Science in China Series F: Information Sciences

www.scichina.com
info.scichina.com
www.springerlink.com

AutoSyn: A new approach to automated synthesis of composite web services with correctness guarantee

HUAI JinPeng[1,2], DENG Ting[1,2†], LI XianXian[1,2], DU ZongXia[1,2] & GUO HuiPeng[1,2]

[1] National Laboratory of Software Development Environment, Beihang University, Beijing 100191, China;

[2] School of Computer Science & Engineering, Beihang University, Beijing 100191, China

How to compose existing web services automatically and to guarantee the correctness of the design (e.g. freeness of deadlock and unspecified reception, and temporal constraints) is an important and challenging problem in web services. Most existing approaches require a detailed specification of the desired behaviors of a composite service beforehand and then perform certain formal verification to guarantee the correctness of the design, which makes the composition process both complex and time-consuming. In this paper, we propose a novel approach, referred to as AutoSyn to compose web services, where the correctness is guaranteed in the synthesis process. For a given set of services, a composite service is automatically constructed based on L^* algorithm, which guarantees that the composite service is the most general way of coordinating services so that the correctness is ensured. We show the soundness and completeness of our solution and give a set of optimization techniques for reducing the time consumption. We have implemented a prototype system of AutoSyn and evaluated the effectiveness and efficiency of AutoSyn through an experimental study.

business protocol, synthesis, composite service, correctness constraints, L^* algorithm

1 Introduction

Service-oriented computing has become a promising paradigm for realizing distributed applications, as more and more web services are being developed and published on the Internet based on SOAP (http://www.w3.org/TR/2007/REC-soap12-part0-20070427/), WSDL (http://www.w3.org/TR/wsdl) and BPEL (www.ibm.com/develo-perworks/library/ws-bpel). These services can serve as the fundamental components for building complex applications. Recently, the web service composition issue has emerged as an important and challenging problem in web service applications, which is concerned with combining existing web services when a client request cannot be satisfied by any individual service[1].

Received April 10, 2009; accepted July 13, 2009
doi: 10.1007/s11432-009-0155-0

†Corresponding author (email: dengting@act.buaa.edu.cn)

Supported by the National High-Tech Research & Development Program of China (Grant No. 2007AA010301), the National Basic Research Program of China (Grant No. 2005CB321803), the National Natural Science Foundation of China for Distinguished Young Scholar (Grant No. 60525209), the National Natural Science Foundation of China (NSFC)/Research Grants Council (RGC) Joint Research Project (Grant No. 60731160632), and the Program for New Century Excellent Talents in University (Grant No. NCET-05-0186)

Citation: Huai J P, Deng T, Li X X, et al. AutoSyn: A new approach to automated synthesis of composite web services with correctness guarantee. Sci China Ser F-Inf Sci, 2009, 52(9): 1534–1549, doi: 10.1007/s11432-009-0155-0

There have been many efforts towards automated service composition, and most of them are based on formal methods including automata theory, logical reasoning, planning in AI and theorem proving[1]. Most of these approaches require developers to provide a detailed specification of the desired behaviors of a composite service (e.g., goal service in ref. [2] or conversation protocol in ref. [3]) with formal models or a specification language (e.g., BPEL4WS). To ensure the correctness of the design, developers of a composite service need to perform formal verification of the correctness constraints such as freeness of deadlock and unspecified reception[4], temporal properties[3,5], etc. The process of design, verification, analysis and correction makes the composition synthesis a difficult and time-consuming task.

In this paper we focus on a new approach to synthesizing the composite service from the published behavior descriptions of existing services where a composite service designer only needs to set the correctness constraints on the desired behaviors of the targeted service and the synthesis will be automatically performed with the correctness guaranteed.

Because of the loosely coupled and autonomous nature of web services, it is necessary that the composite service (or mediator, we use these two words interchangeably in this paper) can coordinate services to satisfy the correctness constraints no matter how the services behave. Therefore, we hope to synthesize a *most general* mediator, that is, the most general way of coordinating services to satisfy the correctness constraints, which is modeled as the most general plan or controller synthesis problem and pay an exponential cost in computation time[6], so an efficient synthesis algorithm is needed. Furthermore, most of existing approaches for controller synthesis and planning require the availability of the model of the whole state space (e.g., planning domain in ref. [7]), which induces the expensive consumption for time and space, so an on-the-fly synthesis approach is needed.

In light of these, in this paper, we give a novel automated composition synthesis framework AutoSyn. In terms of two kinds of important properties in software verification including safety properties and liveness properties, we divide the synthesis process in AutoSyn into two steps: we first synthesize a mediator such that the composite system satisfies the safety properties and is free of deadlock and unspecified reception, and then check whether it satisfies the liveness properties. Moreover, we prove the first step is PSPACE-hard and give an efficient synthesis algorithm based on the L^* algorithm in ref. [8]. Finally, we develop a prototype system and perform experimental study of AutoSyn. The experimenting results show that the AutoSyn is effective and efficient.

The organization of the paper is as follows: In section 2, we use Travel Agency as a motivating example to demonstrate the application scenario. In section 3, we introduce our synthesis model including formal models of services, composition requirement in service composition and composite service we want to build. We give an overview of our synthesis process and prove intractability results in section 4 and give detailed algorithms in section 5. We present the implementation of prototype system AutoSyn and experimental results in sections 6 and 7, respectively, discuss related work in section 8 and conclude the work in section 9.

2 Motivating example

In this section, we describe our running example that will be used throughout this paper.

Example 1. We want to develop a travel agency TA offering vacation packages to users by composing existing flight booking service FB and hotel booking service HB, where TA can coordinate FB and HB so that the user may directly request TA to book vacation package.

Figure 1 shows the business protocols of FB and HB. For the FB (the interaction process in business protocol of HB is similar to FB's and not detailed here), after receiving the request for booking flights for a given date, location and number of tickets (?fReq(loc, date, fNum)), it returns an offer with cost and other flight information if there are flights available (!fRe(fInfo, fcost)); otherwise, it tells the user no available flight (!noF). Then FB will receive the booking confirmation (?fAck) if the client accepts the offer or rejection, otherwise (?f-

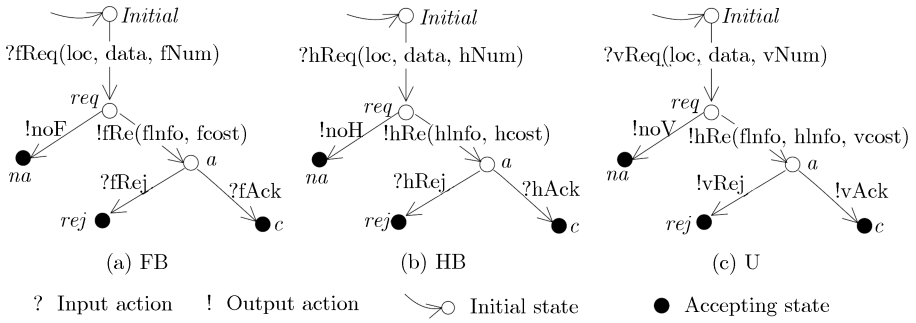


Figure 1 Business protocols of component services in travel agency.

Rej). We also consider the user of TA as a service U which specifies how users can interact with TA. As shown in Figure 1(c), after sending the request for vacation package (!vReq(loc,date,vNum)), U receives the offer from TA if flights and hotels are available (?vRe(fInfo,hInfo,vcost)). Then he will send the confirmation to book it (!vAck) if he likes the offer or rejects it (!vRej), otherwise.

In order to develop desired TA, we set correctness constraints on control dependencies and data dependencies. An example of correctness constrains on control dependencies is that the function of the TA is to "sell vacation package" which means the user must book both flight and hotel or book nothing when there is no available flight or hotel, or the user does not like the offer.

In addition, an example of correctness constrains on data dependencies is that the location, date and numbers of flight tickets and hotel rooms in request to FB and HB must be the same ones in the request TA received for vacation package from U, which brings about temporal constraints on messages exchanges: only after receiving messages vReq(loc, data, vNum) from U, can TA send messages fReq(loc, data, fNum) and hReq(loc, data, hNum) to FB and HB, respectively. We require that the whole composite system composed of TA, HB, FB and U should satisfy correctness constraints.

Based on the above correctness constraints, a typical interaction scenario of TA with successful booking is as follows. After receiving the request for vacation package from U, TA sends requests for flight ticket and hotel room to FB and HB, respectively. After receiving the information and costs of flight ticket and hotel room from FB and HB the TA sends these information to U. When U confirms the order, the TA sends the confirmation of order to the FB and HB.

3 System model and problem formulation

In this section, we introduce our composition synthesis model in AutoSyn, which mainly includes models for services, correctness constraints and the mediator we synthesize.

3.1 Services model

We specify a service in the level of business protocol or abstract business process in BPEL4WS. In Figure 1, we give the graph representations of the business protocols of services FB, HB and U. Intuitively, a service model specifies the message exchange sequences supported by a service, so a user can know how to invoke it. Here we model a service as a deterministic finite automata (DFA), where states represent the different phases when it interacts with a user and transitions are triggered by the messages it sends or receives.

Services. Formally, a service S is a tuple $S = (Q, \Sigma, q_0, F, \delta, Lab)$, where Q is the finite set of states of S; $q_0 \in Q$ is the initial state of S; $F \subseteq Q$ is the set of accepting states (or final states) of S; Σ is the set of input actions and output actions executed by S; δ is the partial state transitions function from $Q \times \Sigma$ to Q such that a state transition $\delta(q, a) = q'$ means that S reaches state q' after executing action a in state q; and Lab is the labeling function from Q to the power set

$\mathcal{P}(AP)$ that labels each state of S with the set of atomic propositions true in that state, where AP is the set of atomic proposition of services S and the atomic proposition is of the form "$x = v$" with variable x and its value $v \in D(v)$.

Notations. For a message m, the symbols $?m$ and $!m$ refer to input action for receiving message m and output action for sending message m, respectively. We represent the complementary action $\bar{a}$ of a as $\bar{a} = ?m$ if $a = !m$; otherwise $\bar{a} = !m$. Also we define the complementary set $\overline{\Sigma}$ of action set Σ as $\overline{\Sigma} = \{\bar{a} | a \in \Sigma\}$. Note that $\bar{a} \in \Sigma$ if $a \in \overline{\Sigma}$. Finally, we denote by $L(S)$ the languages accepted by service S.

In this paper, we do not distinguish the notion of services and users and treat users as services too. A mediator coordinates available services by routing the output of one service to the input of another, in order to deliver a requested service by invoking available services as component services. As shown in Example 1, the travel agency TA is a mediator which interacts with services FB, HB and U to purchase vacation package for users. So the mediator is characterized by the message exchange sequences with its component services. We formally define the interaction between mediator and its component services as the synchronization on input and output actions.

Interactions. Let $\boldsymbol{S} = \{S_1, S_2, \ldots, S_n\}$ be a given set of services, where $S_i = (Q_i, \Sigma_i, \delta_i, q_{i0}, F_i, Lab_i)(i \in [1, n])$ with atomic proposition set AP_i. For a mediator $M = (Q_M, \Sigma_M, q_{M0}, F_M, \delta_M, Lab_M)$ and $\Sigma_M \subseteq \overline{\Sigma_1 \cup \cdots \cup \Sigma_n}$, the interaction $M||\boldsymbol{S}$ between mediator M and services in $\boldsymbol{S}$ is a tuple $(Q, \Sigma, q_0, F, \delta, Lab)$, where

• $Q \subseteq Q_1 \times \cdots \times Q_n \times Q_M$; $\Sigma = \Sigma_M$; $q_0 = (q_{10}, \ldots, q_{n0}, q_{M0})$; $F \subseteq (F_1 \cup \{q_{10}\}) \times \cdots \times (F_n \cup \{q_{n0}\}) \times F_M$; and $Lab(q) = Lab_1(q_1) \cup \cdots \cup Lab_n(q_n)$, for $q = (q_1, \ldots, q_n, q_M)$.

• $\delta(q, a) = q'$ if $\delta_M(q_M, a) = q'_M$, $\delta(q_i, \bar{a}) = q'_i$, and $q_j = q'_j (j \neq i)$, for $q = (q_1, \ldots, q_n, q_m)$ and $q' = (q'_1, \ldots, q'_n, q'_m) \in Q$, $a \in \Sigma_M$ and $\bar{a} \in \Sigma_i$.

In the rest of this paper, we discuss the composition synthesis problem over the service set $\boldsymbol{S}$.

3.2 Correctness constraints

In this paper, we use the computation tree logic (CTL) to characterize correctness constraints on control dependencies and data dependencies. CTL logic is a branching-time temporal logic and CTL formulas are constructed from atomic propositions, logical operators $\wedge$, $\vee$, and $\neg$, path quantifiers **A** (for all computation paths) and **E** (for some computation path) and temporal operators **F**(eventually), **G**(globally), **X**(next time), and **U**(until). Given an interaction $M||\boldsymbol{S}$ and a CTL property φ, we denote the fact that $M||\boldsymbol{S}$ satisfies φ by $M||\boldsymbol{S} \models \varphi$. We assume that the readers are familiar with the CTL logic and do not detail it due to the paper space limitation. Readers can get detailed introduction of CTL logic in ref. [9].

Correctness constraints on control dependencies. The correctness constraints on control dependencies characterize the correct behavior of $M||\boldsymbol{S}$, where safety properties and liveness properties are two kinds of important properties. Informally, a safety property expresses that "some thing bad will not happen" during a system execution while a liveness property expresses that eventually "something good must happen".

According to the results in ref. [10], every CTL formula can be always represented as a conjunction of two CTL formulas representing universally safety property and existentially liveness property, respectively, when the semantic of CTL logic is defined in terms of a finitely branching trees. Here the universally safety properties are those that correspond to linear time safety properties over all computations while existentially live properties refer to those which require that there should exist at least one live computation. For example, the CTL formula **AG** P (meaning "along every computation all states satisfy P") is a universally safety property, while **EF** P (meaning "there is a computation along which some states satisfy P") is an existentially liveness property. In this paper, all services can be "unfolded" in a finitely branching tree, so we define the correct constraints on control dependencies φ_{control} as the following CTL formula:

$$\varphi_{\text{control}} = \varphi_{\text{safe}} \wedge \varphi_{\text{live}}.$$

Here $\varphi_{\text{safe}} = \mathbf{A}(\psi)$, where ψ refers to a linear time safety property, and $\varphi_{\text{live}} = \mathbf{E}(\psi')$, where ψ' refers to a linear time liveness property.

Example 2. For the travel agency TA in Example 1, let $q_{\mathrm{FB}}, q_{\mathrm{HB}}$ and q_{U} be the state variables of the FB, HB and U, respectively, which represent their different execution phases. We assume that the values req, a, na, c and rej of state variables mean that the order (for flight, hotel or vacation package) is requested, available, unavailable, confirmed or rejected, respectively. Let proposition p_r^v refer to "$q_r = v$", where q_r is state variable of service r (r may be FB, HB and U) and v may be req, a, na, c and rej. For example, the proposition p_U^{req} refers to "$q_{\mathrm{U}} = req$", which means that the U has sent request message vReq(loc,date,vNum) to TA. Then the requirements to "sell vacation package" in Example 1 can be represented as CTL formula φ_1:

$$\begin{aligned}\varphi_1 = \mathbf{AG}\neg((&p_{\mathrm{FB}}^c \wedge (p_{\mathrm{HB}}^{rej} \vee p_{\mathrm{HB}}^{na} \vee p_{\mathrm{U}}^{rej} \vee p_{\mathrm{U}}^{na}))\\ &\vee (p_{\mathrm{HB}}^c \wedge (p_{\mathrm{FB}}^{rej} \vee p_{\mathrm{FB}}^{na} \vee p_{\mathrm{U}}^{rej} \vee p_{\mathrm{U}}^{na}))\\ &\vee (p_U^c \wedge (p_{\mathrm{FB}}^{rej} \vee p_{\mathrm{FB}}^{na} \vee p_{\mathrm{HB}}^{rej} \vee p_{\mathrm{HB}}^{na}))).\end{aligned}$$

Moreover, we hope the TA is "active", which means that only when knowing that flight ticket or hotel room is not available, can TA tell U no available vacation package; in other words, the TA must provide users the vacation as much as possible. This can be represented as φ_2:

$$\varphi_2 = \mathbf{AG}\neg((\neg p_{\mathrm{FB}}^{na} \wedge \neg p_{\mathrm{HB}}^{na} \wedge p_{\mathrm{U}}^{na})).$$

Furthermore, it is desired that when knowing one service is unavailable (e.g. flight ticket is unavailable), the U would not query the other service (e.g. HB). This can be represented as φ_3:

$$\varphi_3 = \mathbf{AG}\neg((p_{\mathrm{FB}}^{na} \rightarrow \mathbf{X}p_{\mathrm{HB}}^{req}) \vee (p_{\mathrm{HB}}^{na} \rightarrow \mathbf{X}p_{\mathrm{FB}}^{req})).$$

Lastly, the TA must be "meaningful", i.e. there exists at least an interaction scenario in which users can make a successful booking; otherwise, it is meaningless. This can be represented as CTL formula φ_4:

$$\varphi_4 = \mathbf{EF}(p_{\mathrm{FB}}^c \wedge p_{\mathrm{HB}}^c \wedge p_{\mathrm{U}}^c).$$

Then the correct constraints of TA on control dependencies is $\varphi_{\mathrm{control}} = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$, where $\varphi_{\mathrm{safe}} = \varphi_1 \wedge \varphi_2 \wedge \varphi_3$ is a safety property and $\varphi_{\mathrm{live}} = \varphi_4$ is a liveness property.

Correctness constraints on data dependencies. The correctness constraints on dataflow refer to the data dependency between exchanged messages. We give the notion of message mappings to represent a kind of simple constraints on data dependencies. A message mapping over services set $\boldsymbol{S}$ is a mapping of the form $h(m_1, m_2 \ldots, m_l){=}m'$, where every m_i is an output message of one service in $\boldsymbol{S}$ and m' is an input message of one service in $\boldsymbol{S}$. The message mapping h indicates that only after receiving messages $m_1, m_2, \ldots, m_l$ from services in $\boldsymbol{S}$ can the mediator construct the message m' and send it. We denote by $MM(\boldsymbol{S})$ the set of all message mappings over $\boldsymbol{S}$.

Example 3. The data dependencies in Example 1 can be represented as the message mappings h_1 and h_2:

$$h_1(\mathrm{vReq(loc, date, vNum)}){=}\mathrm{fReq(loc, date, fNum)},$$
$$h_2(\mathrm{vReq(loc, date, vNum)}){=}\mathrm{hReq(loc, date, hNum)}.$$

Additionally, the flight and hotel information in return message from the TA to the U must be consistent with ones in return messages from FB and HB to TA, and the cost of vacation package may be the sum of the cost of flight tickets and hotel rooms plus some additional handling charge. So only after receiving fRe(fInfo, fcost) and hRe(hInfo, hcost) from FB and HB, can TA construct the message vRe(fInfo, hInfo vcost) and send it to U, which can be represented as h_3:

$$\begin{aligned}&h_3(\mathrm{fRe(fInfo, fcost)}, \mathrm{hRe(hInfo, hcost)})\\ &= \mathrm{vRe(fInfo, hInfo, vcost)}.\end{aligned}$$

As shown in Example 3, temporal constraints are implied in the message mappings. Accordingly we translate message mappings in $MM(\boldsymbol{S})$ into equivalent CTL logic formulas. Firstly, we define a kind of special atomic proposition as follows:

For an output action $!m$ of one service in $\boldsymbol{S}$, we say that $M||\boldsymbol{S}$ satisfies the atomic proposition m in state q, denoted by $M||\boldsymbol{S}, q \models m$, if and only if the mediator M has received message m in state q. For the input action $?m$, we have an analogous definition.

So a message mapping $h(m_1, m_2, \ldots, m_l) = m'$ can be denoted by the CTL formula φ_h equivalently:

$$\varphi_h = \mathbf{AG}(m' \rightarrow m_1 \wedge \cdots \wedge m_l),$$

which means that before sending the message m', the mediator M must have received $m_1, m_2, \ldots, m_l$.

Assume that there exist more than one construction approach for message m', (e.g. there are two hotel booking services HB1 and HB2, so there are two message mappings for input message vRe(fInfo, hInfo, vcost) of U); that is, there are message mappings $h_1, h_2, \ldots, h_t (t > 1)$ such that m' occurs in the RHS of every h_i. We define $\varphi(m')$ as

$$\varphi(m') = \varphi_{h_1} \vee \varphi_{h_2} \vee \ldots \vee \varphi_{h_t}.$$

Here $\varphi(m')$ means that once one message mapping h_i is satisfied; that is, the mediator has received all the messages required by h_i, the mediator can construct the message m' from h_i and omit other message mappings. We denote by $\Sigma^{in}_{MM(\boldsymbol{S})}$ the set of all input messages that occur in the RHS of one message mapping in $MM(\boldsymbol{S})$, and then the set $MM(\boldsymbol{S})$ is equivalent to φ_{data}, where

$$\varphi_{\text{data}} = \bigwedge_{m' \in \Sigma^{in}_{MM(\boldsymbol{S})}} \varphi(m').$$

Obviously, φ_{data} is safety property because every φ_{h_i} is safety property and safety properties are closed over conjunction and disjunction.

Example 4. The equivalent CTL formula corresponding to the message mapping in Example 3 is (we omit the parameters in every message)

$$\varphi_{\text{data}} = \mathbf{AG}((\text{fReq} \rightarrow \text{vReq}) \wedge (\text{hReq} \rightarrow \text{vReq}) \wedge (\text{vRe} \rightarrow \text{fRe} \wedge \text{hRe})).$$

3.3 Problem statement

For services U, FB and HB in Example 1, we want to construct a travel agency TA such that the interaction between TA and U, FB, HB satisfies the correctness constraints given in Examples 2 and 4. Moreover, we hope that there exist no deadlocks nor unspecified receptions in the interaction. Specifically, the property free of unspecified reception is an important condition, since it is possible to bring about disagreement in that the state of the whole composite system is between mediator and its component services if one message sent cannot be received. For example, when the mediator cannot receive the acknowledge message for booking from U in the current state, then U thinks of the booking as successful while the mediator does not think so. Moreover the asynchronous nature of web services can be embodied in our synthesis model to a certain extent because it supports any order of messages arrival as long as the correctness constraints are not violated.

We characterize our composition synthesis problem formally: given a set $\boldsymbol{S}$ of component services and correctness constraints $\varphi_{\text{control}} \wedge \varphi_{\text{data}}$ in CTL logic formula based on control dependencies and data dependencies, we aim to construct a mediator M such that

- $M||\boldsymbol{S} \models \varphi_{\text{control}} \wedge \varphi_{\text{data}}$;
- $M||\boldsymbol{S}$ is free of deadlock, i.e., $M||\boldsymbol{S}$ does not terminate in its non-accepting states;
- $M||\boldsymbol{S}$ is free of unspecified receptions, i.e., in every state, M and services in $\boldsymbol{S}$ cannot be offered a message while the corresponding state transition is unspecified.

We denote the mediator satisfying the above three conditions by $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$.

4 Synthesis approach overview

In this section, we investigate solutions to the composition synthesis problem defined in the last section. We begin by outlining our approach by the following example, and then show that the composition synthesis approach is intractable.

Example 5. We want to construct a mediator TA for the given services set {U, FB, HB} in Example 1 and correctness constraints in Examples 2 and 4. Intuitively, the whole synthesis process is separated into two steps. Firstly, we consider only property $\varphi_1, \varphi_2, \varphi_3$ (in Example 2), and φ_{data} (in Example 4); that is, we synthesize the TA such that the interaction between TA and services set {U, FB, HB} is free of deadlock and unspecified receptions and satisfies the the safety property $\varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_{\text{data}}$. Then we check whether the interaction satisfies liveness property φ_{live}, and if it does, the TA is the final solution; otherwise, the synthesis fails.

Given a services set $\boldsymbol{S}$ and correctness constrains $\varphi_{\text{control}} \wedge \varphi_{\text{data}}$, where $\varphi_{\text{control}} = \varphi_{\text{safe}} \wedge \varphi_{\text{live}}$, our

aim is to find the mediator $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$. We divide the synthesis process into two steps. Firstly, we only consider synthesizing mediator $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ which is a prefix-closed regular language because $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$ is a safety property. We show that the synthesis of $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ is PSPACE-hard by reduction from Q3SAT problem which is known as PSPACE-complete[11].

Theorem 1. The synthesis of $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ is PSPACE-hard.

In order to deal with the intractability, we use learning approach, specifically the L^* algorithm in ref. [8], to synthesize a safe mediator M_{safe} so that $M_{\text{safe}}||\boldsymbol{S} \models \varphi_{\text{safe}} \wedge \varphi_{\text{data}}$; moreover, M_{safe} is most general such that $L(M') \subseteq L(M_{\text{safe}})$ for every safe mediator M' (i.e. $M'||\boldsymbol{S} \models \varphi_{\text{safe}} \wedge \varphi_{\text{data}}$). The L^* algorithm is a learning algorithm for unknown regular language U which guarantees to generate a minimal DFA to accept U by asking two kinds of queries to a teacher, where the number of queries is polynomial in the size of generated DFA.

However, we observe that $M_{\text{safe}}||\boldsymbol{S}$ may have a deadlock. A state is a deadlock state of $M_{\text{safe}}||\boldsymbol{S}$ if it is a non-accepting state and $M_{\text{safe}}||\boldsymbol{S}$ cannot leave it once entering it. We analyze the existence of deadlock states in $M_{\text{safe}}||\boldsymbol{S}$ and give an approach to eliminating them successfully when it is possible, and then we get a most general $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$. As a result, when $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ satisfies the liveness property φ_{live} we get the mediator $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$. Figure 2 illustrates the whole synthesis process for $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$. We will detail the synthesis approach of most general $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ in the next section.

5 Learning-based composition synthesis approach

In this section, we present the detailed introduction for synthesizing the mediator $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ which includes synthesizing M_{safe} by using the L^* algorithm and analyze the deadlock in $M_{\text{safe}}||\boldsymbol{S}$.

5.1 L^* algorithm

In this subsection we introduce the L^* algorithm in ref. [8]. Figure 3 illustrates an overview of L^* algorithm. The L^* algorithm is a learning algorithm which aims to learn an unknown regular language U on alphabet Γ and generates a minimal DFA C to accept the regular language U. In the learning process, the L^* algorithm asks a teacher two kinds of questions: membership queries and equivalence queries, and infers the C based on the answers of the teacher. Here the membership query asks whether a string γ on Γ is in U, and the equivalence query for a candidate DFA C asks whether C is equivalent to U. The teacher answers true or false for queries, and gives a counterexample when the answer to equivalence query is false.

More specifically, the L^* algorithm maintains an observation table (ST, EX, T), where ST and EX are sets of strings over alphabet Γ, and T is a function from $(ST \cup ST \cdot \Gamma) \cdot EX$ to {true, false}. For γ in $(ST \cup ST \cdot \Gamma) \cdot EX$, $T(\gamma)$ =true if and only if the membership query on γ is true (i.e. $\gamma \in U$). Specifically, $T(\varepsilon)$ =true, where ε is an specific action which does nothing.

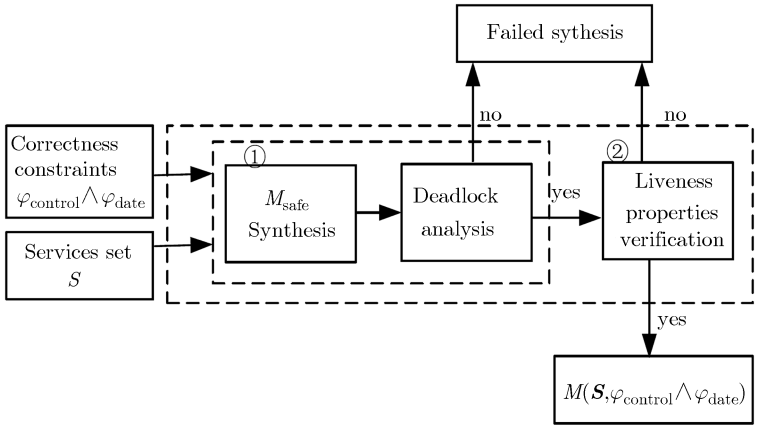


Figure 2 The synthesis process in AutoSyn.

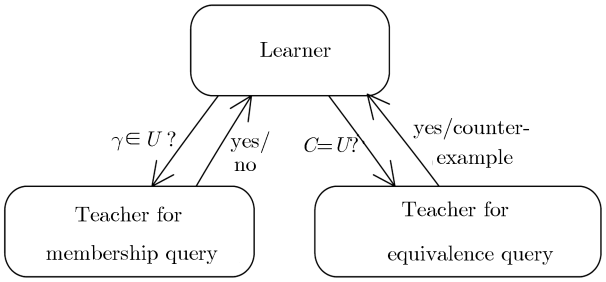


Figure 3 Overview of L^* algorithm.

Algorithm 1: L^* algorithm

```
1. Initialize the (ST, EX, T);
2. repeat: {
3.    while (ST, EX, T) is not closed due of
                    s · a
        /* s ∈ ST, a ∈ Γ */
4.    {
5.       Add s · a to ST
6.       Update T based on s · a;
7.    }
8.    Construct candidate DFA C from (ST,
      EX, T);
9.    if equivalence query on C is true
10.      then return C;
11.   else get an experiment exp from the
                counterexample of equivalence query;
12.   Add exp to EX
13.   Update the T based on exp;
14. }
```

First, the algorithm initializes the observation table (ST, EX, T) with $ST = EX = \{\varepsilon\}$ (line 1). Then it checks whether the observation table is closed (line 3); that is, for each s in ST and a in Γ, there exists an s' in ST such that $T(s \cdot a \cdot e) = T(s' \cdot e)$ for each e in EX. If not, the $s \cdot a$ answering for the un-closeness of observation table is added into ST (line 5) and T is updated based on $s \cdot a$; that is, for every $e \in EX$, we get $T(s \cdot a \cdot e)$ through membership query (line 6); otherwise, the algorithm constructs a candidate DFA $C = (Q_c, \Sigma_c, q_c, F_c, \delta_c, Lab_c)$ (line 8) as follows:

$Q_c = ST$, $q_c = \varepsilon$, $\Sigma_c \subseteq \Gamma$, $F_c = \{s \in ST | T(s) = true\}$, $Lab_c = \emptyset$, and for each $s \in ST$ and $a \in \Gamma$, $\delta_c(s, a) = q'$ such that $T(s \cdot a \cdot e) = T(s' \cdot e)$ for every $e \in EX$.

Next, the algorithm asks the equivalence query for the candidate DFA C. If the answer is true, the algorithm terminates and outputs the C (lines 9, 10); otherwise, the algorithm gets an experiment exp from the counterexample $cex \in ((L(C) \setminus U) \cup (U \setminus L(C)))$ (line 11), adds exp into EX (line 12) and updates the T based on exp (line 13). Note that ref. [8] gave an approach for extracting exp from cex. We will not detail it due to space limitation.

Characteristics of L^* algorithm. As pointed in ref. [8], the L^* algorithm guarantees to terminate and construct a minimal DFA for the regular language U. More importantly, it needs only the polynomial number of membership and equivalence queries: $O(kl^2 + l \log m)$ membership queries and at most $l - 1$ equivalence queries, where k equals $|\Gamma|$, l is the number of states of DFA returned by L^* algorithm and m is the length of the longest counterexample provided by the teacher for equivalence queries.

5.2 Synthesis through L^* algorithm

In this subsection, we show how we can use L^* algorithm to learn the most general safe mediator M_{safe} for a given set $\boldsymbol{S}$ of component services and safety properties $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. We denote the language accepted by M_{safe} by L_{safe} which is a regular language on $\overline{\Sigma}$, where $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_n$(see the definition of service set $\boldsymbol{S}$ in subsection 3.1).

Because the L^* algorithm always assumes that there exists a teacher answering queries, we have to give an approach to realizing the teacher in our synthesis process. As with ref. [12], we divide the equivalence query (i.e. whether $L(C)$ is exact the L_{safe}) into two steps: we perform subset query (i.e. whether $L(C)$ is a subset of L_{safe}) first and perform the superset query (i.e. whether $L(C)$ is a superset of L_{safe}) next if the answer of subset query is true.

5.2.1 Membership and subset queries. As shown in ref. [12], for a string $\sigma = a_1 \ldots a_n \in \overline{\Sigma}$, the membership query can be reformulated as a subset query on C_σ, where

$C_\sigma = (Q_\sigma, \Sigma_\sigma, q_{\sigma 0}, \delta_\sigma, Lab_\sigma)$ such that $Q_\sigma = \{q_{\sigma 0}, q_1, \ldots, q_n\}$, $\Sigma_\sigma = \{a_1, \ldots, a_n\}$, $F_\sigma = Q_\sigma$, $Lab_\sigma(q) = \emptyset$ for each $q \in Q_\sigma$ and $\delta_\sigma(q_{\sigma 0}, a_1) = q_1$, $\delta_\sigma(q_i, a_{i+1}) = q_{i+1}, i \in [1, n-1]$.

Obviously, C_σ accepts σ and its prefixes exactly. So we only discuss how to answer subset queries here.

Given a candidate DFA $C = (Q_c, \Sigma_c, q_{co}, \delta_c, F_c, Lab_c)$, the subset query asks whether $L(C)$ is a subset of the language L_{safe}, which means that all the interaction between services in $\boldsymbol{S}$ and C (i.e. $C||\boldsymbol{S}$) must satisfy correctness constraints $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. It is desirable that the candidate mediator C has no "useless" state transitions in the sense that services in $\boldsymbol{S}$ can interact with all input and output actions of C. So we introduce a notion of complete interaction which is similar to the notion of strong simulation in ref. [13].

Complete interaction. We say that services set $\boldsymbol{S}$ can interact with C completely if there exists a mapping g from the state set Q_c of C to the power set $\mathcal{P}(Q_1 \times \cdots \times Q_n)$ such that 1) $(q_{10}, \ldots, q_{n0}) \in g(q_{c0})$, and 2) if $\delta_c(q_c, a) = q'_c$, assume that $\bar{a} \in \Sigma_i$, then for every $(q_1 \ldots, q_n) \in g(q_c)$, there exists $q'_i \in Q_i$ such that $\delta_i(q_i, \bar{a}) = q'_i$ and $(q'_1, \ldots, q'_n) \in g(q'_c)$, where $q'_j = q_j\ (j \neq i)$. We call the interaction $C||\boldsymbol{S}$ a complete interaction if services set $\boldsymbol{S}$ can interact with C completely.

Following the above definition, it is easy to construct the mapping g and complete interaction $C||\boldsymbol{S}$ through breadth-first search on C from its initial state. Then the answering of subset queries is divided into two steps: first we check whether services set $\boldsymbol{S}$ can interact with C completely, and if it does, we get the complete interaction $C||\boldsymbol{S}$ and invoke the model checker NuSMV to check whether $C||\boldsymbol{S}$ satisfies $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. So the subset query will return a counterexample if (a) services set $\boldsymbol{S}$ cannot interact with C completely, or (b) the model checker NuSMV returns a counterexample when $C||\boldsymbol{S}$ does not satisfy $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$.

Counterexample of subset query. In the above case (b), a counterexample of subset query is provided by model checker NuSMV directly. For case (a), we record the construction process of $C||\boldsymbol{S}$ as a tree, where the root node r is labeled as $(q_{10}, \ldots, q_{n0}, q_{c0})$ and $(q_{10}, \ldots, q_{n0}) \in g(q_{c0})$. An edge labeled action $a \in \overline{\Sigma_i}$ from a node u labeled $(q_1, \ldots, q_n,\ q_c)$ such that $(q_1, \ldots, q_n) \in\ g(q_c)$ to a node v labeled $(q'_1, \ldots q'_n, q'_c)$ such thta $(q'_1, \ldots q'_n) \in g(q'_c)$ means that service S_i in $\boldsymbol{S}$ can interact with C on action a, i.e., $\delta_c(q_c, a) = q'_c$, $\delta_i(q_i, \bar{a}) = q'_i$, and $q_j = q'_j (j \neq i)$. If at node u, service S_i cannot synchronize the action a of C, we get a counterexample $\sigma \cdot a \in L(C) \setminus L_{\text{safe}}$, where σ is the path composed of edges from root node r to node u.

5.2.2 Superset queries. Given a candidate automaton C on which the subset query is true, the superset query asks whether $L(C)$ is a superset of L_{safe}. Then the teacher must check that for each string $\sigma \in \overline{\Sigma}^*$, $\sigma \notin L(C) \rightarrow \sigma \notin L_{\text{safe}}$.

For every string $\sigma \notin L(C)$, let $p(\sigma)$ be the minimal length prefix of σ such that $p(\sigma) \notin L(C)$. It is obvious that $\sigma \notin L_{\text{safe}}$ if $p(\sigma) \notin L_{\text{safe}}$ because L_{safe} is a prefix-closed language.

We denote the set of all this kind of minimal length prefixes by P, i.e., $P = \{p(\sigma) | \sigma \in \overline{\Sigma}^*, \sigma \notin L(C)\}$. Then the teacher needs only to check that for each string $\sigma \in \overline{\Sigma}^*$, $\sigma \in P \rightarrow \sigma \notin L_{\text{safe}}$. We get a counterexample $cex = \sigma' \in P$ of superset query on C if $\sigma' \in L_{\text{safe}}$.

In order to get the set P and to answer the superset query, we give another kind of interaction in which services in $\boldsymbol{S}$ interact with action sequence of C followed by at most one action not allowed by C; in other words, services in $\boldsymbol{S}$ interact with C in the same way as in complete interaction until one service in $\boldsymbol{S}$ executes a action $\overline{a}$ while a is not supported by C first time, and then the interaction terminates. We use the model checker NuSMV to check whether the first action not allowed by C always makes the composite system violate safety property $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. If it does, the answer for superset query is true; otherwise a counterexample is generated.

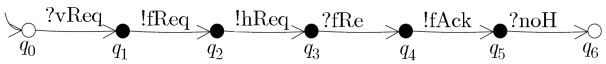
Counterexample of superset query. We also record the above interaction process as a tree in the same way in subset query except one edge corresponding to action a cannot be supported by C: for the node u labeled with $(q_1, q_2, \ldots, q_n, q_c)$, assume that $\overline{a} \in \Sigma_i$, if service S_i can execute action $\overline{a}$ in state q_i, and C cannot execute action a in state q_c, there is an edge from u to node v labeled with $(q'_1, q'_2, \ldots, q'_n, q_c)$ where $q'_i = \delta(q_i, \overline{a})$ and $q_j = q'_j (j \neq i)$. We denote this tree by $T(q_c, a,$

S_i). If the tree $T(q_c, a, S_i)$ satisfies the specification $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$, the edges sequence of the tree from the root node to a is a counterexample of superset query.

5.3 Deadlock analysis

The L^* algorithm always generates the most general safe mediator M_{safe} such that the interaction $M_{\text{safe}}||\boldsymbol{S}$ satisfies the safety property $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. However, it is possible that there is a deadlock in $M_{\text{safe}}||\boldsymbol{S}$. A state of $M_{\text{safe}}||\boldsymbol{S}$ is a deadlock state if it is a non-accepting state and when reaching it $M_{\text{safe}}||\boldsymbol{S}$ cannot leave it. Recall that $q = (q_1, q_2, \ldots, q_n, q_M)$ is a non-accepting state of $M_{\text{safe}}||\boldsymbol{S}$ if (a) q_M is a non-accepting state of M_{safe}, or (b) there exists q_i such that q_i is neither the initial state nor the accepting state of service S_i. We use the following example to illustrate a situation in which there exist deadlock states in $M_{\text{safe}}||\boldsymbol{S}$ and show how to remove them.

Example 6. Figure 4 shows a message exchange sequence σ of TA, where the state q_6 is a non-accepting state. The scenario characterized by σ is that "after receiving the requests for vacation package from U, TA requests FB and HB for flight tickets and hotel rooms, respectively; when receiving the offer of FB, TA sends a confirmation message to book the flight, and then it is informed that there is no hotel room available to book" —In this scenario, the whole travel agency system (including U, FB, HB and TA) violates the correctness constrains φ_1 (i.e. selling vacation package). We find that TA will terminate in state q_6 when it receives message ?noH in state q_5 (q_5 is an accepting state of TA). So if TA can never reach state q_5 this scenario can be avoided.



?: Input action !: Output action ●: Accepting state

Figure 4 An example of deadlock.

Note that TA reaches state q_5 by sending messages fAck in state q_4, so we can avoid this scenario by controlling the behavior of TA through removing the output action !fAck and states q_5, q_6. Furthermore, if q_5 can be reached by another state q on action a in TA or there are other states except q_5 reached from q_4, we must treat q and q_4 in the same way if composite system reaches deadlock state when TA reaches q or q_4 after having removing q_5, q_6.

Let $M_{\text{safe}}||\boldsymbol{S} = (Q, \Sigma, q_0, F, \delta, Lab)$, and the set of all deadlock states of $M_{\text{safe}}||\boldsymbol{S}$ be DL. We design algorithm RemoveDeadlock which performs a backward reachability analysis on $M_{\text{safe}}||\boldsymbol{S}$ from every deadlock state in DL which traverses only input actions and removes all states reachable and states in DL. Then the set DL is constructed in terms of resulting DFA and the above process is continued until DL is empty. Intuitively, if the initial state of $M_{\text{safe}}||\boldsymbol{S}$ is removed, there will be an action sequence σ of M_{safe} such that M_{safe} must send messages according to σ when services in $\boldsymbol{S}$ send messages consistent with σ, and then $M_{\text{safe}}||\boldsymbol{S}$ will reach deadlock state.

Algorithm 2: RemoveDeadlock

1. Compute DL of $M_{\text{safe}}||\boldsymbol{S}$;
2. **while** $(DL \neq \phi)$\{
3. $D_0 = DL$;
4. **do** $D_{k+1} = D_k \bigcup InputBack(D_k)$ **until** $(D_{k+1} = D_k)$;
5. **if** $(q_0 \in D_k)$ **then return** failed;
6. **else** $M_{\text{safe}}||\boldsymbol{S} = (M_{\text{safe}}||\boldsymbol{S}) \backslash D_k$;
7. Compute DL of $M_{\text{safe}}||\boldsymbol{S}$;
8. \}**return** $M_{\text{safe}}||\boldsymbol{S}$;

We introduce the operator $InputBack$ over the power set $\mathcal{P}(Q)$. Intuitively, for a set Q' of states of $M_{\text{safe}}||\boldsymbol{S}$, the set $InputBack(Q')$ contains states of $M_{\text{safe}}||\boldsymbol{S}$ in which $M_{\text{safe}}||\boldsymbol{S}$ can enter a state in Q' by taking an input action. Formally,

$$InputBack(Q') = \{q' \in Q | \delta(q', ?m) \in Q'\}.$$

So if DL is not empty, the algorithm iterates the InputBack from DL until no new states are to be found (lines 2–4). If the initial state of $M_{\text{safe}}||\boldsymbol{S}$ is found by operator $InputBack$ the algorithm terminates in failing synthesis (line 5), which means M_{safe} cannot prevent a deadlock state from being entered in one or more steps; otherwise it removes all the states found by $InputBack$ (line 6). Then the algorithm repeats the above process until $DL = \emptyset$. Obviously, the time complexity of the RemoveDeadlock is $O(|Q| + |\Sigma|)$.

5.4 Discussion

We now analyze the soundness and completeness of AutoSyn and give optimization techniques for it.

Theorem 2. The synthesis approach AutoSyn can terminate and always produces mediator $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$ for a given services set $\boldsymbol{S}$ and correctness constraints $\varphi_{\text{control}} \wedge \varphi_{\text{data}}$ when it exists.

Proof sketch. The L^* algorithm guarantees to terminate and produce the most general safe mediator M_{safe} such that $M_{\text{safe}}||\boldsymbol{S} \models \varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. Moreover, $M_{\text{safe}}||\boldsymbol{S}$ has no unspecified reception because M_{safe} is most general. So the synthesis approach AutoSyn always terminates to produce $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$ if M_{safe} goes through the algorithm RemoveDeadlock and liveness properties checking.

We have proved that the synthesis of M_{safe} is PSPACE-hard, so the problem space may be exponential in the number of component services. The L^* algorithm needs only the polynomial number of membership and equivalence queries in worst case and the number of queries is dependent on the size of DFA it learns. So the time complexity of synthesizing M_{safe} is dependent on the size of M_{safe}. As a result, the time complexity of whole synthesis process for $M(\boldsymbol{S}, \varphi_{\text{control}} \wedge \varphi_{\text{data}})$ is dependent on the size of M_{safe} too. Then it is highly desired when the size of M_{safe} is much smaller than the whole state space.

Optimization techniques. In order to improve the efficiency of algorithm, it is desired to optimize the algorithm by reducing the number of queries as much as possible. Here we give two approaches to reducing the number of membership queries.

We do not ask membership query on string $\sigma \in \overline{\Sigma}^*$ in the following two situations because the answer must be false:

(1) if one of prefixes of σ has been known to be not in L_{safe}, the answer for membership query on σ must be false because L_{safe} is prefix-closed[14];

(2) let $\sigma = \sigma' \cdot a$, $a \in \Sigma_i$ and the membership query on σ' be true. Assume that the interaction $C_{\sigma'}||\boldsymbol{S}$ terminates in state $(q_1, q_2, \ldots, q_n, q_c)$. If service $S_i \in \boldsymbol{S}$ cannot execute action $\overline{a}$, the membership query on σ must be false because services in $\boldsymbol{S}$ cannot interact with C_σ completely, where C_σ (or $C_{\sigma'}$) is DFA accepting σ (or σ') and its prefixes exactly.

6 Implementation

We have implemented a prototype system AutoSyn with Java to verify our synthesis approach. As depicted in Figure 5, the AutoSyn system is composed of five main modules: the CTL Input GUI, the Message Mapping Input GUI, the Mediator Synthesizer, the BPEL2DFA Translator and the DFA2BPEL Translator.

The CTL Input GUI and Message Mapping Input GUI modules provide the users interfaces to input the composition requirements including CTL specification and message mappings.

The Mediator Synthesizer is the key module to implement the composition synthesis process in Figure 2, which comprises six submodules.

• The Message Mapping to CTL Translator is responsible for translating message mappings received through Message Mapping Input GUI into equivalent CTL specification.

• The M_{safe} Synthesizer implements the L^* algorithm: it takes in input the DFAs generated by the BEPL2DFA Translator and CTL specifications from CTL Input GUI and Message mapping to CTL Translator, and compute the most general safe mediator M_{safe} following section 5.

• The Deadlock Analysizer implements the deadlock analysis algorithm which eliminates all the deadlock states of $M_{\text{safe}}||\boldsymbol{S}$ if possible, and then outputs the result to Liveness Properties Verifier; otherwise it shows the synthesis fails.

• The Liveness Properties Verifier is responsible for checking the liveness properties of $M_{\text{safe}}||\boldsymbol{S}$ in composition requirements by invoking the NuSMV through the NuSMV Access Proxy.

• The NuSMV module is a model checker integrated in the system directly, which is invoked by the M_{safe} Synthesizer to answer queries and the Liveness Propeties Verifier to check liveness properties.

• The NuSMV Access Proxy is responsible for generating the input file of the NuSMV, extract-

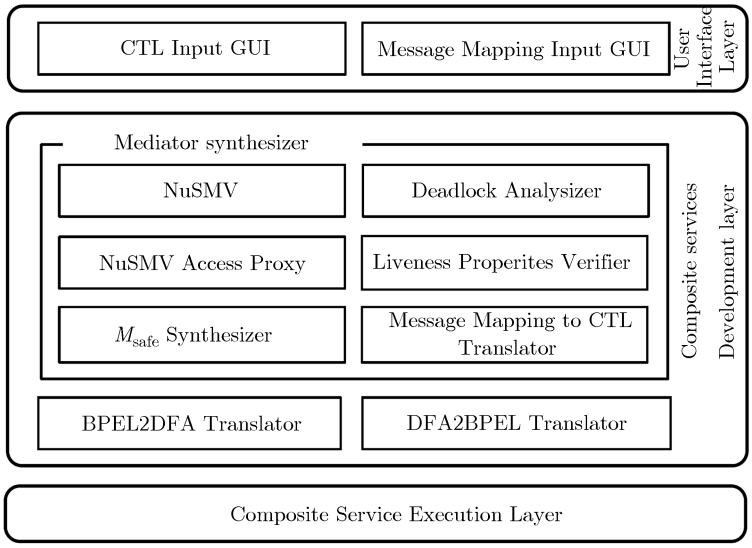


Figure 5 The AutoSyn implementation architecture.

ing useful information from the output file of the NuSMV and outputting them

The BPEL2DFA Translator and the DFA2BPEL Translator modules are used to translate the BPEL files of services into DFAs and translate the DFA model of final M into an executable BPEL file. We have implemented them by adopting the transformation mechanisms in refs. [15, 16], respectively.

7 Experimental study

In this section, we report the results of experiments we conducted for efficiency evaluation on AutoSyn. All experiments were performed on a PC using a 3.4 GHz Pentium D processor with 2 GB memory.

We take the synthesis of TA corresponding to Examples 1, 2 and 4 as an example. We synthesized the most general safe TA by using of L^* algorithm with 2898 member queries, 15 subset queries and 3 superset queries, where the NuSMV tool was invoked 555 times, and the execution time of L^* algorithm is about 25.094 s (see case 4 in Table 1). The most general safe TA has 40 states including one non-accepting state. Then we used the algorithm RemoveDeadlock to remove 13 deadlock states and related state transitions from the most general safe TA, and the result is shown in Figure 6. Note that for the sake of readability, we omit the message parameters in each action of TA in Figure 6, e.g., the action ?vReq refers to the action ?vReq(loc,date,vNum).

As shown in Figure 6, the path composed of only bold edges represents an interaction scenario for successfully booking vacation package, where the rest paths refer to failing booking because there is no available flight or hotel, or the user does not like the offers.

For example, the following is a successful scenario: after receiving requirement from the U (?vReq), the TA sends the requirement for hotel to HB(!hReq). Once receiving the hotel information from HB(?hRe), the TA sends the requirement for flight to FB(!fReq). And then the TA sends vacation information including hotel and flight information to the U (!vRe) after it receives the flight information from FB(?fRe). When the U confirms the booking (?vAck), the TA confirms the booking to HB (!hAck) and FB(!fAck).

Table 1 shows the preliminary experiment results including the number of states of most general safe mediator (States), number of membership queries (MQ), subset queries (SubQ), superset queries (SupQ) and times NuSMV has been invoked (NuSMV), and the execution time of L^* algorithm in seconds (Time). We observe that the runtime for deleting the deadlock states and checking the liveness properties is very short compared with the runtime of L^* algorithm, so we do not give them in Table 1.

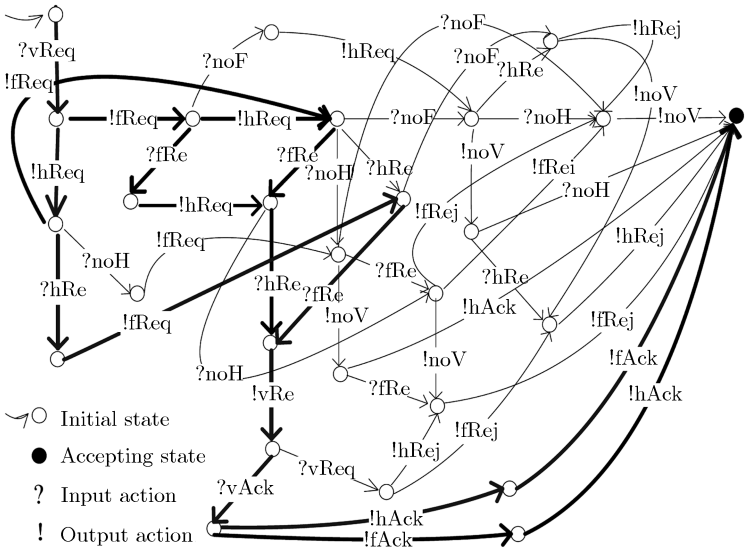


Figure 6 Business protocol of TA.

Table 1 Results of experiments

Case	States	MQ	SubQ	SupQ	NuSMV	Time (s)
1	9	147	5	2	44	1.968
2	14	370	4	6	91	4.750
3	24	984	12	2	212	9.375
4	40	2898	15	3	555	25.094
5	59	8555	29	9	909	62.078

In cases 1 and 2, we consider the travel agency as only composed of U and FB, where case 2 has more complex interaction scenarios than case 1. Specifically, in case 2, U and FB are the same as those in Example 1, while they are essentially request-response protocols in case 1. In cases 3 and 4, the travel agency is composed of U, HB and FB. Case 4 is exactly the example corresponding to Examples 1, 2 and 4, and case 3 is simplified such that U, HB and FB are request-response protocols. To validate the results of this experimental evaluation, in case 5, we conducted a more complex travel agency in which a train booking service TB is added in case 4, with the interaction process in business protocol of TB similar to that of FB's and HB's. We require that the TA can book both flight and hotel or train and hotel if there are no flights available.

The experimental results in Table 1 show that for the typical travel agency example in SOA application, our synthesis approach is feasible and takes a rather low amount of time. The experimental results also show that the run time of L^* algorithm is affected by the number of states of resulting most general safe mediator and number of queries, especially numbers of times NuSMV have been invoked for answering some queries. Different from the work in refs. [12, 17] in which each query must be answered by invoking model checker NuSMV, our algorithm implementation only occasionally invokes NuSMV to answer a small proportion of queries, making the AutoSyn for composition synthesis problem efficient.

8 Related work

A lot of approaches have been developed to realize the automated synthesis of composite services which rely on formal methods such as automata theory, AI planning, logical reasoning, theorem proving, etc.[1].

There are many studies based on automata model related to our work. Bultan et al.[3] modeled global behavior of e-service composition in asynchronous messages as conversation based on Mealy machine and discussed the realizability and synchronization of conversations. The Roman model[18] specified the goal service and e-services as finite state machine and reduced the composition synthesis into satisfiability problem of PDL formula. The Colombo model[2] extended[18] with message passing and world state of local database. Fan et al.[19] modeled the goal service and services as alternating finite automata and discussed the

time complexity of composition synthesis by exploring connections between composition synthesis and query rewriting using view. Pathak[20] presented a framework MoSCoE in which the services and goal service were modeled as symbolic transition system, and an algorithm was given to realize the automated choreographer synthesis. Mitra[21] modeled services and goal service as I/O automata, and reduced the existence problem of choreography to the simulation of I/O automata.

The above researches all need developer to provide a detailed behavior specification for composite service beforehand, e.g. the conversation[3] and goal service[2,18−21]. In this paper, we only need developer to provide the correctness constraint on composition and automatically synthesize the behavior of composite service. In addition, the work in refs. [19, 21] only limited in theoretical research without providing practice algorithm for composition.

Pistore et al.[7] presented an approach Astro which modeled automated service composition as a planning problem. They modeled services as state transition system (STS) and used the EAGLE language to denote composition requirement for control dependencies. They further modeled dataflow requirement as STS too in ref. [22], and constructed the planning domain as the synchronized product of services and dataflow requirement. In contrast, we only consider simple dataflow dependency and unify the model of control dependencies and data dependencies into CTL specification, and perform the on-the-fly synthesis without building the whole state space like planning domain, thus reducing the time and space consumption. Moreover, our approach guarantees that the composite service is minimal which is not satisfied in refs. [7, 22].

In addition, there are other papers on automated service composition which consider a service as function with input and output, and use the classical planning approach in AI[23,24] or type derivation algorithm[25]. However, in AutoSyn, services are characterized as more general DFA model which has the function with input and output as a special case.

Finally, there is other work of using L^* algorithm to learn a DFA[12,17]. Alur et al.[12] used L^* algorithm to synthesize a safe interface specification for Java classes. Cobleigh et al.[17] used L^* algorithm to learn assumptions for compositional verification. In this paper, we use the L^* algorithm to synthesize a mediator for a given services set and correctness constraints. In their work, every query needs to be answered by invoking model checker NuSMV, while in our setting, only part of queries need invoking NuSMV, which reduces running time dramatically.

9 Conclusions

We have proposed a novel approach AutoSyn for automatically synthesizing a mediator from a given services set and correctness constraints on control dependencies and data dependencies. The technique is based on learning algorithm. A salient feature of the method is that it guarantees that the generated mediator be minimal DFA and the most general way of coordinating component services, so that the correctness constraints are not violated. Despite the intractability of the synthesis problem, the time complexity of our synthesis algorithm based on L^* algorithm is dependent on the size of the most general safe service M_{safe}, which reduces the time complexity dramatically when the size of M_{safe} is much smaller than the entire state space. We have also implemented a prototype system and the preliminary experimentation shows promising results.

There is much more to be done. First, in AutoSyn, there are two cases in which the synthesis is bound to fail; that is, at least one deadlock state cannot be removed or the liveness properties are not satisfied. So, we will design an approach adjusting the correctness constraints to make the synthesis successful. Second, fault handling and compensation are also an important problem in service composition, which needs a more expressive synthesis model. So an investigation into how to extend our work to deal with fault handling and compensation will be our future work.

We thank Professor Wenfei Fan of University of Edinburgh for his fruitful suggestions and kind help in the writing of this paper,

and anonymous reviewers for their constructive comments and suggestions.

Appendix A Proof of Theorem 1

We show the PSPACE-hardness by reduction from Q3SAT which is known to be PSPACE-complete[11]. An instance of Q3SAT is a quantified Boolean formula $\phi = Q_1x_1Q_2x_2\ldots Q_nx_nE$, where $E = C_1 \wedge \ldots \wedge C_n$ is an instance of 3SAT in which all the propositional variables are $x_1, \ldots, x_n$, and $Q_i \in \{\forall, \exists\}$ for each $i \in [1, n]$. The problem is to decide, given such a quantified Boolean formula ϕ, whether or not ϕ is true.

Given $\phi = Q_1x_1Q_2x_2\ldots Q_nx_nE$, without loss of generality, we assume that n is even and $Q_i = \forall$ if i is odd; otherwise, $Q_i = \exists$ (we can extend any quantified Boolean formula into this form by adding propositional variables). We encode ϕ in terms of a service set $\boldsymbol{S}$ with $(n+2)/2$ component services $S_1, \ldots, S_{(n+2)/2}$ and constraints $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$. For every $i \in [1, n]$, there are two output actions $!m_i, !m'_i$ corresponding with x_i if $Q_i = \forall$ (i.e. i is odd); otherwise, there are two input actions $?m_i, ?m'_i$ corresponding with x_i.

Intuitively, service S_1 assigns different true values to x_1 after sending messages m_1 and m'_1 in the initial state respectively; service $S_i (1 < i < (n+2)/2)$ first assigns different truth values to variable x_{2i-2} after receiving messages m_{2i-2} and m'_{2i-2} in the initial state, respectively, then assigns different truth values to variable x_{2i-1} after sending messages m_{2i-1} and m'_{2i-1}, respectively; and service $S_{(n+2)/2}$ assigns different truth values to x_n after receiving messages m_n and m'_n in the initial state, respectively.

More specifically, let p_i and $\neg p_i$ refer to the truth value assignment $x_i = 1$ and $x_i = 0$, respectively,

(1) $S_1 = (Q_1, \Sigma_1, q_{10}, \delta_1, Lab_1)$, where

- $Q = \{q_{10}, q_{11}, q'_{11}\}$,
- $\Sigma_1 = \{!m_1, !m'_1\}$,
- $\delta(q_{10}, !m_1) = q_{11}$, $\delta(q_{10}, !m'_1) = q'_{11}$,
- $Lab_1(q_{10}) = \emptyset$, $Lab_1(q_{11}) = \{p_1\}$ and $Lab_1(q'_{11}) = \{\neg p_1\}$.

(2) for $1 < i \leqslant n/2$, $S_i = (Q_i, \Sigma_i, q_{i0}, \delta_i, Lab_i)$, where

- $Q = \{q_{i0}, q_{i1}, q'_{i1}, q_{i2}, q'_{i2}, q''_{i2}, q'''_{i2}\}$,
- $\Sigma_i = \{?m_{2i-2}, ?m'_{2i-2}, !m_{2i-1}, !m'_{2i-1}\}$,
- $\delta(q_{i0}, ?m_{2i-2}) = q_{i1}$, $\delta(q_{i0}, ?m'_{2i-2}) = q'_{i1}$, $\delta(q_{i1}, !m_{2i-1}) = q_{i2}$, $\delta(q_{i1}, !m'_{2i-1}) = q'_{i2}$, $\delta(q'_{i1}, !m_{2i-1}) = q''_{i2}$, $\delta(q'_{i1}, !m'_{2i-1}) = q'''_{i2}$.
- $Lab_i(q_{i0}) = \emptyset$, $Lab_i(q_{i1}) = \{p_{2i-2}\}$, $Lab_i(q'_{i1}) = \{\neg p_{2i-2}\}$, $Lab_i(q_{i2}) = \{p_{2i-2}, p_{2i-1}\}$, $Lab_i(q'_{i2}) = \{p_{2i-2}, \neg p_{2i-1}\}$, $Lab_i(q''_{i2}) = \{\neg p_{2i-2}, p_{2i-1}\}$, $Lab_i(q'''_{i2}) = \{\neg p_{2i-2}, \neg p_{2i-1}\}$.

(3) $S_{(n+2)/2} = (Q_{(n+2)/2}, \Sigma_{(n+2)/2}, q_{(n+2)/2,0}, \delta_{(n+2)/2}, Lab_{(n+2)/2})$, where

- $Q_{(n+2)/2} = \{q_{(n+2)/2,0}, q_{(n+2)/2,1}, q'_{(n+2)/2,1}\}$,
- $\Sigma_{(n+2)/2} = \{?m_n, ?m'_n\}$,
- $\delta(q_{(n+2)/2,0}, ?m_n) = q_{(n+2)/2,1}$ and $\delta(q_{(n+2)/2,0}, ?m'_n) = q'_{(n+2)/2,1}$.
- $Lab_{(n+2/2)}(q_{(n+2)/2,0}) = \emptyset$, $Lab_{(n+2/2)}(q_{(n+2)/2,1}) = \{p_n\}$, $Lab_{(n+2/2)}(q'_{(n+2)/2,1}) = \{\neg p_n\}$.

Moreover, let the safety properties $\varphi_{\text{safe}} = \mathbf{AG}\, p$, where for service S, we say that $S, q \models \neg p$ if the formula $\phi = 0$ under the truth assignment of S in state q. And φ_{data} is as follows:

$$\varphi_{\text{data}} = \bigwedge_{j=1}^{n/2} (\varphi(m_{2j}) \wedge \varphi(m'_{2j})).$$

Here $\varphi(m_{2j}) = \mathbf{AG}(m_{2j} \rightarrow m_{2j-1} \vee m'_{2j-1})$ and $\varphi(m'_{2j}) = \mathbf{AG}(m'_{2j} \rightarrow m_{2j-1} \vee m'_{2j-1})$. Obviously, the above transformation is in PTIME. It is easily verified that ϕ is true if and only if there exists a mediator $M(\boldsymbol{S}, \varphi_{\text{safe}} \wedge \varphi_{\text{data}})$ which can receives all output messages from services $S_i (i \in [1, n/2])$ and sends messages to $S_{i+1} (i \in [1, n/2])$ such that the interaction do not violate $\varphi_{\text{safe}} \wedge \varphi_{\text{data}}$ and is free of deadlock and unspecified receptions.

1 Hull R, Su J. Tools for composite web services: a short overview. SIGMOD Rec, 2005, 34(2): 86–95

2 Berardi D, Calvanese D, Giacomo G, et al. Automatic composition of transition-based semantic web services with messaging. In: Proceeding of 31th International Conference on Very Large Databases, Trondheim, Norway, 2005. 613–624

3 Fu X, Bultan T, Su J. Conversation protocols: a formalism for specification and verification of reactive electronic services.

Theor Comput Sci, 2004, 328(1-2): 19–37
4 Probert R L, Saleh K. Synthesis of communication protocols: Survey and assessment. IEEE Trans Comput, 1991, 40(4): 468–476
5 Magee J, Kramer J, Uchitel S, et al, LTSA-WS: a tool for model-based verification of web service compositions and choreography, In: Proceeding of International Conference on Software Engineering, Shanghai, China, 2006. 771–774
6 Rintanen J. Computational complexity of plan and controller synthesis under partial observability. Technical Report, 2005. http://users.rsise.anu.edu.au/ jussi/Rintanen05compl.pdf.
7 Pistore M, Traverso P, Bertoli P, et al. Automated synthesis of composite bpel4ws web services. In: Proceeding of International Conference on Web Services, Orlando, Florida, USA, 2005. 293–301
8 Rivest R L, Schapire R E. Inference of finite automata using homing sequences. Inf Comput, 1993, 103(2): 299–347
9 Clarke E M, Grumberg O, Peled D A. Model Checking. Cambridge, MA: MIT Press, 2000
10 Manolios P, Trefler R J. Safety and liveness in branching time. In: Proceeding of 16th Annual IEEE Symposium on Logic in Computer Science, Boston, MA, USA, 2001. 366–374
11 Garey M R, Johnson, D S. Computers and Intractability: A Guide to the Theory of NP-Completeness. New York: Freeman W. H. & Co., 1990
12 Alur R, Cerny P, Madhusudan P, et al. Synthesis of interface specifications for java classes. In: Proceeding of Symposium on Principles of Programming Languages, Long Beach, California, USA, 2005. 98–109
13 Milner R. Communicating and Mobile Systems: the Pi-Calculus. Cambridge: Cambridge University Press, 1999
14 Berg T, Jonsson B, Leucker M, et al. Insights to angluin's learning. In: Proceeding of International Workshop on Software Verification and Validation, Mumbai, India, 2003
15 Wombacher A, Fankhauser P, Neuhold E J. Transforming bpel into annotated deterministic finite state automata for service discovery. In: Proceeding of International Conference of Web Services, San Diego, California, USA, 2004. 316–323
16 Berardi D, Calvanese D, Giacomo G D, et al. $\mathcal{ESC}$: A tool for automatic composition of services based on logics of programs. In: Proceeding of Workshop on Technologies for E-Services, Toronto, Canada, 2004. 80–94
17 Cobleigh J M, Giannakopoulou D, Pasareanu C S. Learning assumptions for compositional verification. In: Proceeding of International Conference on Tools and algorithms for the Construction and Analysis of Systems, Warsaw, Poland, 2003. 331–346
18 Berardi D, Calvanese D, Giacomo G, et al. Automatic composition of e-services that export their behavior. In: Proceeding of International Conference on Service Oriented Computing, Trento, Italy, 2003. 43–58
19 Fan W, Geerts F, Gelade W. Complexity and composition of synthesized web services. In: Proceeding of Symposium on Principles of Database Systems, Vancourver, Canada, 2008. 231–240
20 Pathak J, Basu S, Lutz R, et al. Parallel web service composition in moscoe: A choreography-based approach. In: Proceeding of European Conference on Web Services, Zurich, Switzerland, 2006. 3–12
21 Mitra S, Kumar R, Basu S. Automated choreographer synthesis for web services composition using i/o automata. In: Proceeding of International Conference on Web Services, Salt Lake City, Utah, USA, 2007. 364–371
22 Marconi A, Pistore M, Traverso P. Specifying data-flow requirements for the automated composition of web services. In: Proceeding of IEEE International Conference on Software Engineering and Formal Methods, Pune, India, 2006. 147–156
23 Ponnekanti S, Fox A. Sword: A developer toolkit for web service composition. In: Proceeding of World Wide Web Conference, Honolulu, Hawaii, USA, 2002
24 Sirin E, Parsia B, Wu D, et al. HTN planning for web service composition using shop2. J Web Semant, 2004, 1(4): 377–396
25 Pu K, Hristidis V, Koudas N. Syntactic rule based approach toweb service composition. In: Proceeding of International Conference on Data Engineering, Atlanta, Georgia, USA, 2006. 31–40