Vietnam J Comput Sci (2017) 4:111–126
DOI 10.1007/s40595-016-0079-8

REGULAR PAPER

A bitwise-based indexing and heuristic-driven on-the-fly approach for Web service composition and verification

Khai T. Huynh[1] · Tho T. Quan[1] · Thang H. Bui[1]

Received: 6 February 2016 / Accepted: 27 August 2016 / Published online: 9 September 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract During the last decade, software engineering community has witnessed the emerging of SOA architecture, where Web services play a crucial role. It prompts the concept of Web Service Composition (WSC). Even though interesting, this issue posed some remarkable challenges, one of which is constraint handling. To be more precise, one needs to ensure that the composite Web service fulfills, at the same time, many constraints, including functional constraints, Quality of Service (QoS), and the execution order of the component services, or temporal relations. Those constraints are of different natures, thus finding an efficient verification on all kinds of constraints during the composition process is by no means a trivial task. Backed by a solid foundation of temporal logic, model checking (MC) is a suitable approach to handle this issue. However, MC-based approach suffers from the infamous problem of state-space explosion, making it limited when applied to real-life situations. The work in this paper addresses the problem by proposing various approaches for handling the state-space exploration, including (i) an introduction of an LTS-based model known as LTS4WS, which can avoid generating full schema of Web service composition and allow on-the-fly verification on the state space; (ii) heuristics strategies to find the best potential composition, and (iii) a bitwise-based indexing mechanism for fast location of suitable Web services. All of those approaches are unified in a single tool, known as WSCOVER. As a result, a significant improvement of performance has been made, especially as compared with the other existing works in the same field.

✉ Khai T. Huynh
htkhai@cse.hcmut.edu.vn

Tho T. Quan
qttho@cse.hcmut.edu.vn

Thang H. Bui
thang@cse.hcmut.edu.vn

[1] Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam

Keywords Web service composition · On-the-fly Web service composition and verification · Web service composition tool · Bitwise-based Web service indexing · Heuristic-driven Web service composition

1 Introduction

1.1 Web service composition and verification

Nowadays, Web Service Composition (WSC) has been raised as an important issue of Service-Oriented Architecture (SOA) [1]. WSC is the process of creating the complexly structured composite Web services from component Web services [2], which has been one of the challenging problems in recent years, when the number of provided component Web services increases and the composition requirements from users become more complex. When composing Web services, in addition to the requirements on functional properties, also known as hard constraints, the requirements on Quality of Service (QoS) properties, also known as soft constraints, are also a very important factor determining the outcome of the composition. There are many QoS properties of Web services, such as response time, execution cost, availability, or reputation, etc. In some circumstances, many services that satisfy hard constraints cannot be used in a composition, because of the dissatisfactions of soft constraints.

Let us consider the following example. Suppose that a user is organizing his trip using the Web services presented in Table 1. By providing information on travel place ($Sightseeing$) and the traveling dates ($Dates$), the user

Table 1 Travel Booking Web service repository

#	Service name	Input(s)	Output(s)	respTime
1	HotelReserveService (HR)	Dates, Hotel	HotelReservation	5
2	CityHotelService (CH)	City	Hotel	3
3	HotelCityService (HC)	Hotel	City	3
4	HotelPriceInfoService (HP)	Hotel	Price	10
5	SightseeingCityService (SC)	Sightseeing	City	2
6	SightseeingCityHotelService (SCH)	Sightseeing	City, Hotel	16
7	CitySightseeingService (CS)	City	Sightseeing	4
8	ActivityBeachService (ABS)	Activity	Beach	5
9	AreaWeatherService (AWS)	Area	Weather	5
10	CityWeatherService (CWS)	City	Weather	5

Table 2 Requirements for travel booking Web service

Constraint	Value	
Hard constraint:	Input:	$Dates, Sightseeing$
	Output:	$Price, HotelReservation$
Soft constraint:	$respTime \leq 30$	
Temporal relation:	$\Box(\neg HotelReservation \text{ U } Price)$	

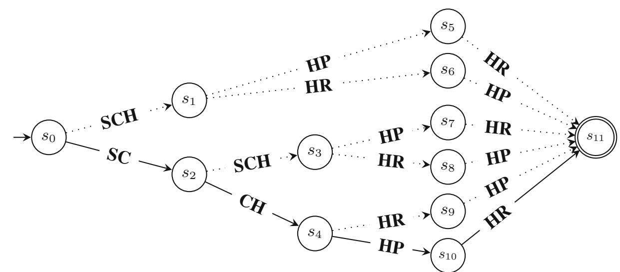


Fig. 1 Full composition schema for travel booking problem

wants to find hotel booking price ($Price$) and the hotel reservation information ($HotelReservation$) of the hotels near the $Sightseeing$. Obviously, as observed in Table 1, there is no individual Web service that can completely fulfill the requirement from the user. Thus, one needs to find a composition of the available Web services. Beside the requirements on the above functional constraints (hard constraints), the user can also specify soft constraints, such as “The total response time of the composite Web service should not exceed 30 s”. In addition, the user may also want a certain order of service execution, or temporal relation. For instance, we may also need to obtain the information about $Price$ before proceeding on $HotelReservation$. All of requirements discussed are summarized in Table 2.

Typically, a WSC problem is a Satisfiability (SAT) problem, such that the composition of Web services is considered as a process that creates new logic formula which can satisfy the target formula, known as goal. Verifying the satisfaction of composition with the constraints becomes a theoretical verification problem. Since the hard constraint is typically represented by a traditional form of logic, often as First-Order Logic [3], the verification of whether a WSC satisfies hard constraint or not can adopt a traditional approach, such as SAT solver, in the classic AI planning approach [4].

However, the formal verification of soft constraint becomes more difficult, because the soft constraint conditions may be mathematical expressions representing QoS properties. These functions can be of the non-linear form, which causes serious difficulties for the current provers.

Due to such characteristics, research on the verification of composite Web services often focuses on hard constraint. Recently, an approach that can verify combination of both hard and soft constraints is introduced in [5]. In this approach, first of all, all compositions which satisfy the hard constraint are created, forming a full composition schema. For example, Fig.1 illustrates a full composition schema for the travel booking problem. Subsequently, model checking can be used to verify whether the compositions satisfy the soft constraint or not. As a result, only compositions satisfying soft constraints, for instance, denoted by solid line in Fig.1, are retained. Furthermore, concerning the temporal property described in Table 2, only the composition of $\{s_0 \rightarrow s_2 \rightarrow s_4 \rightarrow s_{10} \rightarrow s_{11}\}$ should be kept. To our knowledge, this is the sole work claimed to verify hard and soft constraints at the same time. Unfortunately, this approach must be extremely expensive for the creation of full schema, which is a classic NP-hard problem. Intuitively, one of the ways that can handle this is to build the schema in an on-the-fly manner to find a satisfied composition without building a full schema in advance. However, unfortunately, in this approach, the full composition schema is used as the model to be verified, and hence, it has to be constructed beforehand.

1.2 Web service indexing

When the number of Web services in repository increases, the cost of composition problem also increases significantly.

There have been many given suggestions to resolve this problem, such as clustering [6,7], or indexing [8,9]. In the clustering approach, we have to address a variety of problems, such as how to compute the similarity between services, how many clusters should be generated, or which clustering algorithm should be chosen, etc.

Alternatively, indexing is considered as a simple and effective approach to reduce the composition time. In literature, there are many ways to index, such as using the hashtable [8] or using the weighted vector [9]. However, in these approaches, the construction of index structure and the application of the processing functions are in high complexity. In general, Web service indexing is a preprocessing step of the Web service composition process. Once an index structure is available, the composition process can quickly retrieve the needed Web services. However, the indexing process is definitely computationally expensive, and it also requires additional space to store the index table. Therefore, we need an effective and low-cost method to build the index table, which should be updated easily when the stored data are modified.

Contributions In this paper, we introduce an extension of our previous work [10], which proposes a novel and effective Web service composition approach. It bases on the model checking approach, but does not require prior full schema of compositions. Instead, the composition will be generated according to the path condition on the state space when the model is verified. Thus, if we use an on-the-fly model checker, such as PAT [11], the composition and verification will be processed in an on-the-fly manner, as well. As a result, we can find a solution that simultaneously satisfies the hard and soft constraints represented by a temporal logic[1] formula, without the need of creating a full composition schema. In addition, this approach makes room for some performance improvement when the outcome of a composition step can be used to optimize the candidate selection for the next step. This optimization process can be implemented by some heuristic rules.

The contribution of the work on [10] is as follows.

- We propose an approach to represent the Web services by the *Labelled Transition System* (LTS), known as *LTS4WS*, to serve for the application of model checking for Web service composition.
- We use the LTS4WS model to verify the hard and soft constraints on WSC. This model checking approach allows us to verify temporal relation on the constraints as well.
- We apply some heuristic based on the characteristics of Web service when performing model checking, so verification performance is improved visibly.

[1] Currently, we only support Linear Temporal Logic (LTL).

To extend this work, we propose a bitwise-based indexing technique to organize the Web services in the repository to support the Web service retrieval process more accurately and efficiently. For the motivating example above, our approach just needs to traverse 18 states in the case of using on-the-fly tactic, and is further reduced to 10 states when heuristics are applied (as compared with 108 states needed to be visited to make the full schema in PORSCE II [4]).

The contributions of our extension can be summarized as follows.

- We proposed an approach which combines indexing and model checking for composition and verification of Web services. Compared with [10], this work is enhanced with indexing technique to enjoy a significant improvement of performance.
- We presented a bitwise-based approach to present Web services and indexing technique. This representation allows us to use the bitwise operators in processing and further enjoy more improvement on performance.
- The experiments are performed on multiple real data sets and the results show the effectiveness of this approach compared with [10] and other approaches.

Regarding the novelty of the work presented in this paper, we want to note that using model checking to solve the Web service composition or Web service verification has already been reported in [5,10,12–14]. Likewise, Web service indexing for faster composition is also not a new idea, see [8,9,15]. However, this paper is the first work proposing the combination of these two techniques.

Outline The rest of the paper is organized as follows. Section 2 presents a model for the Web service composition problem. In Sect. 3, we present heuristics to improve the Web service composition performance. We propose a bitwise-based Web service indexing approach in Sect. 4, together with a case study. Then, in Sect. 5, we present our experimentations from a repository of real Web services. In Sect. 6, we present the related works. The conclusion and future work are discussed in Sect. 7.

2 LTS4WS—the model for Web service composition

In this paper, we formalize the composition task as a state-based searching problem. Each state corresponds to a composition of multiple Web services, from which new state can be generated by extending the current composition with another Web service, bringing a new composition. That is, we regard the set of all possible Web services as a model, whose states are feasible combinations of them. Meanwhile,

user requirements are regarded as properties, which can be verified over the model using the model checking techniques, in an on-the-fly manner. To carry out this study, our model is developed based on the following definitions.

Definition 1 (Web Service) A *web service* $\mathcal{W}$ is a six-tuple $\mathcal{W} = (\mathcal{N}, \mathcal{I}, \mathcal{O}, \mathcal{P}, \mathcal{E}, \mathcal{Q})$, where

- $\mathcal{N}$ is a string representing the unique name of $\mathcal{W}$.
- $\mathcal{I} = \{t_1^i, t_2^i, \ldots, t_n^i\}$ is a set of input functional properties. We denote $\widetilde{t_j^i}$ as an *input logical term* representing the current informative status of the property t_j^i. $\widetilde{t_j^i} = true$ means that the information of t_j^i is currently available and vice versa.
- $\mathcal{O} = \{t_1^o, t_2^o, \ldots, t_m^o\}$ is a set of output functional properties. We denote $\widetilde{t_k^o}$ as an *output logical term* representing the current informative status of the property t_k^o. Likewise, $\widetilde{t_k^o} = true$ means that the information of t_k^o is currently available and vice versa.
- $\mathcal{P}$ is a logic expression representing the *pre-condition* of $\mathcal{W}$ that must hold before $\mathcal{W}$ is invoked. It is simply a *conjunctive normal form* of all input logical terms, as $\mathcal{P} = \widetilde{t_1^i} \wedge \widetilde{t_2^i} \wedge \ldots \wedge \widetilde{t_n^i} = \bigwedge_j^n \widetilde{t_j^i}$.
- $\mathcal{E}$ is a set of assignment expressions describing the *effect* after $\mathcal{W}$ is invoked. $\mathcal{E}$ has the form of $\{\forall k = 1..m : \widetilde{t_k^o} \triangleq true\}$, where $\triangleq$ is the assignment operator. Because $\widetilde{t_k^o}$ is a logical term, we can simply represent it as $\{\forall k = 1 \ldots m : \widetilde{t_k^o}\}$ or $\{\widetilde{t_1^o}; \ldots; \widetilde{t_m^o};\}$.
- $\mathcal{Q}$ is a set of QoS properties, each of which is a pair of $\langle name : value \rangle$, where *name* is the name of the property and *value* is a numerical amount which evaluates the value returned by $\mathcal{W}$ w.r.t this property.

Example 1 Let us consider the first Web service in Table 1, which provides the hotel reservation information (*HotelReservation*) of a specific hotel (*Hotel*) and the reservation dates (*Dates*). The response time (*respTime*) of this service is of 5 s. This Web service is described by Definition 1 as follows.

$$\begin{aligned} \mathcal{W} = (&\mathcal{N} = HR, \\ &\mathcal{I} = \{Hotel, Dates\}, \\ &\mathcal{O} = \{HotelReservation\}, \\ &\mathcal{P} = \widetilde{Hotel} \wedge \widetilde{Dates}, \\ &\mathcal{E} = \{Hotel\widetilde{Reservation}\}, \\ &\mathcal{Q} = \{respTime : 5\}) \end{aligned}$$

The logic expression $\widetilde{Hotel} \wedge \widetilde{Dates}$ is the pre-condition that must hold before the Web service is invoked. It means that to invoke the service *HR* (*HotelReserveService*), we must have the information of *Hotel* and *Dates* to reserve this hotel. The effect of this Web service invocation is that we have the reservation information of this hotel (*HotelReservation*), i.e., $Hotel\widetilde{Reservation} = true$. It needs five units of time (*respTime*: 5) in execution.

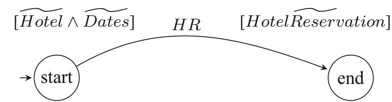


Fig. 2 Visual representation of an LTS

Definition 2 *(Labelled Transition System)* A *Labelled Transition System* (LTS) is a five-tuple $\mathcal{L} = (V, S, s_0, L, \delta)$, where

- V is a set of variables,
- S is a set of states,
- $s_0 \in S$ is the initial state,
- L is a set of *action labels*,
- $\delta : S \times L \to S$ is a transition relation, where $(s, a, s') \in \delta$ is denoted as $(s \times a \to s')$. A transition may also have a *pre-condition* (or the *guard*), a logical expression built over V, which must always hold before the transition is fired. In addition, a transition may also have an *effect*, a set of expressions built over V, which expresses the effect after firing the transition. A LTS that supports those kinds of transitions is called *guarded LTS*, whose transition is represented as:

  [*guard*]*transition*[*effect*]

Example 2 Given a simple LTS containing two states $\{start, end\}$, $start$ is the initial state, as shown in Fig. 2, and a set of variables V consist of $\widetilde{Hotel}$, $\widetilde{Dates}$, and $Hotel\widetilde{Reservation}$. This LTS describes that in the initial state, if we fire the transition (or invoke the service) *HR* (*HotelReserveService*), the system will switch to state *end*. The LTS is visually described in Fig. 2 and represented as $System = (V, S, s_0, L, \delta)$, where

- $V = \{\widetilde{Hotel}, \widetilde{Dates}, Hotel\widetilde{Reservation}\}$,
- $S = \{start, end\}$,
- $s_0 = start$,
- $L = \{HR\}$,
- $\delta = \{[\widetilde{Hotel} \wedge \widetilde{Dates}]start \times HR \to end$ $[Hotel\widetilde{Reservation}]\}$

In Fig. 2, we describe the representation of Web service *HR (HotelReservation)* as a transition in an LTS system, where the pre-condition and effect of the Web service are also used as the guard and effect of the transition. In general, all of Web services can be represented as this transition with appropriate guards and effects.

Definition 3 (LTS for Web Services)Let $WS = \{\mathcal{W}_1, \mathcal{W}_2, \ldots, \mathcal{W}_n\}$ be a set of Web services, where $\mathcal{W}_i = (\mathcal{N}_i, \mathcal{I}_i, \mathcal{O}_i, \mathcal{P}_i, \mathcal{E}_i, \mathcal{Q}_i)$ as defined in Definition 1. A *LTS for Web Services* (LTS4WS) of WS is a guarded LTS $\mathcal{L}^{WS} = (V, \{s_0\}, s_0, L, \delta)$, where

- $V = (\bigcup_i^n \mathcal{I}_i) \bigcup (\bigcup_i^n \mathcal{O}_i)$.
- $\{s_0\}$ is a set of states, which has only one state s_0,
- s_0 is the initial state,
- $L = \{\mathcal{N}_1, \mathcal{N}_2, \ldots, \mathcal{N}_n\}$,
- δ is a transition relation of the form $[\mathcal{P}_i]s_0 \times \mathcal{N}_i \rightarrow s_0[\mathcal{E}_i]$.

Example 3 The LTS4WS model of the ten Web services in Table 1 is defined as follows.

- The set of variables: $V = \{\widetilde{Dates}, \widetilde{Hotel}, \widetilde{HotelReservation}, \widetilde{City}, \widetilde{Price}, \widetilde{Sightseeing}, \widetilde{Activity}, \widetilde{Beach}, \widetilde{Area}, \widetilde{Weather}\}$
- The set of states: $\{s_0\}$
- The initial state: s_0
- The set of label actions is the acronym of the name of web services: L=$\{HR, CH, HC, HP, SC, SCH, CS, ABS, AWS, CWS\}$
- The transition relations: $\delta = \{\ [\widetilde{Hotel} \wedge \widetilde{Dates}]\ s_0 \times HR \rightarrow s_0\ [\widetilde{HotelReservation}],$
  $[\widetilde{City}]\ s_0 \times CH \rightarrow s_0\ [\widetilde{Hotel}],$
  $[\widetilde{Hotel}]\ s_0 \times HC \rightarrow s_0\ [\widetilde{City}],$
  $[\widetilde{Hotel}]\ s_0 \times HP \rightarrow s_0\ [\widetilde{Price}],$
  $[\widetilde{Sightseeing}]\ s_0 \times SC \rightarrow s_0\ [\widetilde{City}],$
  $[\widetilde{Sightseeing}]\ s_0 \times SCH \rightarrow s_0\ [\widetilde{City}; \widetilde{Hotel}],$
  $[\widetilde{City}]\ s_0 \times CS \rightarrow s_0\ [\widetilde{Sightseeing}],$
  $[\widetilde{Activity}]\ s_0 \times ABS \rightarrow s_0\ [\widetilde{Beach}],$
  $[\widetilde{Area} \wedge \widetilde{Dates}]\ s_0 \times AWS \rightarrow s_0\ [\widetilde{Weather}],$
  $[\widetilde{City} \wedge \widetilde{Dates}]\ s_0 \times CWS \rightarrow s_0\ [\widetilde{Weather}]\}$

This LTS4WS model is represented visually in Fig. 5, where $\mathcal{P}_X$ and $\mathcal{E}_X$ are the pre-condition and effect of Web service X, respectively.

In Fig. 3, we describe an LTS4WS model for a Web service repository with ten Web services. As defined in Definition 3, this is an LTS which has only one state, from and to which all of transitions corresponding to the Web services come and go. This simplicity of this LTS renders an important advantage when performing composition and verification as we do not need to generate the full schema in advance like the previous work [5].

A model checker will search over the state space generated from this LTS for a state meeting the requirement specified by hard and soft constraints. For example, the searching problem given in the motivating example can be represented as an LTL formula of $(\widetilde{Dates} \wedge \widetilde{Sightseeing} \rightarrow \widetilde{Price} \wedge \widetilde{HotelReservation}) \wedge (respTime \leq 30)$. Figure 4 shows a situation, where the goal is found without searching the whole space. Note that we can always prune the paths that violate soft constraints, but virtually, we cannot justify whether a path that will eventually satisfy a hard constraint or not. However, we can choose to explore the path that is most potential. If this path happens to reach the desired goal, other branches are not needed to be considered, as shown in Fig. 4. In the following section, we introduce the heuristic-based approach for doing so.

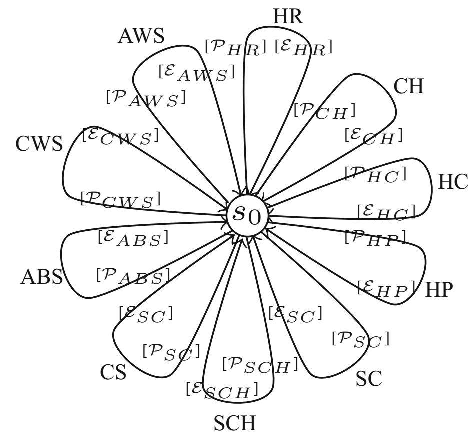


Fig. 3 Example of the LTS4WS model

Listing 1 describes the structure of LTS4WS model in the format of the XML language. Under this structure, the declaration of variables and constants is presented in *Declaration* section; *Process* contains the main contents of the model. In particular, *States* contains the states of the model, which has only one state s_0 as discussed; *Transitions* contains the transitions of the model. In this section, *Event* contains the names of transitions; *Guard* contains the expressions describing the guards of transitions; and Effect contains the expressions describing the effects of transitions.

Listing 1 *LTS4WS model is represented in XML-style*

```
  <Declaration>
  <!--variables, constants declaration-->
  </Declaration>
<ProcessName="System">
  <States>
    <State Name="s0" Init="True"/>
  </States>
  <Transitions>
    <Transition From="s0" To="s0"
      <Event>WebService_Name</Event>
      <Guard>Guarded_Expression</Guard>
      <Effect>Effect_Expression</Effect>
    </Transition>
    <!-- Other transitions -->
  </Transitions>
</Process>
</LTS4WS>
```

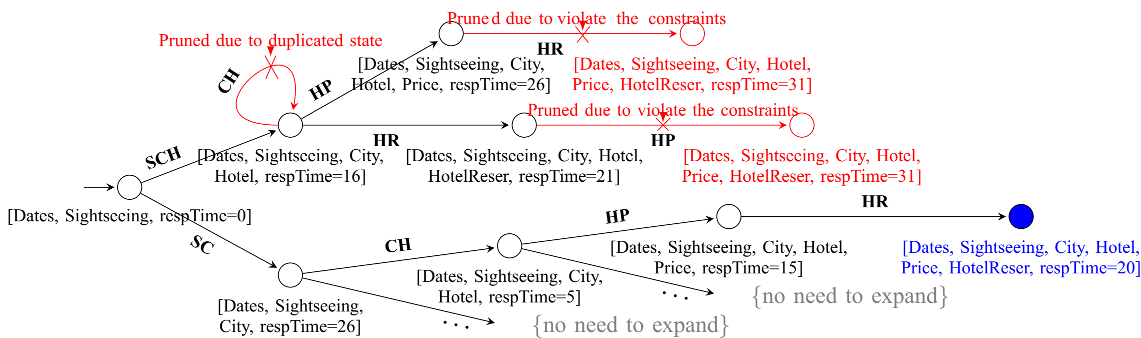


Fig. 4 State-space exploration

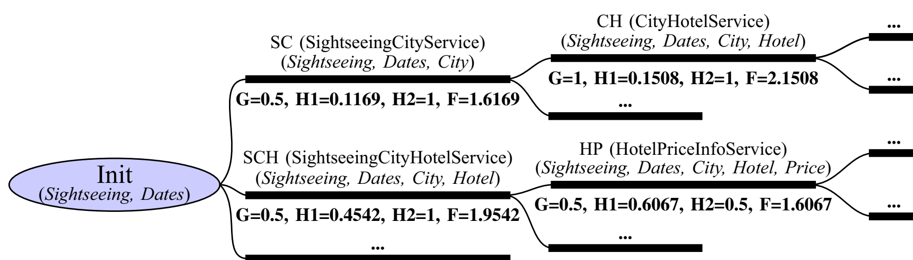


Fig. 5 Example illustrates the calculation of the value of G, $H1$, $H2$, and F

3 Heuristics-based approaches for web service composition

In our research, to perform the process of Web service composition and verification, we use the model checker PAT [11]. As well as other general model checkers, PAT has main functionality of verifying a temporal expression on a model. When the verification fails, a counter example is returned. It is not guaranteed to be an optimal path (the shortest path for instance). This is a general drawback of the model checkers. Another downside of model checkers is the state-space explosion problem. These two disadvantages originate from the causes that PAT (and other model checkers) typically adopts an exhaustive search algorithm, such as breadth first search or depth first search. Therefore, to apply PAT to our research and overcome its weaknesses, this study proposes using the heuristic search method (inspired from the well-known $A*$ algorithm) to improve the search process.

Heuristic function—F(n)

To apply heuristic search, we define the heuristic function $F(n)$ as the following formula:

$$F(n) = \alpha G(n) + \beta H(n), \quad (1)$$

where

- $G(n)$ is the real cost from state $init$ to state n,
- α, β are the corresponding weights of $H(n)$ and $G(n)$.
- $H(n)$ is the estimated cost (heuristic) from n to $goal$, as

$$H(n) = \beta_1 H1(n) + \beta_2 H2(n), \quad (2)$$

where

- $H1(n)$ is the estimated cost based on the QoS properties,
- $H2(n)$ is the estimated cost based on the functional properties,
- β_1, β_2 are the corresponding weights of $H1(n)$ and $H2(n)$.

The value of α, β, β_1, and β_2 can be empirically determined and adjusted by users. By default, all of those parameters have the same influence value of 1.

The real cost function—G(n)

$$G(n) = \frac{i}{l}, \quad (3)$$

where

- i is the number of composition step from $init$ to n.
- l is the number of functional properties that composite Web service must be satisfied.

We assume that each functional property needs one Web service. With l properties, the smaller i, the smaller $G(n)$ value.

Example 4 Supposed that we have a set of Web services as in Table 1 and the requirements as in Table 2. In this case, the

number of functional properties is $l = 2$. In the composition process, the values of G are computed as described in Fig. 5. For example, at a state immediately, after the $Init$ state, we have $i = 1$ and $G = 1/2$.

Estimated cost function based on QoS properties—H1(n)

The QoS properties are divided into two categories, positive group and negative group [16]. The higher value in negative property indicates the lower quality whilst the higher one in positive property reflects higher quality and vice versa. In addition, each QoS property has a different calculation unit, such as second (s) for response time, % for availability, etc. Therefore, we also normalize the value sets of the QoS properties before processing. With two different kinds of QoS properties, we then have the different normalized functions, respectively.

Let Q be a QoS property and q the value of Q. Then, the normalized value (q_{nrm}) is calculated as the following formula:

- If Q is a negative property, then

$$q_{\text{nrm}} = \begin{cases} \frac{q - q_{\min}}{q_{\max} - q_{\min}} & \text{if } q_{\max} - q_{\min} \neq 0 \\ 1 & \text{if } q_{\max} - q_{\min} = 0. \end{cases} \tag{4}$$

- If Q is a positive property, then

$$q_{\text{nrm}} = \begin{cases} \frac{q_{\max} - q}{q_{\max} - q_{\min}} & \text{if } q_{\max} - q_{\min} \neq 0 \\ 1 & \text{if , } q_{\max} - q_{\min} = 0. \end{cases} \tag{5}$$

Here $q_{\max}$ and $q_{\min}$ is the upper bound and lower bound of q, respectively.

After obtaining the normalized value for each property, we calculate the overall average value ($H1(n)$) for all QoS properties of Web services. The formula for calculating $H1(n)$ is as follows:

$$H1(n) = \sum w_i * q_{nrm_i} \tag{6}$$

where

- w_i is the weight of property ith, which can be changed by user. By default, all QoS properties have $w_i = 1$.
- q_{nrm_i} is a normalized value of property ith at state n.

Example 5 Concerning the previous example, we now consider a Web service with a QoS property of response time. The response time has value in range from 1 ($q_{\min} = 1$) to 60 s ($q_{\max} = 60$) and the weight (w_1) of 1. The values of QoS property of Web services are given in Table 1.

With the given QoS property, the value of $H1$ of each state is calculated as follows:

- The initial state (init): $H1 = 0$
- The state after invoking the SC service (*SightseeingCityService*) (state SC in Fig. 5):
  – The response time: $q_{\text{nrm}_1} = \frac{2-1}{60-1} = 0.0169$
  – $H1 = \sum w_i * q_{\text{nrm}_i} = 0.0169$

Similarly, we will calculate the value of $H1$ for every state, as shown in Fig. 5.

The estimated cost function based on functional properties—H2(n)

For each state in the state space, we have a set of variables so-called environment variables, describing the considered properties. For the functional properties, the corresponding variables will have values of 0 or 1, which implies whether the corresponding property has sufficient information or not.

$H2(n)$ function aims to look ahead, evaluate, and select the state which has the most functional properties satisfying the goal state. $H2(n)$ is calculated by the following formula:

$$H_2(n) = \frac{\text{diff}(\text{funcvar}(goal), \text{funcvar}(n))}{l} \tag{7}$$

where

- $\text{funcvar}(x)$ is a set of functional properties which we will have (whose value is 1) at the state x.
- $\text{diff}(A, B)$ is a function which counts the number of functional properties appear in A, but does not appear in B. Thus, $\text{diff}(\text{funcvar}(goal), \text{funcvar}(n))$ is the number of functional properties required by users that have not yet achieved at state n. When $\text{diff}(\text{funcvar}(goal), \text{funcvar}(n)) = 0$, in aspect of functionality, state n satisfies the user requirements.

Example 6 Supposed that we have a set of Web services as in Table 1 and the requirements as in Table 2. The set of functional properties of state $goal$ is $\text{funcvar}(goal) = \{Price, HotelReservation\}$, $l = 2$. The value of $H1$ of each state is calculated as follows:

- The initial state (init):
  – $\text{funcvar}(init) = \{Sightseeing, Dates\}$
  – $H2(init) = \frac{\text{diff}(\text{funcvar}(goal), \text{funcvar}(init))}{l} = \frac{2}{2} = 1$
- The state after invoking the HP service (*HotelPriceInfoService*) (state HP in Fig. 5):
  – $\text{funcvar}(HP) = \{Sightseeing, Dates, City, Hotel, Price\}$
  – $H2(HP) = \frac{\text{diff}(\text{funcvar}(goal), \text{funcvar}(HP))}{l} = \frac{1}{2} = 0.5.$

After calculating all values G, $H1$, $H2$, we will calculate the value of the heuristic cost function F by formula (1). Fig. 5 describes a general example of the values of

G, $H1$, $H2$, and F calculated in the process of composition, with the α, β, β_1, and β_2 are equal to 1 and the value of the QoS properties are given in Table 1. Based on these values, at each processing step, composition process will select the state, whose value of cost F is the lowest, to perform further processing.

4 Bitwise-based Web service indexing

In the previous sections, we have presented an approach of representing a repository of Web services as an LTS model, based on which a WSC request will be considered a state-space search problem. We have also discussed using heuristics to reach the search goal on the state space potentially faster. Hence, each composition step corresponds to a subgoal of the search problem.

Obviously, when looking for the candidates of the next composition step, one should only consider the Web services that are relevant to the current subgoal and discard the rest. However, it is not easy to quickly identify the relevant Web services at a current state. In this section, we address this problem by presenting a bitwise-based indexing technique to index the Web service repository. Based on the index information, one can easily locate the Web services relevant to some desired properties.

The idea of this technique is summarized as follows. Each Web service in the repository is represented as a pair of bitwise-based vectors, consisting an input vector and an output vector. In these vectors, each element is a bit representing a property, or feature. If the value of a bit is 0, the corresponding feature is not covered by the Web service and vice versa. The user requirements and subgoals are represented as a pair of bitwise-based vectors in similar manner.

4.1 Index table construction and manipulation

In the index table, each index item is a vector representing one feature. Web services are indexed using bitwise and ($\bar{\wedge}$) operator between its input vector and the vector of each index item. Thus, we can quickly select Web services relevant to the feature represented by the index item. The selection of index items relevant to the user requirements/subgoals is also done in the same way. As a result, we can find Web services relevant to user requirement/subgoals within almost constant complexity. The details of these steps are represented as follows.

4.1.1 The ordered feature set of Web service repository

Given a repository of n Web services with k distinct functional properties (k distinct features), an ordered feature set ($\mathcal{S}$) of the Web service repository is an ordered set with k elements for k sorted features.[2]

Example 7 The Web service repository given in Table 1 can be represented by an ordered set with 12 sorted features as follows. $\mathcal{S}$ ={*Activity*, *Adventure*, *Area*, *Beach*, *City*, *Dates*, *Hotel*, *HotelReservation*, *Price*, *RuralArea*, *Sightseeing*, and *Weather*}.

4.1.2 Bitwise-based vectors of a Web service and user functional requirements

Definition 4 *(Bitwise-based vector of feature set)* Let $\mathcal{S}$ be a set of properties, or *features*, and $\trianglelefteq$, an order relation on $\mathcal{S}$. We denote $\trianglelefteq(V, i)$ as the ith element of S when sorted by $\trianglelefteq$. The bitwise-based vector of a subset s of $\mathcal{S}$, denoted as $V_{\mathcal{S}}^{s}$, is given as follows:

$$V_{\mathcal{S}}^{s}[i] = \begin{cases} 1 & \text{if } \trianglelefteq (V, i) \in s; \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Example 8 For the ordered feature set $\mathcal{S}$ given in Example 7 and subset of features $s = \{Dates, Hotel\}$, we have the bitwise-based vector of s on $\mathcal{S}$ as follows:

$$V_{\mathcal{S}}^{s} = 000001100000. \tag{9}$$

For each Web service, we have two kinds of features (functional properties), which are inputs and outputs. Therefore, we construct the corresponding input and output bitwise-base vectors as follows.

Definition 5 (Bitwise-based vector of Web service) The bitwise-based vectors of a Web service is a pair $\langle V_i, V_o \rangle$, where V_i and V_o are the bitwise-based vectors of the input features and output features, respectively. V_i and V_o are defined as presented in Definition 4.

Example 9 The bitwise-based input and output vectors of all Web services in Table 1 are in Table 3.

Similarly, the user requirement is also represented as two bitwise-based vectors. They are so-called the bitwise-based supplied vector and the bitwise-based goal vector.

Example 10 The user requirement in the Table 2 can be represented as the bitwise-based vectors as in Table 4.

4.1.3 Bitwise-based indexing for Web service repository

After vectorizing Web services in the repository, we construct the indexing table based on the input vectors of Web services.

[2] We can apply any sorting criterion here. However, for convenience, the features are sorted by the alphabetical order unless indicated otherwise.

Table 3 Bitwise-based input and output vectors of Web services in Table 1

#	Web service	Bitwise-based input vector	Bitwise-based output vector
1	HotelReserveService (HR)	000001100000	000000010000
2	CityHotelService (CH)	000010000000	000000100000
3	HotelCityService (HC)	000000100000	000010000000
4	HotelPriceInfoService (HP)	000000100000	000000001000
5	SightseeingCityService (SC)	000000000010	000010000000
6	SightseeingCityHotelService (SCH)	000000000010	000010100000
7	CitySightseeingService (CS)	000010000000	000000000010
8	AdventureRuralAreaService (ARA)	010000000000	000000000100
9	ActivityBeachService (ABS)	100000000000	000100000000
10	AreaWeatherService (AWS)	001000000000	000000000001

Table 4 Bitwise-based vector of user requirement

User requirement	Bitwise-based supplied vector	Bitwise-based goal vector
Input: $Dates, Sightseeing$	000001000010	000000011000
Output: $Price, HotelReservation$		

The indexing table is a two-column table. The first column contains unit vectors, each of which represents a feature in the ordered feature set. To represent the ith feature, the ith bit of the corresponding unit vector is set as 1, whereas other bits of the vector are kept as 0. In the same row of the index table, the second column stores the Web services relevant to the feature represented by the unit vector at the first column.

The process of construction of the index table is presented in Algorithm 1.

Algorithm 1 Index table construction

Input: A Web service repository with n Web services
Output: The index table

Step 1. Build the ordered feature set $\mathcal{S}$

1: Extract features from Web services and build the ordered feature set $\mathcal{S}$

Step 2. Build the initial index table – T

2: **for** i from 0 to $\mathcal{S}.size() - 1$ **do**
3: Build the unit vector v_i of which bit i^{th} is 1
4: $T[i][0] \leftarrow v_i$

Step 3. Update the index table

5: **for** each Web service w in the repository **do**
6: Build the input vector $v_{w_{in}}$ and output vector $v_{w_{out}}$ of w
7: **for** i from 0 to $T.size() - 1$ **do**
8: $v_i \leftarrow T[i][0]$
9: **if** $v_i \barwedge v_{w_{in}} \neq 0$ **then** // $\barwedge$ *is the bitwise AND operator*
10: $T[i][1].add(w)$
11: return T

Example 11 The index table of the Web service repository in Table 1, built by Algorithm 1, is in Table 5.

Table 5 Indexing table of Web service repository

Index item	List of Web services
100000000000	⟨100000000000,000100000000⟩ (ABS)
010000000000	⟨010000000000,000000000100⟩ (ARA)
001000000000	⟨001000000000,000000000001⟩ (AWS)
000100000000	
000010000000	⟨000010000000,000000100000⟩ (CH), ⟨000010000000,000000000010⟩ (CS)
000001000000	⟨000001100000,000000010000⟩ (HR)
000000100000	⟨000001100000,000000010000⟩ (HR), ⟨000000100000,000010000000⟩ (HC), ⟨000000100000,000000001000⟩ (HP)
000000010000	
000000001000	
000000000100	
000000000010	⟨000000000010,000010000000⟩ (SC), ⟨000000000010,000010100000⟩ (SCH)
000000000001	

4.1.4 Choosing indexing items and choosing Web services

As discussed earlier in this text, at each composition, the Web services that are matched with indexing items corresponding with the current subgoal (i.e., they are relevant to the subgoal features) will be chosen as the candidates for the next step. The matching mechanism is implemented as follows.

Definition 6 *(Bitwise-based matching)* Let V be the bitwise-based input vector of the current subgoal, and V_k be the bitwise-based vector representing the kth index item. The kth item is a *matched index item* iff:

$$V \barwedge V_k = V_k \tag{10}$$

where $\barwedge$ is the bitwise AND operator.

Example 12 Taking the user requirement in Table 2 as the initial current subgoal, the inputs at the first composition step are $Sightseeing$, $Dates$ with the corresponding input vector 000001000010. The chosen index items are 000001000000 (corresponding to the value of sixth bit being 1) and 000000000010 (corresponding to the value of eleventh bit being 1). Therefore, the chosen Web services are *HR, SC, SCH*. In this case, the model checker will verify and choose SC and SCH to expand (Fig. 4). The Web service HR is rejected, as it requires the input of $Dates$ that we do not have at that point.

After selecting the Web services to put into the composition process at each composition step, the composition process will execute normally and the heuristic function will choose the best Web service to compose first, as previously discussed. The output of this web service is added the current input of the next subgoal. In that case, a Composite Bitwise-based Vector is generated as follows.

Definition 7 *(Composite bitwise-based vector)* Let V be the bitwise-based vector of the input of the current subgoal, WS_k be the chosen Web service to compose, $V_{k_{\text{out}}}$ be the bitwise-based vectors of the output of WS_k. The bitwise-based vector V_c of the resulted composite Web service is calculated as follows:

$$V_c = V \veebar V_{k_{out}} \tag{11}$$

where $\veebar$ is the bitwise OR operator.

Example 13 Assumed that the input of current subgoal is *Sightseeing, Dates*, corresponding to vector $V = 000001000010$. Supposed that the Web service *SightseeingCityService* is selected with $V_{k_{\text{in}}} = 000000000010$ and $V_{k_{\text{out}}} = 000010000000$. The bitwise-based vector of the composite Web service (V_c) is calculated as follows:

$$\begin{aligned} V_c = V \veebar V_{k_{\text{out}}} &= 000001000010 \veebar 000010000000 \\ &= 000011000010. \end{aligned} \tag{12}$$

4.1.5 Reducing the number of chosen Web services

In the approach of selecting Web services, as presented in Sect. 4.1.4, after each composition step, the number of obtained features, i.e., the 1-valued bits on the bitwise-based vector of the current composite Web service, obviously increases. Therefore, the number of selected index item and selected Web services considered as candidates for each step also increases. After several steps, the number of chosen index items will be enormous and almost all of Web services will be selected. This unfavorably increases the complexity of the composition process. Therefore, we need to consider reducing the number of selected Web services at each composition step reasonably.

To reduce the number of selected Web services, we consider the following remarks.

Eliminating the redundant Web services

The redundant Web service is the Web service of which the bitwise-based output vector is contained in bitwise-based vector of the current input. The contained relationship between two bitwise-based vectors is defined as in Definition 8.

Definition 8 *(The bitwise-based contained relationship)* Let V and $V_{k_{\text{out}}}$ are two bitwise-based vectors. V contains $V_{k_{\text{out}}}$ or $V_{k_{\text{out}}}$ is contained in V iff

$$V \barwedge V_{k_{out}} = V_{k_{\text{out}}} \tag{13}$$

where $\barwedge$ is the bitwise AND operator.

Example 14 Suppose that at a specific composition step, we have obtained information of the features of *Sightseeing, Dates, City*, corresponding to the bitwise-based vector of the current input $V = 000011000010$. The index items which satisfy vector V are 000010000000, 000001000000, and 000000000010, and the satisfied Web services which will be used in the composition process are HR, SC, SCH, CH, and CS (see details in Table 5). Let us consider the Web service CS (*CitySightseeingService*), which takes in the *City* and returns the *Sightseeing*, corresponding to the bitwise-based out vector is $V_{k_{\text{out}}} = 00000000010$. Since $V_{k_{\text{out}}}$ and V satisfy the bitwise-based contained relationship as follows:

$$\begin{aligned} V \barwedge V_{k_{\text{out}}} &= 000011000010 \barwedge 00000000010 \\ &= 000000000010 = V_{k_{\text{out}}} \end{aligned} \tag{14}$$

the Web service CS is a redundant Web service and will not be selected to put into the considered set.

4.1.6 The satisfied composite Web service

The composition process stops when the obtained features satisfy the initial user requirement. It means that the bitwise-

based vector of current input (V) contains the bitwise-based vector of user goal (V_g), or

$$V \barwedge V_g = V_g \quad (15)$$

4.2 Case study

In this case study, we consider the set of Web services listed in Table 1. Then, the corresponding bitwise-based indexing table is represented as in Table 5. For the user requirement presented in Table 2, the composition process is performed with the following steps.

Step 1

Current input features	Current input vector (V)	Chosen index	Chosen web services		
			WS	Bitwise-based input vector	Bitwise-based output vector
Sightseeing, Dates	000001000010	000001000000	HR	000001100000	000000010000
		000000000010	SC	000000000010	000010000000
			SCH	000000000010	000010100000

Supposed that in the composition process, the heuristic function evaluates that the Web service SC is the most suitable Web service to be invoked. After invoking Web service CS, the bitwise-based vector of current input $V = V \veebar V_{CS_{\text{out}}} = 000001000010 \veebar 000010000000 = 000011000010$. This vector does not contain the vector of user goal ($V \barwedge V_g \neq V_g$). Therefore, the composition process will be continued.

Step 2

At this step, we have three chosen index items with five Web services (*HR, SC, SCH, CH, CS*). However, the Web services SC and CS are eliminated, as they are redundant. The Web service selection process is represented as in the following table. The strike-through lines are the eliminated Web services.

Current input features	Current input vector (V)	Chosen index	Chosen web services		
			WS	Bitwise-based input vector	Bitwise-based output vector
Sightseeing, Dates, City	000011000010	000001000000	HR	000001100000	000000010000
		000000000010	SCH	000000000010	000010100000
		000010000000	CH	000010000000	000000100000

As in above table, we have three Web services used in composition (HR, SCH, and CH). The heuristic function chooses the Web service CH to compose in next step. After invoking this Web service, the bitwise-based vector of current input $V = V \veebar V_{CH_{out}} = 000011000010 \veebar 000000100000 = 000011100010$. This vector does not contain the vector of user goal ($V \barwedge V_g \neq V_g$). The composition process will be continued.

Step 3

At this step, we have four chosen index items with seven Web services (*HR, SC, SCH, CH, CS, HC, HP*), as in the table below.

Current input features	Current input vector (V)	Chosen index	Chosen web services		
			WS	Bitwise-based input vector	Bitwise-based output vector
Sightseeing, Dates, City, Hotel	000011100010	000001000000	HR	000001100000	000000010000
		000000000010			
		000010000000			
		000000100000			
			HP	000000100000	000000001000

According to above table, many Web services will be filtered out. In particular, SC, CH, SCH, CS, and HC are thrown away, because they are the redundant Web service; HR in the sixth row is removed, because it is duplicated. Therefore, we have only two remained Web services of HR and HP. The heuristic function will choose the Web service HP to compose, and the current input vector is updated

Table 6 Experiment data sets

Data set	No. of Web services	Description
Travel Booking (TB)	20	Including web services providing information to serve the travel booking
Medical Services (MS)	50	Services support to look up hospital, treatment, medicine, etc.
Education Services (EDS)	100	Services related to education, such as scholarship, courses, degrees, etc.
Economy Services (ECS)	200	Including services provided information on goods, restaurant, food, etc.
Global	1000	1000 random services from OWL-S-TC [17]

as $V = V \veebar V_{HP_{\text{out}}} = 000011100010 \veebar 000000001000 = 000011101010$. This vector does not contain the vector of user goal ($V \barwedge V_g \neq V_g$). The composition process will be continued in Step 4.

Step 4

At this step, with the input vector of $V = 000011101010$, we have one more index item, corresponding to the vector 000000001000. However, this index item does not contain any Web service. Therefore, there is only one service HR to compose, as depicted in the following table.

Current input features	Current input vector (V)	Chosen index	Chosen web services		
			WS	Bitwise-based input vector	Bitwise-based output vector
Sightseeing, Dates, City	000011100010	000001000000 000000000010 000010000000 000000100000	HR	000001100000	000000010000
		000000001000	*There is no web service*		

After invoking the Web service HR, the up-to-date bitwise-based vector V is 000011111010 (for obtained features of *Sightseeing, Dates, City, Hotel, Price* and *Hotel-Reservation*). This vector contains the vector of user goal ($V \barwedge V_g = 000011111010 \barwedge 000000011000 = 000000011000 = V_g$). Therefore, the composition process stops.

In summary, we have the composite Web service which consists of four Web services as $SC \bullet CH \bullet HP \bullet HR$.

5 Experimentations

In this section, we present the experimental results of our approach, which was built as the tool known as WSCOVER[3]. The tool WSCOVER is experimented on the real data sets obtained from the project OWL-S-TC [17]. OWL-S-TC provides over 1000 Web services classified into different domains, described by OWL-S [18]. In this data set, we select five sub-data sets, whose numbers of web services are varied 20 to 1000 services, as shown in Table 6.

We conduct experiment scenarios based on four different approaches, named full schema verification, on-the-fly composition and verification, heuristic-driven on-the-fly composition and verification, and combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification. The first one is the typical approach adopted by [5]. The last three approaches are our proposed ones, with and without using heuristics, and combined heuristic with bitwise-based indexing. The experiment is performed on a PC with core i5-5200 processor (4 x 2.7 GHz), 8.0 GB RAM, running on the 64-bit Windows 7 operating system. The experimental results are evaluated in three aspects: the number of expanded states, the number of visited states, and the execution time, and analyzed statistically, as depicted in Table 7 and Fig. 6.

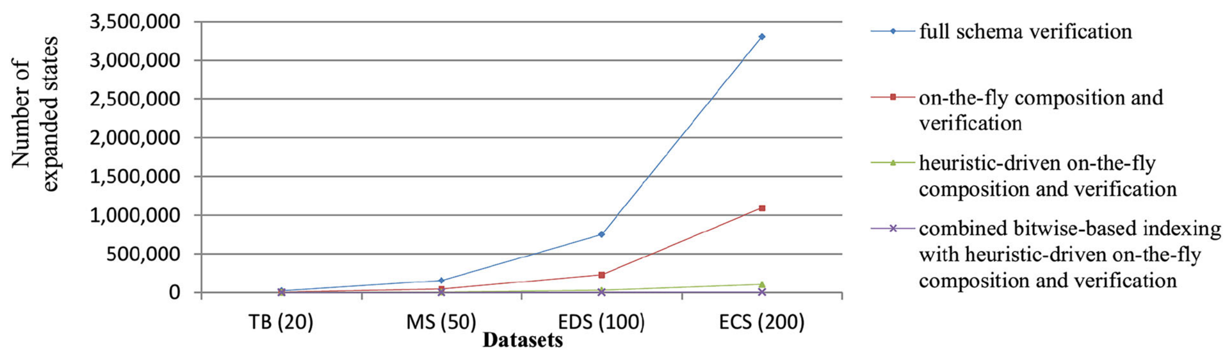
The experimental results confirm our hypothesis that the indexing helps in reducing the number of expanded states (see Fig. 6a) significantly.

Note that the heuristic algorithm in the WSCOVER tool always chooses the best way, i.e., the way that is evaluated most potential to reach to goal, to travel in the state space. Our heuristic approach showed it efficiency when there is almost no back-tracking step taken when the model checker explores the state space (i.e., the states suggested by the heuristic algorithm are, in fact, the states in the right path to the goal). Thus, the third and the fourth approaches, empowered by this heuristic strategy, enjoy a far less numbers of visited states, as compared with those in the first and the second approaches (Fig. 6b).

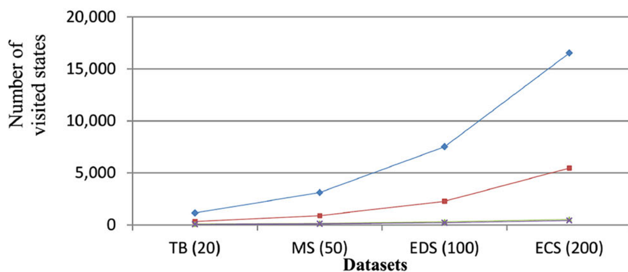
[3] This tool and its guidelines and also all experimental data sets can be downloaded from http://cse.hcmut.edu.vn/~save/project/wscover/start.

Table 7 Experimentation results

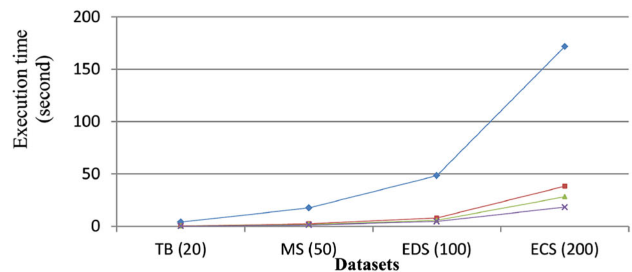
Data sets	Approaches	Expanded states	Visited states	Execution time (s)
TB (20)	Full schema verification	1100	56	0.250
	On-the-fly composition and verification	825	56	0.210
	Heuristic-driven on-the-fly composition and verification	316	35	0.155
	Combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification	316	35	0.155
MS (50)	Full schema verification	16,720	210	2.597
	On-the-fly composition and verification	12,331	210	2.022
	Heuristic-driven on-the-fly composition and verification	2143	119	1.538
	Combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification	316	35	0.155
EDS (100)	Full schema verification	27,400	275	5.570
	On-the-fly composition and verification	20,824	275	4.579
	Heuristic-driven on-the-fly composition and verification	3021	151	3.294
	Combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification	316	35	0.155
ECS (200)	Full schema verification	102,800	515	28.198
	On-the-fly composition and verification	76,586	515	23.030
	Heuristic-driven on-the-fly composition and verification	8324	287	18.273
	Combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification	316	35	0.155
Global (1000)	Full schema verification	Out-of-memory	Out-of-memory	Out-of-memory
	On-the-fly composition and verification	Out-of-memory	Out-of-memory	Out-of-memory
	Heuristic-driven on-the-fly composition and verification	91,457	1429	597.228
	Combined bitwise-based indexing with heuristic-driven on-the-fly composition and verification	316	35	0.155



(a) Visual comparison between approaches according to the number of expanded states



(b) Visual comparison according to the number of visited states



(c) Visual comparison according to execution time

Fig. 6 Visual representation of the experimental results

As expected result, where the execution time mostly depends on the number of expanded states and visited states, all of our proposed methods are faster than the original one in existing work and our new proposed bitwise-based indexing approach achieves the fastest execution time (Fig. 6c).

6 Related works

6.1 Web service composition and verification

Research community has a lot of work related to WSC problem, which can be classified into three groups as follows.

6.1.1 Composition based on hard constraints

WSC only involves the functional properties (*hard constraints*) which is the classic problem of SOA, which are mostly based on the theory of planning of the artificial intelligence field (AI Planning), such as [4] and [19]. Some recent studies are based on abstract models, such as Petri net or Colored Petri Net [12–14] to compose and verify Web services. PORSCE II [4] is a framework implementing the WSC-based on the requirements on input and output of the services. Similarly, OWL-S-XPlan [19] also uses Web services expressed by OWL-S to transform the problem from WSC domain to planning domain and uses the planner named XPlan, constructed by author. The studies [12–14] give the automatic WSC techniques based on Petri net (or Colored Petri net).

6.1.2 Composition based on hard and soft constraints

WSC method which combines functional properties (hard constraints) and QoS properties (soft constraints) has been proposed in [16]. In [16], the authors have proposed applying genetic algorithm (GA) to solve the problem with each possible composition encoded as a gene, to calculate value for specific kinds of QoS properties. However, this study only provides us a mechanism to choose the best (possible) composition from a set of composition ways (full composition schema) rather than composes from the component Web services. Besides, the application of genetic algorithms has increased the complexity of the problem and thus very difficult to apply in practice.

A different approach proposes the automated recovery when a WSC falls into the failure state (a Web service could not be accessed or unsatisfied the user requirement) [20]. With this approach, we have to have a full composition schema described in BPEL [21] language, which is transformed into a Labelled Transition System (LTS), monitored by a *monitor automata*. When an error arises (a state that cannot be reached, corresponding to a Web service cannot be accessed), the system will start calculating to choose the recovery plan using the genetic algorithm. The difference between [20] and [16] is that the size of gene in [20] is unfixed, which depends on the number of back-tracking steps from the error state.

6.1.3 Web service verification

As discussed, most of current researches of Web service verification only verify separately hard or soft constraints of the services. WS-Engineer [22] is typical work for the Web service verification based on the functional properties. A recent study carried out to verify combined functional and non-functional requirements of WSC is introduced as VeriWS [5], which takes in a full composition schema expressed in BPEL and uses the model checker to verify.

6.2 Web service indexing

The number of web services is increasing rapidly. That makes the Web service composition approaches become more complex and face many difficulties. Therefore, the Web service indexing was suggested to support Web service accessing more efficient. This issue has received the attention of many researchers.

Aiello et al. [8] index the Web service repository based on the services descriptions represented by the WSDL language. This approach uses the hash table to implement the index. The index structure consists of two tables, Partname Index and Service Index. Partname Index uses a hashtable to maintain the mapping from each partname into two lists of service names, namely, in list and out list. The Service Index utilizes a hashtable that maps a service name into detail information of the correspondent service. The information is request partnames, and response partnames. The study in [8] was implemented as the VitaLab system which performs the indexing of a large collection of WSDL service description following a semantic description of operations (given as a tree of “is-a” relations) and, given a request for a service, composes the available services to satisfy the request.

Zhou et al. [15] proposed the way by which inverted indexing can be used for fast discovery of Web services. The indexing mechanisms can be either inverted indexing or latent semantic indexing. Here, inverted index can be used as a measure to check OWL-S description contain the given term. Each keyword is connected to a list of document ids, in which keyword occurs.

Czyszczo and Aleksander in [9] proposed the solution for the problem of Web service retrieval by presenting a new approach to indexing of both SOAP and RESTful Web services. This approach uses index structure called parametric index that allows users to retrieve ranked results in accordance with specific parameters. The parameters refer to service’s integral components and are covered in presented formal definition of a Web service. Second, the services are

Table 8 Comparison on some of Web service composition and verification tools

Tool	Composition	Verification	Hard const.	Soft const.	Temporal	Input
OWL-S-XPlan	X		X			OWL-S
PORSCE II	X		X			OWL-S
WS-Engineer		X	X			BPEL
AgFlow		X		X		Statechart
VeriWS		X	X	X	X	BPEL
WSCOVER	X	X	X	X	X	OWL-S

modelled in vector space that allows the evaluation of their mutual relevance and enables obtaining ranked search results. To reduce the index size and to decrease the search time, the approach in [9] uses the method of conceptual indexing which groups relevant service components into concepts.

6.3 Evaluation and comparison of our approach with other studies

Table 8 presents the functional comparison our tool with others previously discussed. As observed, WSCOVER can perform both composition and verification at the same time and on-the-fly. Our tool composes and verifies a composite Web service on all kinds of constraints, hard and soft constraint, and also the temporal relations between component Web services.

To support model checking for composition and verification of Web services against a pre-defined goal in an effective manner, our work suggested representing a repository of Web services as a mathematical model, based on Labelled Transition System (LTS), known as LTS for Web Services (LTS4WS). Some other works, like the VeriWS tool [5], also propose using LTS models for enabling the formal verification of a WSC composition. However, our work distinguishes to this work as follows:

- In [5], the LTS model is rebuilt when we consider new composition goal, even though the repository remains the same. In contrast, our LTS4WS model can be applied for several various goals without being rebuilt.
- The work in [5] only verifies the composite Web service, it does not perform the composition phase. Meanwhile, our approach combines composition and verification at the same time.
- We apply the heuristic search to model checking engine based on the characteristics of Web service, so the verification performance is improved significantly.
- We also apply the bitwise-based indexing technique to index the Web service repository. This approach helps us to retrieve the Web services further faster and more reasonably. To the best of our knowledge, it is the first work that combines formal verification with indexing techniques in the area of Web services processing.

7 Conclusion and future work

The tool WSCOVER, developed in this research, has offered several useful functions that the other similar works do not support, including the capability of composing and verifying Web services, in an on-the-fly manner; and extending the model checker by applying the heuristics specifying the Web service nature. In addition, also through this paper, we have some contributions about the formal representation of Web service; the bitwise-based Web service indexing method to index the Web service repository using the bitwise operators. This helps the building of the index table as well as the retrieval of Web services more efficient.

As a result, WSCOVER can locate a composition solution faster and more optimally. However, in the future, this effort also needs to be compared with some other techniques, such as the clustering approaches to learn more about the pros and cons of our approach. In addition, in this work, we have not yet considered the semantic aspect of the features of Web services. We are going to develop a mechanism to calculate the similarity between two sets of features when they do not match completely.

Acknowledgements This research is funded by Vietnam National University HoChiMinh City under Grant Number C2015-20-10.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Yin, R.: Study of composing web service based on soa, In: Proceedings of the 2nd International Conference on Green Communications and Networks 2012 (GCN 2012), vol. 2, pp. 209–214. Springer (2013)

2. Rostami, N. H., Kheirkhah, E., Jalali, M.: Web services composition methods and techniques: a review. Int J Comput Sci Eng Inf Technol **3**(6), 15–29 (2013)
3. Fitting, M.: First-order logic. In: First-order logic and automated theorem proving, pp. 97–125. Springer, US (1990)
4. Hatzi, O., Vrakas, D., Bassiliades, N., Anagnostopoulos, D., Vlahavas, I.: The porsce ii framework: using ai planning for automated semantic web service composition. Knowl Eng Rev **28**(02), 137–156 (2013)
5. Chen, M., Tan, T. H., Sun, J., Liu, Y., Dong, J. S.: Veriws: a tool for verification of combined functional and non-functional requirements of web service composition. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 564–567 (2014)
6. Kumara, B.T., Paik, I., Chen, W., Ryu, K.H.: Web service clustering using a hybrid term-similarity measure with ontology learning. Int J Web Serv Res (IJWSR) **11**(2), 24–45 (2014)
7. Du, Y.Y., Zhang, Y.J., Zhang, X.L.: A semantic approach of service clustering and web service discovery. Inf Technol J **12**(5), 967–974 (2013)
8. Aiello, M., Platzer, C., Rosenberg, F., Tran, H., Vasko, M., Dustdar, S.: Web service indexing for efficient retrieval and composition. In: E-Commerce Technology, 2006. The 8th IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, The 3rd IEEE International Conference on. IEEE, pp. 63–63 (2006)
9. Czyszczoń, A., Zgrzywa, A.: Indexing method for effective web service retrieval. Int J Intell Inf Database Syst **8**(3), 189–208 (2014)
10. Huynh, K. T., Quan, T. T., Bui, T. H.: Fast and formalized: Heuristics-based on-the-fly web service composition and verification. In: Information and Computer Science (NICS), 2015 2nd National Foundation for Science and Technology Development Conference on. IEEE, pp. 174–179 (2015)
11. Liu, Y., Sun, J., Dong, J. S.: Developing model checkers using PAT. In: International symposium on automated technology for verification and Analysis, pp. 371–377. Springer, Berlin, Heidelberg (2010)
12. Fan, G., Yu, H., Chen, L., Liu, D.: Petri net based techniques for constructing reliable service composition. J Syst Softw **86**(4), 1089–1106 (2013)
13. Maung, Y.W.M., Hein, A.A.: Colored petri-nets (cpn) based model for web services composition. IJCCER **2**, 169–172 (2014)
14. Tian, B., Gu, Y.: Formal modeling and verification for web service composition. J Softw **8**(11), 2733–2737 (2013)
15. Zhou, B., Huang, T., Liu, J., Shen, M.: Using inverted indexing to semantic web service discovery search model. In: Wireless Communications, Networking and Mobile Computing, 2009. WiCom'09. 5th International Conference on. IEEE, pp. 1–4 (2009)
16. AllamehAmiri, M., Derhami, V., Ghasemzadeh, M.: Qos-based web service composition based on genetic algorithm. J AI Data Min **1**(2), 63–73 (2013)
17. Klusch, M.: Owls-tc: Owl-s service retrieval test collection, version 2.1. http://projects.semwebcentral.org/projects/owls-tc/
18. Burstein, M., Hobbs, J., Lassila, O., Mcdermott, D., Mcilraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., et al.: Owl-s: Semantic markup for web services. W3C Member Submission (2004)
19. Klusch, M., Gerber, A., Schmidt, M.: Semantic web service composition planning with owls-xplan. In: Proceedings of the AAAI Fall Symposium on Semantic Web and Agents. AAAI Press, USA (2005)
20. Tan, T. H., Chen, M., André, É., Sun, J., Liu,Y., et al.: Automated runtime recovery for qos-based service composition. In: Proceedings of the 23rd international conference on World wide web. International World Wide Web Conferences Steering Committee, pp. 563–574 (2014)
21. Jordan, D., Evdemon, J., Alves et al.: Web services business process execution language version 2.0. OASIS Stand **11**, 10 (2007)
22. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Wsengineer: A model-based approach to engineering web service compositions and choreography. In: Test and analysis of web services, pp. 87–103. Springer, Berlin, Heidelberg (2007)