IEICE TRANS. INF. & SYST., VOL.E95–D, NO.7 JULY 2012

PAPER

Automated Adaptor Generation for Behavioral Mismatching Services Based on Pushdown Model Checking*

Hsin-Hung LIN[†a)], **Toshiaki AOKI**[†b)], *Nonmembers*, *and* **Takuya KATAYAMA**[†c)], *Member*

SUMMARY In this paper, we introduce an approach of service adaptation for behavior mismatching services using pushdown model checking. This approach uses pushdown systems as model of adaptors so that capturing non-regular behavior in service interactions is possible. Also, the use of pushdown model checking integrates adaptation and verification. This guarantees that an adaptor generated by our approach not only solves behavior mismatches but also satisfies usual verification properties if specified. Unlike conventional approaches, we do not count on specifications of adaptor contracts but take only information from behavior interfaces of services and perform fully automated adaptor generation. Three requirements relating to behavior mismatches, unbounded messages, and branchings are retrieved from behavior interfaces and used to build LTL properties for pushdown model checking. Properties for unbounded messages, i.e., messages sent and received arbitrary multiple times, are especially addressed since it characterizes non-regular behavior in service composition. This paper also shows some experimental results from a prototype tool and provides directions for building BPEL adaptors from behavior interface of generated adaptor. The results show that our approach does solve behavior mismatches and successfully capture non-regular behavior in service composition under the scale of real service applications.

key words: *service adaptation, behavior mismatch, pushdown model checking, unbounded messages*

1. Introduction

Service composition has been recognized as one of the major issues in service oriented computing (SOC). The basic idea of SOC is to reuse services already developed, especially by other providers. Since existing services are already implemented and tested, reusing them with valid composition should save time and costs in developing new applications. However, direct composition of services is nearly impossible because of incompatibilities/mismatches of interfaces. To perform service composition while mismatches exist, service adaptation [2] provides a promising solution to compose mismatching services. Service adaptation introduces a mediate service called adaptor which coordinates interactions of services. In an adapted system of services, all interactions among services are through the adaptor so that mismatches can be avoid without modifying given services, which provides a non-intrusive way for service composition. Therefore, Services to be composed are treated as black boxes so that the purpose of service reuse can be achieved.

Existing approaches for service adaptation are, to our best knowledge, generally based on a conventional framework of software adaptation [3]. The framework is designed to use *behavior interfaces* and *adaptation contracts* to perform automated adaptor generation. More specifically, behavior interfaces requires specifications of behavior interfaces of services represented in labeled transition system (LTS), while adaptation contracts require specifications of mappings of messages to be interacted and the ordering (i.e., expected interactions coordinated by an adaptor) of these mappings represented in LTS. However, given some services that need adaptation, behavior of the adaptor as well as the system behavior may be non-regular and can not be represented in LTS, even if mapping of messages is correctly specified.

In order to tackle non-regular behavior in adaptors, we introduce an approach that uses pushdown automata model for representing behavior of adaptors. The pushdown automata model makes computations with non-regular behavior possible while not being theoretically too complicated. This is reasonable under the view point of software engineering and we still treat services as finite state machines so that our approach is applicable to the systems in conventional framework. Furthermore, our approach uses model checking technique [4] to perform mismatch detection and adaptor generation. The basic idea is to generate Linear Time Logic (LTL) properties corresponding to requirements for a system to be adapted. Thus, adaptation is integrated with verification, i.e. properties related to adaptation and verification can be put together and model checked at the same time. This means a generated adaptor by our approach not only solves behavior mismatches but also guarantees properties such as safety and liveness if specified.

Besides non-regular behavior, our approach also attempts to deal with issues in designing adaptor contracts. In the conventional framework, adaptor contracts include two parts: mapping of transition labels in behavior interfaces, and the execution of interactions corresponding to these mappings. In the former part, mappings are suggestions for solving signature mismatches while the later part is the behavior of an adaptor designed by developers. Generally, both parts have to be specified before performing adaptor generation and it counts on developers to see through behavior interfaces of services to design proper message mappings and behavior of an adaptor. Since determining

Manuscript received April 21, 2011.
Manuscript revised January 31, 2012.

[†]The authors are with Japan Advanced Institute of Science and Technology, Nomi-shi, 923–1292 Japan.

*This paper is an extended version of our recent work [1].

a) E-mail: h-lin@jaist.ac.jp
b) E-mail: toshiaki@jaist.ac.jp
c) E-mail: katayama@jaist.ac.jp

DOI: 10.1587/transinf.E95.D.1882

mappings of transition labels usually needs extra information other than behavior interfaces that need additional techniques such as context aware technology, we assume that necessary mappings are already given and leave this part remaining manually specified. On the other hand, executions of interactions can be obtained by analyzing behavior interfaces of services and therefore directly be used to generate adaptors. Some tools have already been developed to interactively support designing adaptor contacts [5], [6] as LTS.

Unlike these tools, our approach does not generate adaptor contracts but directly generates adaptors from properties obtained from behavior interfaces of services. The properties are automatically retrieved so that fully automated adaptor generation without adaptor contracts is possible. This feature makes our approach especially suitable for adaptation of mobile services being dynamically found and composed. More specifically, by analyzing behavior interfaces of services, we may determine and generate properties for *behavior mismatch*, *unbounded messages*, and *branchings* which reflects requirements implicitly described in behavior interfaces. The first property is the key of solving behavior mismatch and the rest are information about accomplishment of functionalities of services. Among these properties, we are especially interested in *unbounded messages* since this property characterizes non-regular behavior that we are focusing in this approach.

The rest of this paper is organized in the following structure: Section 2 discusses related work and addresses the position of this paper; Section 3 demonstrates our approach using a motivational example to explain core ideas; Section 4 describes formal definitions of web services and adaptors; Section 5 demonstrates the detail of adaptor generation which is the major part of this paper; Section 6 shows a prototype tool and introduces directions of implementing BPEL adaptors. Section 7 shows some experimental results using a prototype tool and gives some discussions; Finally, Sect. 8 summarizes contributions of this paper and gives some future directions.

2. Related Work

As a hot topic in component-based software engineering (CBSE), many approaches are proposed for software adaptation. Most approaches, including ours, focused on solutions for behavior mismatches between abstract behavior interfaces. Brogi et al. proposed a model-based approach [3] which uses Labeled Transition Systems (LTSs) for modeling and calculation of software adaptation. According to our survey, this approach has defined a conventional framework for software adaptation using two basic elements: *behavior interfaces* and *adaptation contracts* which are both modeled by LTS. Some work, though using different approaches on computation, is based on this framework. Tivil et al. [8] proposed a computation technique which directly constructs partial behavior of adaptor from corresponding software components, This technique gives more computational efficiency to adaptor generation while incapable of solving reordering mismatches. This technique can also integrate LTL model checking by directly composition with Büchi automata transformed from specified LTL properties. Mateescu et al. [7] used process algebra for modeling behavior, LOTOS for specification of protocols, and CADP toolbox for automated on-the-fly adaptor generation.

Recently, approaches for service adaptation became popular and techniques of software adaptation mentioned above were extended or modified for service composition. Cubo et al. applied the approach of [3] to WF/.NET framework and added verifications in their approach [9]. Mateescu et al. also extended their work in [7] to service adaptation using the model of Symbolic Transition Systems (STSs) [2]. Some other work used their own definitions for adaptation. Nezhad et al. [10] defined their own interfaces including sets of XML data and introduced an algorithm for solving interface mismatches. Mitra et al. [11] used I/O automata with history to support the multiple uses of same messages in service composition. Compare to above work, our approach attempts to capture non-regular behavior in service composition and uses pushdown automata model for representing adaptors. The use of model checking technique integrates adaptation and verification so that generated adaptor is guaranteed to satisfy both behavior mismatch free and safety/liveness properties if specified. Our previous work [12] proposed the first version of our approach that uses Büchi automata model for behavior interfaces of services. The work proposed a property called *behavior mismatch free* defined from acceptance condition of Büchi automata. To our best knowledge, this work was the first time generation of non-regular behavioral adaptor is tackled.

Another topic in service adaptation is automated generation of adaptation contracts. For web services, it becomes a problem that adaptation contracts have to be manually specified in the conventional framework proposed in [3] while there are mobile services that demand being selected and composed dynamically. Some research about adaptation also tackles this topic in various ways. J.A. Martín and E. Pimental [13] proposed an expert system based approach which combines exploring rules and A^* graph search algorithm. Their approach automatically generates adaptation contracts (mainly mappings of labels) having the best score. Other work provides semi-automated way to guide design of adaptation contracts. Nezhad et al. [10] introduced an interactive way for users to specify adaptation contracts related to behavior mismatches on reordering. Cámara et al. developed an integrated tool ITACA [5] to support composition of BPEL services which provides interactive graphical user interface to guide the design of adaptor contracts. Compare to above work, our approach does not generate adaptation contracts but directly generates an adaptor rely on only information from behavior interfaces of services. Assuming signature mismatches are solved and mappings of labels are specified, our approach provides fully automated adaptor generation. This is proposed in our recent work [1]. Furthermore, we especially address property for *unbounded messages* which characterize non-regular behavior of adaptors.

Though the use of model checking technique is similar to exploring graph structures of behavior interfaces, we argue that the use of model checking brings more advantages since model checking techniques are improved rapidly as well as the feature of performing both service adaptation and verifications at the same time.

3. Approach

In this section, we first use a motivational example to demonstrate core ideas of adaptor generation, then give an overview of our approach.

3.1 Motivational Example

A motivational example "Fresh Market Update Service" is shown in Fig. 1 [†]. In this example, three services are given their behavior interfaces [††] for composition. For each transition, a label prefixed with "!" represents a message emission event and the prefix of "?" represents an event of message reception. Since synchronization is common in designing web services even the implementation is based on asynchronous communication, we assume synchronous composition between services in our approach. For more details, every transition is supposed to be synchronized with another transition in a different service having the label of same message name but opposite prefix symbol. It is easy to notice that among the three services there is an behavior mismatch caused by ordering of messages: message `Start` is designed to be received by `Investor` before it receives `Data` while the ordering of the two messages being sent is first `Data` then `Start`. A proper adaptor may be designed to have behavior like $(?R\ !R\ ?D)^n\ ?E\ !E\ ?S\ !S\ !D^n\ ?C\ !C\ ?A\ !A)$, $n > 1$, where capital letters are abbreviations of message names. An adaptor is supposed to synchronize with services and therefore no direct interaction between services under coordination of an adaptor. Thus, transitions of the adaptor with labels prefixed with "?" and "!" represent reception and delivery of corresponding messages in adaptor and are supposed to be synchronized with corresponding transitions (i.e., same message names with opposite prefixes) in services. Note that in this example, the behavior of the three services implies that numbers of message `RawData` and `Data` being sent and received are the same and results in non-regular behavior of the adaptor.

3.2 Core Ideas

We may call the adaptor with non-regular behavior mentioned in Sect. 3.1 the *expected adaptor* and then discuss the core ideas of generating such an adaptor in our approach.

Pushdown System Model: In our approach, an adaptor should satisfy the following concerns: (1) an adaptor does not generate any message by itself; (2) an adaptor only receives messages sent from services; (3) an adaptor only sends previously received messages; (4) an adaptor is expected to send all received messages eventually. By choosing pushdown automata model for behavior of adaptors, (3) and (4) can be perfectly satisfied. Therefore, functionalities of given services are guaranteed as long as the stack of the adaptor is empty finally at the end of execution. This is, not possible for finite state machines without extra conditions/specifications. Furthermore, for our approach, the general pushdown automata model is not necessary and a simplified pushdown systems model is introduced in Sect. 4.

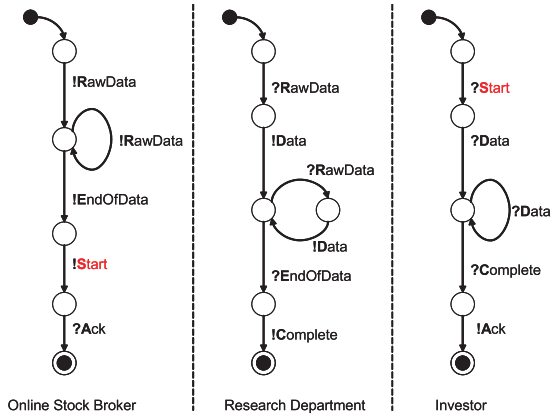


Fig. 1 Fresh Market Update Service.

Unbounded Messages: In the behavior of *expected adaptor*, the numbers of messages `RawData` and `Data` being sent/received are required to be the same arbitrary nature number n. Though it is the structures of behavior interfaces that decide the equality in numbers of messages, an adaptor to be generated is still demanded to reflect the characteristic of arbitrary number n in its behavior. This characteristic is essential to non-regular behavior in service interactions and is representable under the use of pushdown systems model in our approach while the conventional framework using LTS model does not consider or support this characteristic. We may call the messages being sent/received arbitrary multiple times *unbounded messages* and define the set of *unbounded messages* as $A^{UB} = \{a \mid a \in A \ \wedge\ \forall\sigma, \exists\sigma', Occ(\sigma', a) > Occ(\sigma, a)\}$ where σ and σ' are accepting traces of the system behavior and $Occ(\sigma, a)$ represents the number of occurrence of message a in σ.

Service as One Session Process: Services in our approach are required to be defined as one-session processes. This means behavior of a service has a start and an end state and these two states should be different. Generally, this requirement naturally fits definition languages of services such as BPEL [15]. Though services are basically running continuously and services defined in automata models such as LTS are usually have one same state representing both start and end states, this causes a problem in dealing with *unbounded messages*: unwanted traces may also be included in the generated adaptor depending on behavior interfaces of given services. The cause of this problem is that some services may start another session (or run of execution) before

[†]Original version is proposed by X. Fu et al. [14] to show a case that LTL model checking is undecidable.

[††]LTS or any finite state machine model is fine.

all messages are consumed so that these unconsumed messages may be consumed in later sessions. However, since a session is usually considered a period that functionalities are expected to be accomplished, it is preferred that messages sent in a session are consumed in the same session. These demands an implicit requirement that all given services accomplish their common functionalities when they all reaches their end states. This requirement also means that all services should reach their end states before going to next round of execution and is not possible to be represented by only temporal properties. Therefore, we define a constrained behavior model of services called Interface Automata for Web Services (IA4WS) in Sect. 4 in order to reveal this implicit requirement.

3.3 Overview of the Approach

Generally, our approach takes behavior interfaces of services as the input and the output is behavior interface of an adaptor if direct composition of given services does not satisfy designated properties. Behavior interfaces of services are represented by Interface Automata [16] and adaptors are represented by pushdown systems [17]. Our approach has the following steps to perform service adaptation:

Compatibility Check: Our approach focuses on behavior mismatches therefore we assume signature mismatches are solved [†] before applying our approach. This assumption is defined as *compatibility* of given services which means that any message delivered from a service is received by another service.

Detection of Behavior Mismatch: If given services pass the compatibility check, detection of behavior mismatches is performed to check if behavior mismatches exist. First, the system behavior is computed by synchronous composition [18] of services, which is also an interface automaton. Then the property of *behavior mismatch free* for the system behavior is built as a linear temporal logic (LTL) formula [18]. This property says that all execution traces of the system behavior must eventually visit the final state, i.e., deadlock free for the system. With the property and the corresponding labeling function, we can apply model checking for this property. Practically, behavior interfaces of services are implemented in Promela with one synchronous message queue for communication. Then SPIN [18] can do both synchronous composition and model checking for us. Two cases of results are expected: (1) a counterexample which means behavior mismatches exist; (2) the system behavior passes the property so that we do not need an adaptor. For (2), the approach ends here; for (1), we proceed to adaptor generation. Since this part is intuitive and can be easily implemented in Promela model for SPIN, we skip the details in this paper.

Adaptor Generation: In our approach, adaptor generation is called *coordinator guided adaptor generation*. We first need a special adaptor called *coordinator* which is ready to receive and send any message if there is a corresponding service ready for sending or receiving the message. The synchronous composition of given services with *coordinator*, which is also a pushdown system, then captures all possible interleaved interactions of given services. Here we prepare three properties: property of *behavior mismatch free*, fairness property of *looped transitions*, and fairness property of *branching transitions*. The first property represents condition of deadlock free for the coordinator involved system behavior. The second and the third properties represent implicit requirements of *unbounded messages* and *branching*. Then pushdown model checking [17] is performed to check the *negation* of these properties. Thus, a returned counterexample is an execution trace that satisfies these properties and an adaptor is generated from this counterexample. Details about this step will be given in Sect. 5.

4. Formal Definitions of Services and Adaptors

This section gives the definition of models for behavior interfaces of services and adaptors.

4.1 Service

In our approach, services and adaptors are represented by different models: Interface Automata for Web Services (IA4WS) and Interface Pushdown Systems (IPS). IA4WS, defined in Def. 1, is an automaton model modified from Interface Automata (IA) which is defined using the notion of input, output and internal alphabets for the purpose of composing two general software components represented in IA. Since we are focusing on service adaptation, we add constraints as well as extensions to IA to fit the purpose of service adaptation in our approach. An IA4WS is constrained by adding the following conditions on an IA: (1) there is only one initial state and one final state; (2) no transition goes from the final state and to the initial state. The purpose of the two conditions is already mentioned in Sect. 3.2 as the core idea of *services as one session process*. Alphabets in an IA4WS are abstractions of messages being sent or received, or indicating internal actions.

Definition 1 (IA4WS): An interface automaton for web service is defined as $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$, where

Q: finite set of states.
$q^0 \in Q$: initial state.
A^I: finite set of input alphabets.
A^O: finite set of output alphabets.
A^H: finite set of internal alphabets.
$\Delta \subseteq Q \times A \times Q$: set of transition relations, where $A = A^I \cup A^O \cup A^H$
$q^f \in Q_i$: final state.

An IA4WS has to satisfy the following conditions:

$q^0 \neq q^f$
$\nexists t \in \Delta,\ t = (q, a, q'),\ q, q' \in Q,\ q = q^f \vee q' = q^0$

[†]Practically, we can introduce special services that plays the role of mapping labels in adaptor contracts.

$\forall a \in A.\ \exists t \in \Delta,\ t = (q, a, q'),\ q, q' \in Q$

Given a set of services represented in IA4WSs for our approach to perform service adaptation, it is required in our approach that these services meet the requirement of compatibility. Compatibility defined in Def. 2 means that every message sent by one service is always received by another service. This constructs a closed system of services and we call services that satisfy compatibility are composable services. It should be noticed that the compatibility defined in IA is only for two components to be composed in a open system.

Definition 2 (Compatibility): A set of interface automata for web services $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$,$i \in [1, n]$, are composable if

$$A_i^I \cap A_i^O = \emptyset, \quad A_i^I \cap A_j^I = \emptyset,\ i \neq j,$$
$$A_i^O \cap A_i^O = \emptyset,\ i \neq j, \quad \bigcup_i A_i^I = \bigcup_i A_i^O,$$
$$\bigcup_i A_i^H \cap \bigcup_i A_i^I = \emptyset.$$

4.2 Adaptor

We use IPS defined in Def. 3 as model of adaptors. Generally, an IPS is a pushdown system [17] enhanced with notations of input and output alphabets in order to be composed with services represented in IA4WSs. Note that in a transition rule $(p, a, \gamma) \hookrightarrow (p', w)$, w does not represent the contents of the stack but only the word that replaces the head symbol of stack γ after transition is fired. Transitions of an IPS are restricted in three kinds: push, pop and internal. Push transitions represent message reception and pop transitions represent message delivery. Internal transitions does not related to message exchange with services. The definition of IPS is shown in Def. 3.

Definition 3 (Interface Pushdown System): An interface pushdown system is defined as tuples: $S = (Q, q^0, \Gamma, z, T, F)$, where

Q: finite set of states.
q^0: initial state.
Γ: finite set of stack symbols.
z: stack start symbol representing bottom of stack. $z \in \Gamma$.
$T \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$: set of transition relations.
F: finite set of final states.
T is restricted to the following three patterns:

$< p, \gamma > \hookrightarrow < p', a\gamma >$: push transition,
$< p, a > \hookrightarrow < p', \epsilon >$: pop transition,
$< p, \gamma > \hookrightarrow < p', \gamma >$: internal transition,
where $a, \gamma \in \Gamma$, $a \neq z$.

In our approach, an adaptor is defined as an IPS with further constraints: (1) alphabets of an adaptor are the union of alphabets of given service; (2) all states in an adaptor are considered final states and stack emptiness is used to define the acceptance condition. The definition of adaptor is shown in Def. 4.

Definition 4 (Adaptor): Given a set of composable IA4WS: $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$. An adaptor for P_i, $i \in [1, n]$ is an IPS: $D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$, where $\Gamma_D = A_D \cup \{z\}$, $A_D = \bigcup_i A_i^I = \bigcup_i A_i^O$, and $F_D = Q_D$.

5. Adaptor Generation

This section gives technical details of adaptor generation called *coordinator guided adaptor generation*.

5.1 Coordinator Guided Adaptor Generation

As mentioned in Sect. 3.3, the adaptor generation in our approach is called *coordinator guided adaptor generation* that uses *coordinator*, a special adaptor to compose with given service and then perform pushdown model checking for the negation of required properties. The overview of adaptor generation in our approach is shown in Fig. 2. The detail of each step will be described below.

5.2 Coordinator

Coordinator has to be capable of receiving any message if the message is ready to be sent by a service, and sending any message that is in the stack head when there is a service ready to receive the message. We say that the behavior of the coordinator is *over-behavioral* and may imagine an one state IPS that has only self transitions pushing and popping all messages of services. Therefore, the behavior of the coordinator should cover all possible communications for coordinating given services, including both matching and mismatching behavior. In our approach, we build a coordinator for given services following the definition in Def. 5.

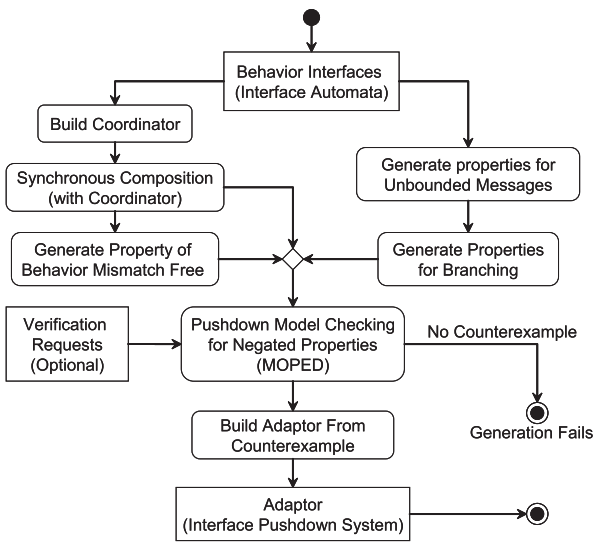


Fig. 2 Overview of adaptor generation.

Definition 5 (Coordinator): Given a set of composable IA4WS $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$. A coordinator is an IPS $C = (Q_C, q_C^0, \Gamma, z, T_C, F_C)$, where

$Q_C = \{q_C^0\}$ is the finite set of states has only the initial state q_C^0;
$A_C = \bigcup_i A_i^I = \bigcup_i A_i^O$ is the finite set of alphabets;
$\Gamma = A_C \cup \{z\} \cup \{\epsilon\}$ is the finite set of stack symbols;
z is the stack start symbol representing bottom of stack;
$F_C = Q_C$ is the finite set of final states.
$T_C = (Q_C \times A_C \times \Gamma) \times (Q_C \times \Gamma^*)$ is the set of transition relations defined as follows:

$(q_C^0, \gamma) \hookrightarrow (q_C^0, a\gamma)$ is a push transition,
$(q_C^0, a) \hookrightarrow (q_C^0, \epsilon)$ is a pop transition,
where $a \in A_C$, $\gamma \in \Gamma$ is the head symbol of stack.

5.3 Synchronous Composition with Coordinator

The system behavior under coordination of *coordinator* is computed by adapted synchronous composition defined in Def. 6. In the adapted synchronous composition, synchronization of sending and receiving transitions is between services and *coordinator*. As shown in Fig. 3, the resulted transitions are synchronized in two cases: output transitions in services are synchronized with push transitions in coordinator; input transitions in services are synchronized with pop transitions in coordinator. For internal transitions in services, *coordinator* remains same state and same stack content. Since *coordinator* is an IPS, the synchronous composition gives an IPS with only internal transitions. Therefore, we may omit the input and output transitions and treat the system behavior as a pushdown system.

Definition 6 (Adapted Synchronous Composition): Given a set of composable IA4WS: $P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$, with an adaptor $D = (Q_D, q_D^0, \Gamma, z, T_D, F_D)$. The adapted synchronous composition is an interface pushdown system $\Pi_i^D = (Q, q^0, \Gamma, z, T', F)$, where

$Q = Q_1 \times \ldots \times Q_i \times \ldots \times Q_n \times Q_D$: finite set of states.
$q_0 = (q_1^0, q_2^0, \ \ldots, q_n^0, q_D^0)$: initial state.
$T' \subseteq (Q \times \Gamma) \times (Q \times \Gamma^*)$: set of transition relations defined in Fig. 3.
$F = \{(q_1^f, \ \ldots, q_i^f, \ \ldots, q_n^f)\} \times F_D$: finite set of final states.

5.4 Property of Behavior Mismatch Free

The purpose of the property of behavior mismatch free is to define a LTL property representing the condition of deadlock free. According to the system behavior from adapted synchronous composition with *coordinator*, the property of behavior mismatch free is defined as in Def. 7 that all traces have to finally visit the final state of the system behavior along with the condition that the stack is empty. Note that since all given services have only one final state which goes to no other state and all states are final states in an adaptor, practically the system behavior can be treated as having only one final state which is the conjunction of properties that all services are in their final states.

$$
\begin{aligned}
T' = \{ & \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),\gamma) \hookrightarrow ((q_1,\ldots,q_i',\ldots,q_n,q_D'),a\gamma) | \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q_i',\ldots,q_n,q_D') \in Q \wedge \\
& (q_i,a,q_i') \in \Delta_i \wedge a \in A_i^O \wedge (q_D,\gamma) \hookrightarrow (q_D',a\gamma) \in T_D \wedge \gamma \in \Gamma\} \\
& \cup \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),a) \hookrightarrow ((q_1,\ldots,q_i',\ldots,q_n,q_D'),\epsilon) | \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q_i',\ldots,q_n,q_D') \in Q \wedge \\
& (q_i,a,q_i') \in \Delta_i \wedge a \in A_i^I \wedge (q_D,a) \hookrightarrow (q_D',\epsilon) \in T_D\} \\
& \cup \\
& \{((q_1,\ldots,q_i,\ldots,q_n,q_D),a,(q_1,\ldots,q_i',\ldots,q_n,q_D)) | \\
& (q_1,\ldots,q_i,\ldots,q_n,q_D),\ (q_1,\ldots,q_i',\ldots,q_n,q_D) \in Q \wedge \\
& (q_i,a,q_i') \in \Delta_i \wedge a \in A_i^H\} \\
\} &
\end{aligned}
$$

Fig. 3 Definition of transition relations in Def. 6.

Definition 7 (Adapted Behavior Mismatch Free): Given an IPS $S = (Q, q^0, \Gamma, z, T, F)$ which is the composition of a set of composable web services and their adaptor, the property of behavior mismatch free is written in a LTL formula $\Diamond p_{accept}$. p_{accept} is an atomic proposition and a labeling function for state s and stack head γ is defined as $L((s, \gamma)) : \{p_{accept} \mid s \in F \ \wedge \ \gamma = z\}$, where z is the start symbol of stack.

5.5 Fairness Property of Looped Transitions

Recall that the core idea of *unbounded messages* is mentioned in Sect. 3.2 with an incomplete formal definition, Though it is quite intuitive to give a complete formal definition starting from accepting runs of IPS, it goes beyond the scope of pushdown model checking for LTL properties to directly count number of occurrences and find out unbounded messages. Thus, we prefer building LTL properties for this requirement indirectly and skip the complete formal definitions of *unbounded messages* in this paper. Since a pushdown system (i.e., model of system behavior after adapted synchronous composition) has only finite number of states, A trace that has some messages occur arbitrary multiple times is therefore caused by loops. Therefore, locating unbounded messages can be considered the same task of finding out loops in behavior interfaces of services. Here we introduce the idea of locating strongly connected components (SCCs) by the Tarjan's algorithm [19]. Because all members of a SCC are able to reach each other through transitions among them, it is intuitive that loops are constructed from transition within every SCC of given services. We call these transitions *looped transitions* and the algorithm of locating them is shown in Algorithm 1. In this algorithm, the sub process is modified from Tarjan's algorithm by adding the part of building the set of *looped transitions* when a SCC

is found (line 16 to 18). Therefore, from the set of *looped transitions*, a fairness property for these transitions is generated. This property demands all *looped transitions* to be executed at least once so that loops are guaranteed in all accepting traces. Note that in the set of *looped transitions*, every input transition should have corresponding output transition(s) and vice versa. *Looped transitions* that do not have corresponding transitions exist should be deleted from the set. Finally, set of *unbounded messages* can be further determined by gathered messages in labels of looped transitions.

Algorithm 1: Locating Loop Involved Transitions

Input: an IA4WS $P = (Q, q^0, A^I, A^O, A^H, \Delta, q^f)$
Output: set of loop involved transitions: Δ^{loop}

```
1  Procedure SCC(q)
2    begin
3      Lowlink(q) := Number(q) := index := index + 1;
4      push_stack(D, q);
5      foreach (q, a, q') ∈ Δ do
6        if Number(q') is not defined then
7          SCC(q');
8          Lowlink(q) := min(Lowlink(q), Lowlink(q'));
9        else if Number(q') < Number(q) then
10         if on_stack(D, q') then
11           Lowlink(q) :=
             min(Lowlink(q), Number(q'));
12     if Lowlink(q) = Number(q) then
13       while q' = top_stack(D), Number(q') ≥ Number(q)
         do
14         pop_stack(D);
15         Q^scc ⟵ Q^scc ∪ {q'};
16       foreach δ = (q, a, q') ∈ Δ, a ∈ A^I ∪ A^O do
17         if q, q' ∈ Q^scc then
18           Δ^loop ⟵ Δ^loop ∪ {δ};
19       Q^scc ⟵ ∅;
20 Δ^loop ⟵ ∅; Q^scc ⟵ ∅;
21 empty_stack(D);
22 index := 0;
23 foreach q ∈ Q − {q^0, q^f} do
24   if Number(q) is not defined then
25     SCC(q);
26 return Δ^loop
```

5.6 Fairness Property of Branching Transitions

Though branchings in behavior interfaces of services are common, a counterexample generated by pushdown model checking in our approach can only go through one of execution traces of given services. Thus, if an adaptor for a specific branching is desired, fairness properties for corresponding transitions have to be specified to guarantee the generated counterexample goes through the desired branching. Furthermore, when there is a need of an adaptor that goes through exclusive branchings, fairness properties for exclusive branchings should be specified and pushdown model checking has to be performed several times to get all counterexamples for these exclusive branchings. Adaptors generated from these counterexamples are then composed to get an adaptor that supports exclusive branchings. Unfortunately, this process is quite tedious and has a complicate problem of grouping transitions corresponding to exclusive branchings.

To solve this problem, our approach provides another feature that generates an adaptor supporting exclusive branchings by performing pushdown model checking once. Recall the core idea of *services as one session process* mentioned in Sect. 3.2, the composition of services with coordinator is also an one session process and we may connect its final state with initial state by an epsilon transition $(q^f, z) \hookrightarrow (q_0, z)$ which only allows the composed service to start over when all services have all reached their final states. Thus, we can generate fairness properties for all branchings without worrying about grouping transitions and get a counterexample that goes through all exclusive branchings with an execution trace crossing multiple sessions. To locate corresponding transitions for branchings in a service, an intuitive and easy way is to find states having two or more transitions goes out. We call these outgoing transitions *branching transitions*. Automated search can be performed to locate all *branching transitions* and generated corresponding fairness properties. In our approach, fairness property of branching transitions is optional and can be manually specified or automatically generated by locating all branching transitions in behavior of services.

5.7 Pushdown Model Checking

This part performs pushdown model checking using MOPED [20] model checker which accepts pushdown system as input model. The checking target is the system behavior from adapted synchronous composition of services with *coordinator*. The property is the conjunction of behavior mismatch free and fairness of looped and branching transitions. Unlike usual model checking that returns a counterexample which violates specified property, here we perform model checking for the *negation* of the property so that a counterexample is an accepting execution trace that satisfies the property. The counterexample is then used to generate an adaptor.

5.8 Adaptor Generation

Algorithm 2 shows the algorithm for generating an adaptor from the counterexample obtained from pushdown model checking above. The input of the algorithm is a finite sequence of configurations which are pairs of states and stack contents. In this algorithm, we take adjacent two configurations and do the following actions: (1) put the two states into the set of states of the adaptor (line 8); (2) compare the two stack contents for building a transition connecting the two states (line 9 to 14). For the latter, since an adaptor only

has push, pop, and internal transitions, adjacent stack contents are differed by 1 or equal in length. If length of w_i is longer than of w_{i+1} by 1, the transition connecting s_i and s_{i+1} is a pop transition. If length of w_i is shorter than of w_{i+1} by 1, the transition connecting s_i and s_{i+1} is a push transition. If length of w_i is equal to w_{i+1}, the transition connecting s_i and s_{i+1} is an internal transition. Internal transition are actually corresponding to internal actions of given services that make no communication. If s_i and s_{i+1} are final and initial states respectively (line 13), the internal transition has to be neglected since this is a temporary internal transition added for dealing with branchings. Also, our algorithm generates internal transitions bidirectional (line 14) for the possibility of indetermination of internal transitions in given services. The generated adaptor is then ready to guide the system of services to avoid behavior mismatches and to meet the requirements on *unbounded messages* and *branchings*.

Algorithm 2: Counterexample to adaptor

Input: A set of composable IA4WS:
$P_i = (Q_i, q_i^0, A_i^I, A_i^O, A_i^H, \Delta_i, q_i^f)$, $i \in [1, n]$; Configurations $c_i = (s_i, w_i)$, $i \in [0, m]$; Loop start index k.
Output: Adaptor $D = (Q_D, q_D^0, \Gamma_D, z, T_D, F_D)$

1 $Q_D \longleftarrow \{s_k\}$; $q_D^0 \longleftarrow s_0$; $T \longleftarrow \emptyset$;
2 $\Gamma_D := \bigcup_i A_i^O \cup \{z\} \cup \{\epsilon\}$;
3 **foreach** $c_i = (s_i, w_i)$, $i \in [0, m]$ **do**
4 **if** $i = m$ **then**
5 $(s_i', w_i') = (s_k, w_k)$
6 **else**
7 $(s_i', w_i') = (s_{i+1}, w_{i+1})$
8 $Q_D \longleftarrow Q_D \cup \{s_i\}$;
9 **if** $|w_i| - |w_i'| = 1$ **then**
10 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', \epsilon)\}$;
11 **if** $|w_i| - |w_i'| = -1$ **then**
12 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', w_i'(0)w_i(0))\}$;
13 **if** $|w_i| - |w_i'| = 0 \wedge s_i' \neq q_D^0$ **then**
14 $T_D \longleftarrow T_D \cup \{(s_i, w_i(0)) \hookrightarrow (s_i', w_i'(0)),\ (s_i', w_i'(0)) \hookrightarrow (s_i, w_i(0))\}$;
15 $F_D \longleftarrow Q_D$;
16 **return** $D = (Q_D, q_D^0, \Gamma_D, z, T_D, F_D)$

6. Tool and Adaptor Implementation

This section demonstrates a prototype tool of our approach and discusses about implementing generated adaptor as BPEL processes. The issue of ordering of messages is also addressed here.

Prototype Tool: We have implemented a prototype tool to support the automation of service adaptation in our approach. This tool contains several programs developed in C without graphical user interface. The execution is cooperated with SPIN and MOPED model checkers and can be automated by script. The tool can read behavior interfaces of services defined in an input file and do the following tasks:

1. Perform compatibility check.
2. Output the Promela model along with the LTL formula of behavior mismatch free for SPIN.
3. Locate looped transitions in all services.
4. Compute adapted synchronous composition of services with coordinator and output the pushdown system model for MOPED.
5. Output the LTL formula of property of behavior mismatch free and fairness property of looped and branching transitions for MOPED.
6. After a counterexample is generated by MOPED, read the counterexample and generate an IPS, i.e. the adaptor of given services.

The input file specifies behavior interfaces of services by specifying the following information: a) initial state; b) final state; c) transitions having labels with prefix of "!" or "?" or no prefix. The task of compatibility check not only checks compatibility but also checks conditions about constraints for initial and final states of given services. The Promela model is for detection of behavior mismatches by SPIN and the pushdown system model is for MOPED to generate a counterexample for adaptor generation. As mentioned before in Sect. 3.3, the Promela model is coded as a system of distributed services communicating synchronously with each other. Note that since MOPED only accepts a pushdown system model as input, our tool has to compute the adapted synchronous composition and to output the composition as one pushdown system. If MOPED generates a counterexample, the tool then reads the counterexample and generates behavior interface of the adaptor of given services as a pushdown system.

It should be addressed that since MOPED does not provide functionality of labeling transitions [†], we have to make a little change in the pushdown system to define the fairness property of looped and branching transitions. For a push transition rule, for example `s1 <a> --> s2 <b a>` which pushes a symbol `b`, the transition is modified as two transitions: `s1 <a> --> s2 <push_b a>` and `s2 <push_b> --> s2 <b>`. Similarly, for a pop transition, for example `s1 <a> --> s2 <>` which pops a symbol `a`, the transition is modified as two transitions: `s1 <a> --> s1 <pop_a>` and `s1 <pop_a> --> s2 <>`. The two symbols `push_b` and `pop_a` are then specified as atomic propositions in the fairness property.

BPEL implementation: The adaptor generated by our approach can be further implemented as BPEL processes. Since an adaptor is represented as a pushdown system, the implementation should have both finite state machine part and stack part. It is difficult to implement both parts in a BPEL process but much easier to implement as two BPEL processes. Therefore, the states and transitions of sending/receiving messages are implemented in one BPEL process. The stack and stack operations (i.e., push and pop) are implemented in another BPEL process. Here we may call the two BPEL processes *BPEL-adaptor* and *BPEL-stack* for

[†]In MOPED, properties to be checked can only be described by connecting state and stack symbols with logical connectors

convenience. Though our tool does not yet support the automated generation of BPEL-adaptor and BPEL-stack, we have confirmed and succeeded in manual implementation with following directions using Netbeans IDE [22].

(1) Building BPEL-adaptor: This BPEL process captures the finite state machine part of adaptor. Note that BPEL is a description language for workflow so that all states are implemented as activities while transitions are implemented as service actions (invoke/receive). The activity after a receive action should push the received message into the stack by call the push action of the BPEL-stack process. The activity before an invoke action should prepare the message to be sent by calling the pop action of the BPEL-stack process. When building a BPEL process, all messages require corresponding port types for sending and receiving defined in WSDL [21]. Messages can be implemented as data types such as integer, string, or composite data type depending on signatures of given services.

(2) Building BPEL-stack: This BPEL process performs functionalities of a standard stack. The stack content can be defined as an array of user-defined message type and its property *MaxOccurs* has to be set to unbounded. Thus, the push action receives a message and add it into the array with increasing the length of array by one in the next activity. The implementation of pop transition is similar but length of array is decreased by one.

Ordering of Messages: Since our approach uses push-down system model, the use of stack makes the ordering of messages in the style of last-in-first-out (LIFO). Thus, messages sent multiple times such as *unbounded messages* have reversed ordering when being received. This might be unrealistic since services usually communicate in the style of first-in-first-out (FIFO), such as video on demand services providing streaming data. However, here we would like to point out that the ordering of *unbounded messages* in our approach can be maintained in implementation. For example, we can implement an adaptor with extra operations that adjust the ordering of *unbounded messages* in the stack to original ordering. We may also use another way of implementation such as building queues so that each queue corresponds to a specific messages and only store this message. As long as the implemented adaptor follows the behavior of the adaptor generated by our approach, it is irrelevant reverse or not the ordering of messages multiply sent in the implementation. Thus, we say that our approach can support both LIFO and FIFO communications practically.

7. Experiments and Discussions

This section demonstrates an experiment on an extended version of the motivational example and discusses with the results.

7.1 Experiments

In this experiment, we use the prototype tool to solve the motivational example shown in Fig. 1. To show the generality of our approach, the example is modified as shown in Fig. 4 which has exclusive branchings and signature mismatches. In order to apply our approach, signature mismatches have to be solved by specifying mappings of messages. Figure 5 shows two mappings of messages represented as two special services. Note that the two services are manually designed according to signature mismatches observed from Fig. 4. The service in the left merges information of trading order `Trade` and preferred price `Quote` from *Investor* into `Transac` to send to *Online Stock Broker*. The service in the right takes transaction result `Record` from *Online Stock Broker* and sends out `Log` to *Investor*. It also tells *Research Department* there is nothing to analysis by sending `NoRawData`. We call these two services *mapping services*. Mapping services are not necessary to meet the constraint of IA4WS: an IA4WS must have different initial and final states. Furthermore, this example has two exclusive branchings that *Investor* can decide either requesting analyzed data from *Research Department* or ask *Online Stock Broker* to make a transaction.

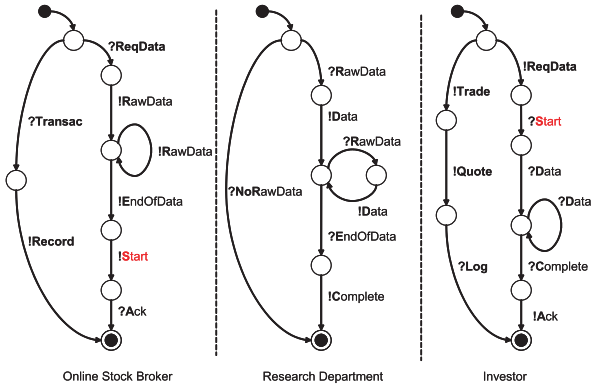


Fig. 4 Modified motivational example.

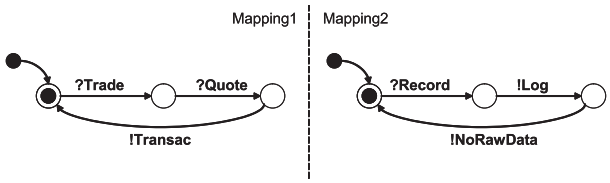


Fig. 5 Mapping services for message mapping.

Now we have five services including two mapping services and are ready to apply our approach. First, the behavior interfaces of the five services are encoded in an input file for the prototype tool. Following the steps described in Sect. 6, our tool checks compatibility of the services including mapping services. The signature mismatches are supposed to be solved by providing the two mapping services so the compatibility check should be passed. Then a Promela model is generated to perform detection of behavior mismatches by SPIN. The Promela model describes the five distributing services as finite state machines communicating synchronously, then SPIN reads the model and checks the reachability to the final state (i.e. property of behavior mis-

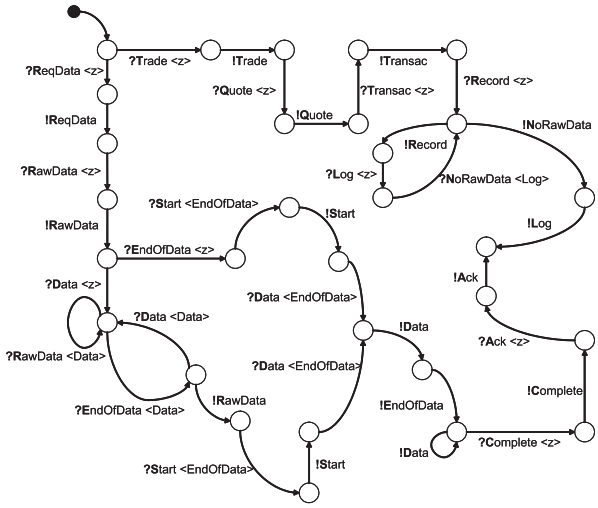


Fig. 6 Example: generated adaptor.

match free) of the system. As described in Sect. 3.1, there are behavior mismatches and the check fails. Therefore we proceed to adaptor generation.

According behavior interfaces of the five services, our tool generates a coordinator for the system and computes the adapted synchronous composition. A pushdown system is generated and output as the input of MOPED model checker. Note that as described in Sect. 6, the pushdown system is modified to support fairness property for transitions. MOPED then checks the system with properties of behavior mismatch free, unbounded messages, and branchings, then successfully returns a counterexample. The counterexample is then read by our tool and an adaptor is generated as shown in Fig. 6.

Compare to the expected adaptor in Sect. 3.1, the generated adaptor has a branching that has the following behavior: $?R\ !R\ ?D\ (?R\ !R\ ?D)^{n-1}\ ?R\ !R\ ?E\ ?S\ !S\ ?D\ !D\ !E\ !D^n\ ?C\ !C\ ?A\ !A$, $n > 1$. It is easy to confirm that in this behavior, all messages received by the adaptor are finally sent, which satisfies the basic requirement of an adaptor. Also, the part related to *unbounded messages*, which emphasized by n and $n-1$, is successfully captured in the behavior. It should be noticed that the behavior of the generated adaptor reflects the structures of behavior interfaces of the three services. Therefore, messages `RawData` and `Data` are not all packed in one group but separated just as corresponding transitions in the behavior interfaces. Also, the other branching of making a transaction is also correctly generated. Thus, we may conclude that our approach successfully generated an adaptor for the three services not only on the requirement of *unbounded messages* but also on the requirement of exclusive branchings as well as signature mismatches with mapping services provided.

The structure shown in Fig. 7 demonstrates the adapted services coordinated by the adaptor cooperating with mapping services. Note that the adaptation part includes the adaptor and mapping services. We have followed the directions of building a BPEL adaptor in Sect. 6 and built two

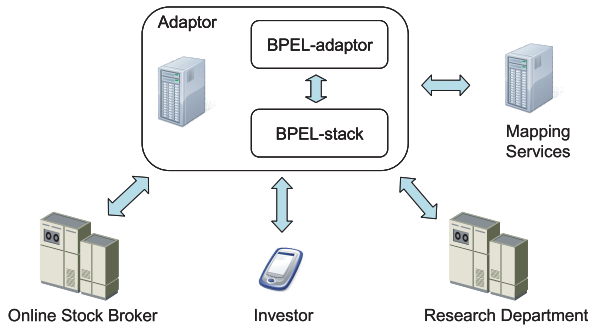


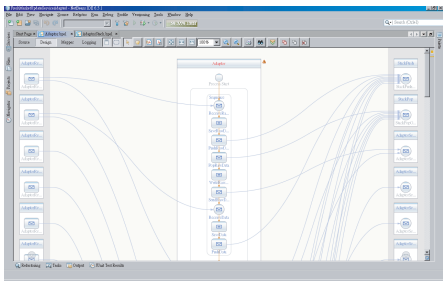
Fig. 7 Example: BPEL adaptor.

BPEL processes: BPEL-adaptor and BPEL-stack, which are partly shown in Fig. 8 (a) and Fig. 8 (b). We also implemented the five services as BPEL processes and constructed a composite application shown in Fig. 8 (c). The tests in Netbeans IDE were successful. The result shows that for given service protocols such as BPEL processes abstracted into finite state machines, our approach is capable of performing adaptor generation and finally implement a BPEL adaptor.

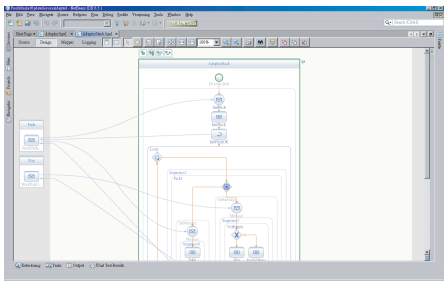
7.2 Discussions

The example shown in Fig. 4 extended from the motivational example shown in Fig. 1 is a basic ordering behavioral mismatching problem with unbounded messages. Though the conventional approach [3] is able to solve ordering behavioral mismatches, the use of LTS for representing adaptor contracts can not express unbounded messages. On the other hand, our approach uses pushdown system model in both expressing behavior interface of adaptor and computation. With the help of the stack, we are able to solve ordering behavioral mismatches with unbounded messages and the later are essential in dealing with non-regular behavior in service composition. Pushdown automata model is considered intuitive in software engineering and our approach that directly uses pushdown systems model in representing behavior interfaces is therefore applicable in service development.

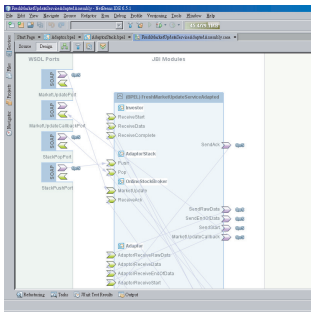
In the experiment in Sect. 7.1, our approach also performed automated adaptor generation with mapping services provided. This is only partly done in previous work [5] in an interactive way. Our basic idea is to figure out properties relating to structures of behavior interfaces then apply pushdown model checking to perform reachability search in the system. It is easy to imagine that to perform a thorough search on behavior interfaces of services, we need to define start and end points, then go through and check every transition. Thus, as described in Sect. 3.2, we use three kinds of properties: behavior mismatch free, unbounded messages, and branchings. These properties are concerning about structures of behavior interfaces in the following three aspects: start/end of service functionalities (behavior mismatch free), loops (unbounded messages), and branchings. Since model checking algorithms is designed to perform through search in a path, we only need to make sure every paths are going to be checked. Thus, we conclude



(a) Example: BPEL-adaptor process



(b) Example: BPEL-stack process



(c) Example: assemble and test

Fig. 8 Example: BPEL processes.

that fairness properties of looping and branching transitions guarantee all paths are exhaustively searched.

Furthermore, our approach provides some advantages in service oriented computing. Since automated adaptor generation in our approach is based on model checking, adaptation and verification are put together. When an adaptor is generated by our approach, the verification purpose, for example, safety check, is also fulfilled if properties for verification are specified. This saves much cost since processes of design/computation and verification are usually separated. The two processes may be even done using different models and two times or more efforts are needed. The characteristic of automated adaptor generation also makes binding services dynamically more easier. With support of our approach, developers only need to select services to be composed and specify necessary mappings. These two tasks can be further automated with support of other technologies such as semantic webs or context aware techniques. Furthermore, the use of pushdown model checking provides theoretical support of verifying pushdown system with stack having infinite length. This is much more better than using bounded model checking that sets the upper bound of the stack length since the upper bound may be set very large for generality and therefore causes state exploration.

To show the applicability to real applications, BPEL adaptor composed of two BPEL processes is built in the experiment following the directions described in Sect. 6. The behavior of BPEL adaptor is considered same as the behavior interface represented in pushdown system generated by our approach. This proofs the effectiveness of our approach on real service development. Furthermore, in service oriented computing, the inner behavior of a service is usually treated as a black box and only the revealed behavior interface is taken into computation. We argue that in the experiment, the size of states of behavior interfaces involved is close to real applications. The extended motivational example demonstrated in the experiment can be considered also a demonstration of adaptation on real applications.

8. Conclusion and Future Work

In this paper, we have proposed an approach for service adaptation. With only specifications of behavior interfaces of services and necessary messages mappings for signature mismatches, our approach automatically generates adaptors that supports non-regular behavior in service composition. The use of pushdown systems successfully captures the essences of adaptors that sent messages are guaranteed being received. The requirement of unbounded messages is especially addressed to characterize the non-regular behavior in service interactions and adaptors generated by our approach is promised to reflect this requirement. We argue that these can not be guaranteed or even concerned in conventional framework where adaptors are represented by finite state machines.

Furthermore, applying model checking technique makes service adaptation integrated with verifications so that an adaptor is verified at the same time being generated. This saves much time and cost than doing adaptation and verification separately. Finally, our experimental results showed that our approach is feasible to be applied on real-world web services like BPEL processes. Our approach also has generality of dealing with exclusive branchings and signature mismatches while mapping services are provided in advance.

For future directions, we are working on fulfilling the ability of the prototype tool with automated generation of BPEL adaptors to conduct more real-world experiments, especially with large-scale systems. We also plan to extend our approach on time constraints to tackle real-time issues on service adaptation.

References

[1] H.H. Lin, T. Aoki, and T. Katayama, “Automated adaptor generation for services based on pushdown model checking,” ECBS’11, pp.130–139, 2011.

[2] R. Mateescu, P. Poizat, and G. Salaün, “Adaptation of service protocols using process algebra and on-the-fly reduction techniques,” ICSOC’08, pp.84–99, 2008.

[3] C. Canal, P. Poizat, and G. Salaün, “Model-based adaptation of behavioral mismatching components,” IEEE Trans. Softw. Eng., vol.34, no.4, pp.546–563, 2008.

[4] E.M.C. Jr., O. Grumberg, and D.A. Peled, Model Checking, The MIT Press, 1999.

[5] J. Camara, J.A. Martin, G. Salaun, J. Cubo, M. Ouederni, C. Canal, and E. Pimentel, “ITACA: An integrated toolbox for the automatic composition and adaptation of web services,” ICSE’09, pp.627–630, 2009.

[6] J. Cámara, G. Salaün, and C. Canal, "Clint: A composition language interpreter (tool paper)," FASE'08/ETAPS'08, pp.423–427, 2008.
[7] R. Mateescu, P. Poizat, and G. Salaün, "Behavioral adaptation of component compositions based on process algebra encodings," ASE'07, pp.385–388, 2007.
[8] M. Tivoli and P. Inverardi, "Failure-free coordinators synthesis for component-based architectures," Sci. Comput. Program., vol.71, no.3, pp.181–212, 2008.
[9] J. Cubo, G. Salaün, C. Canal, E. Pimentel, and P. Poizat, "A model-based approach to the verification and adaptation of WF/.NET components," Electron. Notes Theor. Comput. Sci., vol.215, pp.39–55, 2008.
[10] H.R. Motahari Nezhad, B. Benatallah, A. Martens, F. Curbera, and F. Casati, "Semi-automated adaptation of service interactions," WWW '07, pp.993–1002, 2007.
[11] S. Mitra, R. Kumar, and S. Basu, "Automated choreographer synthesis for web services composition using i/o automata," ICWS'07, pp.364–371, 2007.
[12] H.H. Lin, T. Aoki, and T. Katayama, "Non-regular adaptation of services using model checking," ISORC'10, pp.170–174, 2010.
[13] J.A. Martín and E. Pimentel, "Automatic generation of adaptation contracts," Electron. Notes Theor. Comput. Sci., vol.229, no.2, pp.115–131, 2009.
[14] X. Fu, T. Bultan, and J. Su, "Conversation protocols: A formalism for specification and verification of reactive electronic services," Theor. Comput. Sci., vol.328, no.1-2, pp.19–37, 2004.
[15] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, BPEL4WS, Business Process Execution Language for Web Services Version 1.1. IBM, 2003.
[16] L. de Alfaro and T.A. Henzinger, "Interface automata," SIGSOFT Softw. Eng. Notes, vol.26, no.5, pp.109–120, 2001.
[17] J. Esparza and S. Schwoon, "A bdd-based model checker for recursive programs," CAV'01, pp.324–336, 2001.
[18] G. Holzmann, Spin model checker, the: primer and reference manual, Addison-Wesley Professional, 2003.
[19] R. Tarjan, "Depth-first search and linear graph algorithms," SIAM J. Comput., vol.1, no.2, pp.146–160, 1972.
[20] S. Schwoon, Model-Checking Pushdown Systems, Ph.D. Thesis, Technische Universität München, 2002.
[21] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web service definition language (wsdl)," Tech. Rep. NOTE-wsdl-20010315, World Wide Web Consortium, March 2001.
[22] Netbeans IDE, http://netbeans.org/

Hsin-Hung Lin received his B.S. and Ph.D degrees in Information Science from Japan Advanced Institute of Science and Technology (JAIST) in 2003 and 2011. He is currently a researcher of JAIST under supervision of associate professor Toshiaki Aoki. His research interests focus on applications of formal verification techniques, especially model checking, on software engineering.

Toshiaki Aoki is an associate professor, JAIST (Japan Advanced Institute of Science and Technology). He received B.S. degree from Science University of Tokyo (1994), M.S. and Ph.D. degrees from (1996, 1999). He was an associate at JAIST from 1999 to 2006, and a researcher of PRESTO/JST from 2001–2005. His research interests include formal methods, formal verification, theorem proving, model checking, objectoriented design/analysis, and embedded software.

Takuya Katayama received Ph.D degree in Electrical Engineering from Tokyo Institute of Technology in 1971. He has been a professor there in the Department of Computer Science until 1996. He was a professor in the school of Information Science at Japan Advanced Institute of Science and Technology from1991 to 2007 and is now president. He has been working on software process and its environment, software evolution and formal approach in software engineering.