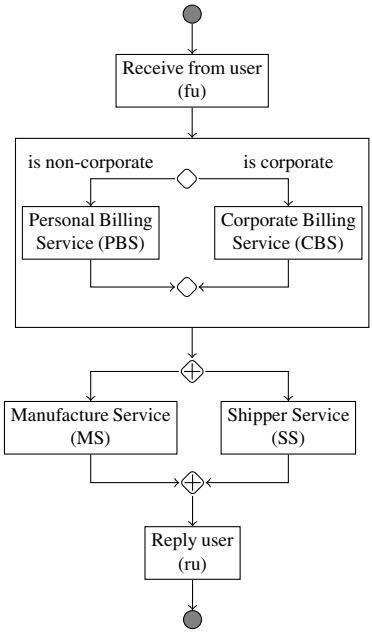

Fig. 1. Computer Purchasing Service

CPS has four component web services, namely Personal Billing Service (PBS), Corporate Billing Service (CBS), Manufacture Service (MS), Shipper Service (SS). CPS is initialized (donated by ●) upon receiving the request from the customer (fu) with the information of the customer and the computer that he wishes to purchase for. Subsequently, an <if> activity (donated by ◇) is used for checking whether the customer is a corporate customer or non-corporate customer. If it is a corporate customer, CBS is invoked synchronously to bill the corporate customer, otherwise, PBS is invoked synchronously to bill the non-corporate customer with credit card information. Upon receiving the reply, a <flow> activity (donated by ⊕) is triggered and MS and SS are invoked concurrently. MS is invoked synchronously to notify manufacture department for manufacturing the purchased computers. SS is invoked synchronously to schedule shipment for the purchased computers. Upon receiving the reply message from SS and MS, reply user (ru) is called to return the result of the computer purchasing to the customer. Then, the workflow of CPS has ended (donated by ●).

A property that CPS must fulfill is that it must invoke reply user (ru) within $5\ ms$. Notice that this property combines the functional (must invoke reply user (ru)) and non-functional (within $5\ ms$) requirements.

2.2 BPEL Notations

In order to present BPEL syntax compactly, we define a set of BPEL notations below:

– $rec(S)$ and $reply(S)$ are used to denote "receive from" and "reply to" a service S;

Table 1. QoS Attribute Values

QoS Attribute	PBS	CBS	MS	SS
Response Time(ms)	1	2	3	1
Availability(%)	90	80	80	80
Cost($)	3	2	2	2

- $sInv(S)$ (resp. $aInv(S)$) is used to denote synchronous (resp. asynchronous) invocation of a service S;
- $P_1||P_2$ is used to denote <flow> activity, i.e., the concurrent execution of BPEL activities P_1 and P_2;
- $P_1 \triangleleft b \triangleright P_2$ is used to denote <if> activity, where b is a guard condition. Activity P_1 is executed if b is evaluated true. Otherwise, activity P_2 will be executed;
- $P_1 \rightarrow P_2$ is used to denote <sequence> activity, where P_1 is executed followed by P_2.

We denote activities that contain other activities as *composite activities*, they are $P_1||P_2$, $P_1 \triangleleft b \triangleright P_2$ and $P_1 \rightarrow P_2$. For activities that do not contain any other activities, we denote them as *atomic activities*, they are $rec(S)$, $reply(S)$, $sInv(S)$ and $aInv(S)$.

3 QOS-Aware Compositional Model

In this section, we define the QoS compositional model used in this work and briefly introduce the semantics of BPEL, captured by labeled transition systems (LTSs). We introduce some definitions used in the semantic model in the following.

3.1 QoS Attributes

In this work, we deal with quantitative attributes that can be quantitatively measured using metrics. There are two classes of QoS Attributes, positive and negative attributes. Positive attributes (e.g., availability) have a good effect on the system, and therefore, they need to be maximized. Availability of the service is the probability of the service being available. Negative attributes (e.g., response time, cost) need to be minimized as they have the negative impact on the system. Response time of the service is defined as the delay between sending a request and receiving the response and cost of the service is defined as the money spent on the service. In this work, we assume the unit of response time, availability and cost to be millisecond (ms) , percentage (%) and dollar ($). Table 1 shows the information of response time, availability and cost of each component service for the CPS example as described in Section 2.1.

Given a component service s with n QoS attributes, we use a vector $Q_s = \langle q_1(s), \ldots, q_n(s)\rangle$ to represent QoS attributes of the service s , where $q_i(s)$ represents the value of ith attribute of the component service s. Similarly, $Q'_{cs} = \langle q_1(cs)', \ldots, q_n(cs)'\rangle$ is used to denote the QoS attributes of the composite service cs, where $q_i(cs)'$ represents the ith attribute of the composite service cs.

Table 2. Aggregation Function

QoS Attribute	Sequential	Parallel	Loop	Conditional
Response Time	$\sum_{i=1}^{n} q(s_i)$	$\max_{i=1}^{n} q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^{n} q(s_i)$
Availability	$\prod_{i=1}^{n} q(s_i)$	$\prod_{i=1}^{n} q(s_i)$	$(q(s_1))^k$	$\min_{i=1}^{n} q(s_i)$
Cost	$\sum_{i=1}^{n} q(s_i)$	$\sum_{i=1}^{n} q(s_i)$	$k * (q(s_1))$	$\max_{i=1}^{n} q(s_i)$

3.2 QoS for Composite Services

A composite service S is constructed using a finite number of component services to reach a business goal. Let $C = \langle s_1, s_2, \ldots, s_n \rangle$ be the set of all component services that are used by S. The QoS of composite services is aggregated from the QoS of the component services, based on the service internal compositional structure, and the type of QoS attributes. Table 2 shows the aggregation functions for each compositional structure. We consider three types of QoS attributes: response time, availability and cost. For response time, in sequential composition, the response time of the composite service is aggregated by summing up the response time of each component service. As for parallel composition, the response time of the composite service is the maximum response time among that of each participating component service. For loop composition, the response time of the composite service is obtained by summing up the response time of the participating component service for k times, where k is the number of maximum iteration of the loop. And for conditional composition, the response time of the composite service is the maximum response time of n participating component services since it is not known that which guard is satisfied at the design phase. For availability, in sequential composition, the availability of the composite service is the product of that of all component services in the sequence because it means all component services are available during the sequential execution. It is similar to parallel and loop composition for aggregation of availability of the composite services. For conditional availability of the composite service, since one component service will be chosen at execution, therefore, we denote the availability as the minimum availability among all component services participated in the conditional composition. For cost, in sequential composition, the cost of the composite service is decided by the total cost of component services. For the conditional composition, the cost of the composite service is the maximum cost of n participating component services. Other common QoS attribute types can be aggregated in the similiar way with these three attributes. For example, QoS attributes like reliability share the same aggregation function with availability.

3.3 Labeled Transition Systems

The QoS-aware composite model in this work is defined using labeled transition systems (LTS). In the following we define various terminologies that will be used in this work.

Definition 1 (System State). *A system state s is a tuple (P, V, Q), where P is the composite service process and V is a (partial) variable valuation that maps variables to their values, Q is a vector which represents QoS attributes of the composite service.*

Two states are equivalent iff they have the same process P , the same valuation V and the same QoS vectors Q. Given a system state $s = (P, V, Q)$, $Q = \langle r, a, c \rangle$ is a vector with three elements, where $r, a, c \in \mathbb{R}_{\geq 0}$, and $0 \leq a \leq 1$. r, a, c represent the response time, availability, and cost of the state s. The response time, availability, and cost are calculated from the execution that starts at initial state s_0 up to the state s. Henceforth, we use the notation $Q(ResponseTime)$, $Q(Availability)$ and $Q(Cost)$ to denote the value of r, a, and c of QoS vector Q, respectively.

Definition 2 (Composite Service Model). *A composite service model $\mathcal{M}$ is a tuple (Var, P_0, V_0, F), where Var is a finite set of variables, P_0 is the composite service process, and V_0 is an initial valuation that maps each variable to its initial value. F is a function which maps component services to their QoS attribute vectors.*

Given a composite service (Var, P_0, V_0, F), an example of valuation V is $\{var_1 \mapsto 1, var_2 \mapsto \perp\}$, where var_1,$var_2 \in Var$, and $var_2 \mapsto \perp$ is used to denote that var_2 is undefined.

Definition 3 (LTS). *An* LTS *is a tuple $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$, where*

- *S is a set of states,*
- *$s_0 \in S$ is the initial state,*
- *Σ is a set of actions,*
- *$\rightarrow : S \times \Sigma \times S$ is a transition relation.*

For convenience, we use $s \xrightarrow{a} s'$ to denote $(s, a, s') \in \rightarrow$ and we denote the LTS of a BPEL service $\mathcal{M}$ as $L(\mathcal{M})$. Given a composite service model $\mathcal{M}$ = (Var, P_0, V_0, F), $L(\mathcal{M})$ =$(S, (P_0, V_0, Q_0), \Sigma, \rightarrow)$. Q_0 is the QoS attribute vector of the initial state, where the availability is 1, cost and response time are equal to 0. Give a state $s \in S$, $Enable(s)$ is denoted as the set of states reachable from s by one transition; formally, $Enable(s) = \{s' | s' \in S \wedge a \in \Sigma \wedge s \xrightarrow{a} s' \in \rightarrow\}$. An $execution$ π of $\mathcal{L}$ is a finite alternating sequence of states and actions $\langle\ s_0, a_1, s_1, \ldots, s_{n-1}, a_n, s_n\ \rangle$, where $\{s_0, \ldots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $0 \leq i < n$. We denote the execution π by $s_0 \xrightarrow{a_1} s_1 \xrightarrow{\cdots} s_{n-1} \xrightarrow{a_n} s_n$. A state s is called reachable if there is an execution that ends in s and starts in an initial state.

Assume a composite service model is $\mathcal{M}$ = (Var, P_0, V_0, F) and the LTS of $\mathcal{M}$ is $L(\mathcal{M})$ =$(S,\ s_0, \Sigma, \rightarrow)$. Every action $a \in \Sigma$ is triggered by an atomic activity. The atomic activities used in this work are $rec(S)$, $reply(S)$, $sInv(S)$, and $aInv(S)$, where S is the component service that the atomic activities are communicated with. For activities $rec(S)$ and $sInv(S)$, they are required to wait for reply from component service S before continuing, therefore their availability, cost and response time are equivalent to the availability, cost and response time of component service S. For activities $reply(S)$ and $aInv(S)$, they are not required to wait reply from the component service S, therefore they are regarded as internal operations. We assume the availability, cost and response time for an internal operations as 100%, \$0 and 0 ms respectively

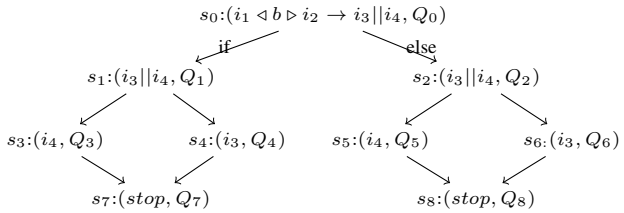

Fig. 2. LTS of CPS where i_1 is sInv(PBS), i_2 is sInv(CBS), i_3 is sInv(MS) and i_4 is sInv(SS)

(see Section 4.3 for discussion). Given two states $s = (P, V, Q)$, $s' = (P', V', Q')$, where $s, s' \in S$, $s \xrightarrow{a} s' \in \rightarrow$, and $a \in \Sigma$, we use the function $AtomAct(a)$ to denote the atomic activity that triggers the action a. As an example, given $s = (sInv(S) \rightarrow rec(S), V, Q)$ and $s' = (rec(S), V, Q)$, the function $AtomAct(a)$ returns the activity $sInv(S)$. We define the function $ResponseTime(a)$, $Availability(a)$ and $Cost(a)$ to map the action a to the response time, availability, and cost of the activity returned by $AtomAct(a)$. Using the previous example, $ResponseTime(a)$ is the response time of activity $sInv(S)$, which is essentially the response time of component service S.

The LTS of CPS as discussed in Section 2 is shown in Figure 2, where we omit the $Receive\ from\ user(fu)$, $Reply\ user(ru)$, all actions $a \in \Sigma$, and component V in the state for the reason of brevity. From state s_0, conditional activity $i_1 \lhd b \rhd i_2$ is enabled. Given that $\{b \mapsto \bot\}$, either i_1 or i_2 might be executed, therefore states s_1 and s_2 are evolved from state s_0. Noted that if guard *b* is defined, then only one branch is explored in the LTS. From state s_1, the flow activity $i_3||i_4$ is enabled, and both activities i_3 and i_4 are allowed to execute. This leads to states s_3 and s_4, respectively. State s_3 evolves into state s_7 after activity i_4 is executed. *stop* activity in state s_7 is a special activity which does nothing. Other states in LTS could be reasoned similarly. We assume that the upper bound on the number of iterations for loop activities is known, therefore, there is no recursive activities in BPEL.

4 Verification of Functional and Non-functional Requirements

This section is devoted to discuss how to verify combined functional and non-functional requirements based on the LTS semantics of web service composition. Current works only verify one aspect of requirements, either functional or non-functional requirement, however, these two aspects are inseparable. For example, some property such as in the CPS example is required to reply the user within 5 ms, involves both functional and non-functional requirements. Therefore, we propose an approach to combine functional and non-functional requirements.

4.1 Verification of Functional Requirement

To verify functional requirements of a BPEL program, LTS of the BPEL program is built from composite service model. We support the verification of deadlock-freeness, reachability of a state. To verify the LTL formulae, we make use of automata-based

on-the-fly verification algorithm [10], by firstly translating a formula to a Büchi automaton and then checking emptiness of the product of the system and the automaton. For fairness checking, we utilize the on-the-fly parallel model checking based on Tarjan strongly connected components (SCC) detection algorithms similar to [11].

4.2 Integration of Non-functional Requirement

In this section, we present our approach in integrating the non-functional requirements into verification framework. Different non-functional properties might have different aggregation functions for different compositional structures, and this poses a major challenge to integrate the non-functional properties into the functional verification framework. In the following, we adopt two different strategies in integrating the non-functional requirements. We first discuss our approach in integration of availability and cost, and following that, we discuss the integration of response time.

Integration of Availability and Cost. In this section, we present our approach to integrate the availability and cost to the verification framework. Given two states $s = (P, V, Q)$, $s' = (P', V', Q')$, where $s, s' \in S$, $s \xrightarrow{a} s' \in \rightarrow$, and $a \in \Sigma$, the availability and cost of state s' is calculated using the following formulae:

$$\begin{cases} s'.Q(availability) = s.Q(availability) * Availability(a) \\ s'.Q(cost) = s.Q(cost) + Cost(a) \end{cases} \tag{1}$$

Example. We illustrate the integration using the LTS of CPS as shown in Figure 3. In state s_0, it has the initial availability of 1 and initial cost of \$0. From state s_0, it evolves into state s_1 after invocation of i_1. Since i_1 has availability of 0.9 and cost of \$3 (refer to Table 1), therefore the resulting QoS vector of state s_1 is $\langle r_1, 1 * 0.9, 0 + 3\rangle = \langle r_1, 0.9, 3\rangle$. From state s_1, it evolves into state s_3 after the invocation of i_3, and since i_3 has availability of 0.8 and cost of \$2, the resulting QoS vector of state s_3 is $\langle r_3, 1 * 0.9 * 0.8, 0 + 3 + 2\rangle = \langle r_3, 0.72, 5\rangle$. Other states are calculated similarly.

In general, given an execution $\pi = s_0 \xrightarrow{a_1} s_1 \xrightarrow{\cdots} s_{n-1} \xrightarrow{a_n} s_n$ in $L(\mathcal{M})$, where $\{s_0, \ldots, s_n\} \in S$ and $s_i \xrightarrow{a_{i+1}} s_{i+1} \in \rightarrow$, for all $0 \leq i < n$

$$\begin{cases} s_{i+1}.Q(availability) = s_0.Q(availability) * \prod_{m=1}^{i} Availability(a_m) \\ s_{i+1}.Q(cost) = s_0.Q(cost) + \sum_{m=1}^{i} Cost(a_m) \end{cases} \tag{2}$$

with $s_0.Q = \langle 0, 1, 0\rangle$.

Integration of Response Time. One might naively think that we can adopt the method of calculating the cost as the method for calculating the response time. However, this would result in incorrect result. Refer to Figure 3, the value of response times r_2, r_5, r_6, and r_8 will be 2 ms, 5 ms, 3 ms, and 6 ms respectively by using the method of calculating the cost in Section 4.2. In such case the value of r_8 is incorrect. The reason

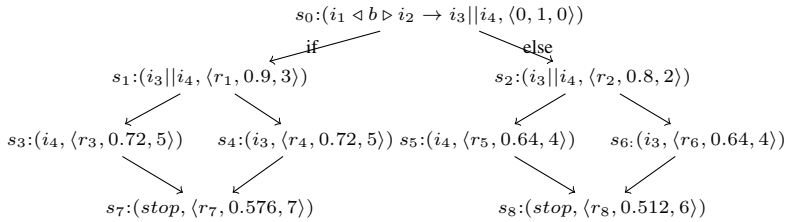

Fig. 3. LTS of CPS with Availability and Cost, where i_1 is sInv(PBS), i_2 is sInv(CBS), i_3 is sInv(MS) and i_4 is sInv(SS)

is that it should be calculated as maximum of value of r_5 and r_6, since parallelism allows both i_3 and i_4 to be executed simultaneously, and the total time for the response time is decided by the maximum response time of i_3 and i_4. A challenge to evaluate the maximum time in state s_8 is that the information of parallism in state s_2 ($i_3||i_4$) is removed in state s_5 and state s_6 (only left with i_3 or i_4). In order to retain this information, we preprocess the BPEL service model $\mathcal{M}$ to associate with a time tag which will be used to calculate the response time in the LTS generation stage.

Algorithm 1 presents the main algorithm for preprocessing. Given a BPEL process P_0, $TagTime(P_0, x)$ returns the process P_0' which is the process P_0 with its internal activities associated with time tags. Given each activity $Acv \in P_0$, a value $timetag \in \mathbb{R}_{\geq 0}$ is associated with Acv, denoted as $Acv.timetag$. $Acv.timetag$ represents the total time delay from the start of process P_0, up to the completion of activity Acv. In the following, we describe the Algorithm 1. The function $TagTime(P_0, x)$ is used to calculate the total time delay from the start of process P_0 up to the completion of activity Acv. Variable $x \in \mathbb{R}_{\geq 0}$ is the the total time delay from the start of process P_0 to the point just before the execution of activity Acv. Lines 1, 5, 9 and 11 are used to detect the structure of the activities. At line 1, if P is detected to be a sequential activity, activity A will be tagged with the delay x (line 2) as A is triggered once P is triggered. Subsequently, activity B will be tagged. Since activity B is executed after the completion of activity A, therefore the x is set to be the value of $A.timetag$ (line 3). Finally, the $timetag$ of P is the same as $timetag$ of B, since the completion of activity B implies the completion of execution of process P (line 4). At line 5, if P is detected to be a concurrent or conditional activity, activity A and activity B will be tagged with value x (lines 6 and 7), since A and B are triggered at the same time once P is triggered. At line 8, the $timetag$ of P is the maximum value of $timetag$ of A and B (refer to Section 3.2 for details). If P is detected to be a synchronous receive activity or invocation activity, the $timetag$ of P is set to the sum of x and $ResponseTime(P)$ (line 10).

Example. In the following, we use an example to illustrate how to calculate the response time for each state in the LTS. Given initial service process $P_0 = sInv(PBS) \triangleleft b \triangleright sInv(CBS) \rightarrow (sInv(MS)||sInv(SS))$, we denote $P_0' = TagTime(P_0, 0)$ and

$$P_0' = [[[sInv(PBS)]^1 \triangleleft b \triangleright [sInv(CBS)]^2]^2 \rightarrow [[sInv(MS)]^5||[sInv(SS)]^3]^5]^5$$

Algorithm 1. Algorithm TagTime(P, x)

input : P, the BPEL process
input : x, the delay from the start to execution of process P
output: P', process P with time tag

1 **if** $P\ is\ A \rightarrow B$ **then**
2 $\quad TagTime(A, x)$;
3 $\quad TagTime(B, A.timetag)$;
4 $\quad P.timetag \leftarrow B.timetag$;
5 **else if** $P\ is\ A||B\ or\ A \lhd b \rhd B$ **then**
6 $\quad TagTime(A, x)$;
7 $\quad TagTime(B, x)$;
8 $\quad P.timetag \leftarrow max(A.timetag, B.timetag)$;
9 **else if** $P\ is\ rec(S)\ or\ sInv(S)$ **then**
10 $\quad P.timetag \leftarrow x + ResponseTime(P)$;
11 **else if** $P\ is\ reply(S)\ or\ aInv(S)$ **then**
12 $\quad P.timetag \leftarrow x$;

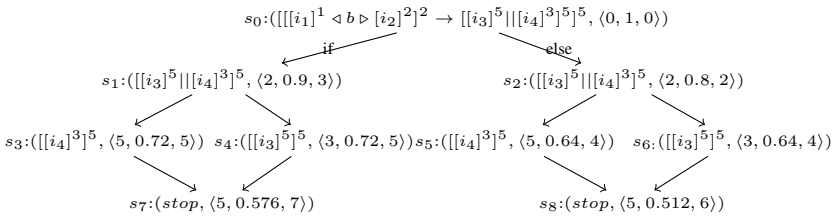

Fig. 4. LTS of CPS with Response Time, Availability and Cost, where i_1 is sInv(PBS), i_2 is sInv(CBS), i_3 is sInv(MS) and i_4 is sInv(SS)

where for each activity $A \in P$, $[A]^t$ is used to denote the activity A with $A.timetag = t$. Next, in the LTS generation stage, Algorithm 2 is used to calculate the response time for each state.

Given the process P of some state $s \in S$, $CalculateTime(P)$ in Algorithm 2 returns the total response time $t \in \mathbb{R}_{\geq 0}$ from the initial state s_0 to s'. The value t is assigned to Q($responseTime$) for state s'. Lines 1, 6, 11 are used to detect the structure of the activities. We introduce a special activity $skip$ to denote the completion of execution of an atomic activity. $skip$ is used for the purpose of calculating the response time, and it will be removed after the calculation. At line 1, if P is detected to be a sequential activity, the activity A is then checked whether it is a $skip$ activity. If it is (line 2), which implies that activity A has finished execution, $A.timetag$ is returned (line 3). Otherwise, $CalculateTime(A)$ is invoked in order to determine the response time (line 5). At line 6, if P is detected to be a concurrent activity or conditional activity, A and B will be determined whether both are $skip$ activities. If it is (line 7), which implies that P has finished execution, $P.timetag$ is returned (line 8).

Algorithm 2. Algorithm CalculateTime(P)

input : P, BPEL process with time tagged
output: $t \in \mathbb{R}_{\geq 0}$, the time delay from the start of initial process P_0 to the completion of P

```
1  if P is A → B then
2      if A is skip then
3          return A.timetag;
4      else
5          return CalculateTime(A);
6  else if P is A||B or A ◁ b ▷ B then
7      if A is skip and B is skip then
8          return P.timetag;
9      else
10         return CalculateTime(PreviousActive(P));
11 else if P is skip then
12     return P.timetag;
```

Otherwise, $CalculateTime(PreviousActive(P))$ is invoked in order to obtain the response time (line 10) where $PreviousActive(P)$ is used to denote previous execution activity. For example, given $s = (i_1||i_2, V, Q)$, $s' = (skip||i_2, V', Q')$, and $s \xrightarrow{a} s' \in \rightarrow$, $PreviousActive(skip||i_2)$ will return $AtomAct(a) = i_1$. At line 11, P is determined to be a $skip$ activity implies that P has finished execution, therefore, $P.timetag$ is returned (line 12). The value of $timetag$ for each BPEL process is obtained using Algorithm 1.

Example. In Figure 4, given the initial state s_0, there are two branches due to the conditional process. If sInv(PBS) is executed, it will evolved into state s_1 with process P_1' where

$$P_1' = [[[skip]^1]^2 \rightarrow [[sInv(MS)]^5||[sInv(SS)]^3]^5]^5$$

By running the Algorithm 2 for PBS to get the response time of PBS, it will return the value 2, therefore state s_1 has the response time of 2 ms. After the calculating the response time, the $skip$ are removed from P_1', which result in process $P_1 = [[sInv(MS)]^5||[sInv(SS)]^3]^5$ as shown in Figure 4. The calculation of other states is similar.

4.3 Discussion

If a system is verified that it does not satisfy the requirement that the response time is less than a ms in a state s, where $a \in \mathbb{R}_{\geq 0}$, it does not necessarily mean that such constraint will be violated in the state s during the execution. The response time is served as an estimated reference value. Furthermore, we do not take the response time, cost, and availability of internal operations into account. In reality, such information can be estimated using runtime monitoring method [12].

Table 3. Experiment Results

Services	Property	Result	#State	#Transition	Time(s)
CPS	(replyUser ∧ (responseTime>5))	invalid	21	29	0.0087
	□ responseTime≤5	valid	26	36	0.0089
	□ availability>0.6	valid	26	36	0.0083
LS	Reach (replyUser ∧ (responseTime>6))	invalid	106	241	0.0584
	□ responseTime≤6	valid	242	572	0.1866
TAS	Reach (replyUser ∧ (responseTime>3))	invalid	128	287	0.0631
	□ responseTime≤3	valid	264	622	0.0642
	Reach (replyUser ∧ (availability≤0.3))	invalid	128	287	0.0437

5 Evaluation

We evaluate our approach using three case studies. Each case study is a composite service represented as a BPEL process. The experiment data was obtained on a system using Intel Core I7 3520M CPU with 8GB RAM. The experimental results are summarized in Table 3.

5.1 Computer Purchasing Service (CPS)

As described in Section 2, CPS is used for allowing users to purchase a computer online using credit cards. The workflow of CPS is illustrated in Figure 1. The property $Reach$ $(replyUser \wedge (responseTime{>}5))$ is to verify whether the activity $reply\ user$ (ru) can be reached with response time more than $5\ ms$. The result is invalid as shown in Table 3, which implies that if the $reply\ user$ (ru) is reached, it will be always be less than $5\ ms$, which is the intended outcome we need. Properties □ $reponseTime{\leq}5$ and □ $availability{>}0.6$ are LTL formulas, which are invariant properties denoted that the CPS's response time must always be less than two milliseconds and the CPS's availability is always larger than 50%. These two properties are both verified to be valid in the CPS system. The number of visited states, total transitions and time used for verification are listed in Table 3.

5.2 Loan Service (LS)

The goal of a Loan Service (LS) is to provide users for applying loans. The loan approval system has several component systems, Loan Record Service (RS), Loan Approval Service (LAS), Customer Details Service (CDS), Customer Loan History Service (CLHS), Customer Credit Card History Service (CCHS), Customer Employment Information Service (CES) and Customer Property Information Service (CPIS). Upon receiving the request from a customer, CDS will be invoked synchronously. If the requested load amount is less than $10000, CES is invoked and then RS is invoked to record the customer's loan information. After that, loan approval message will be

replied to the customer. Otherwise, if the requested amount is not less than $10000, CLHS, CCHS, CES and CPIS are invoked concurrently to obtain more detailed information about the customer. Upon receiving all replies, LAS is invoked to determine whether to approve the load request of the customer or not. If the request is approved, RS is invoked synchronously and then loan approval message will be replied to the customer, otherwise, loan failure message will be replied to the customer. Two properties are verified for LS as listed in Table 3, we omit the discussion of the properties as they are similar to the properties of CPS.

5.3 Travel Agency Service (TAS)

Travel Agency Service (TAS) provides a service that helps users to arrange the flight, hotel, transport, etc., for a trip. Once the request is received from the user, Hotel Booking Service (HBS), Fight Booking Service (FBS), Local Transport Service (LoTS) and Local Agent Service (LAS) are triggered to search for available hotel, flight, local transportation and local travel agent concurrently that fulfill the user's requirements. If all four services have returned non-empty results, Record Booking Information Service (RBS) and Notify Agent Service (NAS) are invoked concurrently to store detailed booking information into the system and notify the agent about the customer's details. Finally, TAS replies the detailed booking information to the user. Otherwise, TAS replies booking failure result to the user. Three properties are verified for TAS as listed in Table 3. Properties $Reach\ (reply\,User \wedge (responseTime{>}3))$ and $\Box\, responseTime{\leq}3$ are similar to the properties verified in CPS, therefore we omit discussion of these two properties here. Property $Reach\ (replyUser\ \wedge\ (availability{\leq}0.3))$ is to verify whether $reply\ user\ (ru)$ can be reached with the availability less than 0.3. The result is invalid as shown in Table 3, which implies that if the $reply\ user\ (ru)$ is reached, the availability is always greater than 0.3, which is the intended result that we need.

The experiment shows that our approach can be used to verify the combined functional and non-functional property for real-world BPEL program efficiently.

6 Related Work

A number of approaches have been proposed to deal with requirements of web service composition. These work can be divided into two major directions. One direction is to transform WS-BPEL processes into intermediate formal models specified in some formal languages and then verify the functional behaviors of the service composition based on the formal models. Foster et al. [13] translate BPEL processes into finite state processes notation. Qian et al. [14] transform BPEL processes into timed automata, and then use Uppaal as the model checker to verify the functional properties of the TA model, such as reachability. In [9,15], the authors transform BPEL processes into Promela models and then use SPIN to verify the models. In [16], Yu et al. present a a lightweight specification language called PROPOLS to describe the temporal logic in a BPEL process. In [17], we translate processes into a new formal language proposed with formal operational semantics by themselves. Different from these approaches, our current approach verifies functional properties of BPEL processes based on its semantics, thus it does not need to be translated into any other formal languages since there are

some disadvantages of using intermediate models as mentioned in Section 1. More important, our work combines verification of functional and non-functional requirements while works above only consider functional verification, which cannot verify functional and non-functional requirements at the same time.

Another direction has its focus on the non-functional aspect of BPEL processes. In [8], Koizumi and Koyama propose a performance model to estimate the processing execution time by integrating a Timed Petri Net model and statistical models. However, it only focuses on one type of non-functional requirements and does not consider the functional behaviors. In [7], Fung et al. propose a message tracking model to support QoS end-to-end management of BPEL processes. This work is based on the run-time data, which needs the deployment of the services, in addition, it does not consider the functional requirements of BPEL processes. Our approach verifies both functional and non-functional requirements at design time, which can detect errors at the early stage. In [18], Xiao et al. propose a framework to use the simulation technique to verify the non-functional requirements before the service deployment, which is similar to our work. While their work only focus on non-functional aspect, our work supports verification of combined functional and non-functional properties. In [19], we propose a fully automatic approach for synthesis the local time requirement based on the given global time requirement of Web service composition. Different from them, our work focuses on checking LTL constraint satisfaction. And to the best of our knowledge, our work is the first one to verify combined functional and non-functional properties.

7 Conclusion

In this paper, we have illustrated our approach to verify combined functional and non-functional requirements (i.e., availability, response time and cost) for web service composition. Furthermore, our experiments show that our approach can work on real-world BPEL programs efficiently. We plan to further improve and develop the technique presented in this paper. Firstly, we will consider various heuristics that could be used to reduce the number of states and transitions. Secondly, we will investigate applying state reduction techniques, such as partial order reduction [20], to improve the efficiency of our approach. Lastly, our work could be extended to other domains such as sensor networks.

References

1. Chinnici, R., Moreau, J.J., Ryman, A., Weerawarana, S.: Web services description language (WSDL) version 2.0, http://www.w3.org/TR/wsdl20/
2. Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J.J., Nielsen, H.F., Karmarkar, A., Lafon, Y.: Simple object access protocol (SOAP) version 1.2, http://www.w3.org/TR/soap12/
3. OASIS Web Service Business Process Execution Language (WSBPEL) Technical Committee: Web Services Business Process Execution Language Version 2.0 (2007), http://www.oasis-open.org/specs/#wsbpelv2.0
4. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? In: ESEC/FSE 2011, pp. 26–36. ACM (2011)

5. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
6. Foster, H., Uchitel, S., Magee, J., Kramer, J.: WS-Engineer: A model-based approach to engineering web service compositions and choreography. In: Test and Analysis of Web Services, pp. 87–119 (2007)
7. Fung, C.K., Hung, P.C.K., Wang, G., Linger, R.C., Walton, G.H.: A study of service composition with qos management. In: ICWS 2005, pp. 717–724 (2005)
8. Koizumi, S., Koyama, K.: Workload-aware business process simulation with statistical service analysis and timed petri net. In: ICWS 2007, pp. 70–77. IEEE CS (2007)
9. Nakajima, S.: Lightweight formal analysis of web service flows. Progress in Informatics 2, 57–76 (2005)
10. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Form. Methods Syst. Des. 1, 275–288 (1992)
11. Sun, J., Liu, Y., Dong, J.S., Pang, J.: Pat: Towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
12. Moser, O., Rosenberg, F., Dustdar, S.: Non-intrusive monitoring and service adaptation for ws-bpel. In: WWW 2008, pp. 815–824. ACM (2008)
13. Foster, H., Uchitel, S., Magee, J., Kramer, J.: WS-Engineer: A model-based approach to engineering web service compositions and choreography. In: Baresi, L., Nitto, E.D. (eds.) Test and Analysis of Web Services, pp. 87–119. Springer (2007)
14. Qian, Y., Xu, Y., Wang, Z., Pu, G., Zhu, H., Cai, C.: Tool support for bpel verification in activebpel engine. In: ASWEC 2007, pp. 90–100 (2007)
15. Li, B., Zhou, Y., Pang, J.: Model-driven automatic generation of verified bpel code for web service composition. In: APSEC 2009, pp. 355–362. IEEE CS (2009)
16. Yu, J., Manh, T.P., Han, J., Jin, Y., Han, Y., Wang, J.: Pattern based property specification and verification for service composition. In: Aberer, K., Peng, Z., Rundensteiner, E.A., Zhang, Y., Li, X. (eds.) WISE 2006. LNCS, vol. 4255, pp. 156–168. Springer, Heidelberg (2006)
17. Sun, J., Liu, Y., Dong, J.S., Pu, G., Tan, T.H.: Model-based methods for linking web service choreography and orchestration. In: APSEC 2010, pp. 166–175. IEEE CS (2010)
18. Xiao, H., Chan, B., Zou, Y., Benayon, J.W., O'Farrell, B., Litani, E., Hawkins, J.: A framework for verifying sla compliance in composed services. In: ICWS 2008, pp. 457–464 (2008)
19. Tan, T.H., André, É., Sun, J., Liu, Y., Dong, J.S., Chen, M.: Dynamic synthesis of local time requirement for service composition. In: ICSE 2013, pp. 542–551 (2013)
20. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL 2005, pp. 110–121. ACM (2005)