IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 11, 2019, accepted April 4, 2019, date of publication April 11, 2019, date of current version April 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2910326

Toward Supporting Unplanned Dynamic Changes of Service-Based Business Processes

CHANG-AI SUN[1,2], **(Senior Member, IEEE), ZHEN WANG**[1], **ZAIXING ZHANG**[1], **PAN WANG**[1], **XIAO HE**[1], **AND JUN HAN**[3]

[1]School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China
[2]Science and Technology on Aerospace Intelligent Control Laboratory, Beijing 100854, China
[3]School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia

Corresponding author: Chang-Ai Sun (casun@ustb.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61872039 and Grant 61370061, in part by the Beijing Natural Science Foundation of China under Grant 4162040, in part by the Aeronautical Science Foundation of China under Grant 2016ZD74004, and in part by the Fundamental Research Funds for the Central Universities under Grant FRF-GF-17-B29. The work of J. Han was supported by the Australian Research Council under Grant LP150100892.

ABSTRACT Service-oriented architecture (SOA) has become an application development paradigm widely recognized both in academia and in industry. Although SOA-based applications may vertically implement business processes through the composition of loosely coupled services, they have to face frequent changes, such as unavailability of service (due to the uncontrollable, dynamic, and distributed environments) or dynamic replacement of service (due to specific user requirements). This indicates that the service compositions are expected to be adaptable enough to cater to such changing situations. In this paper, we propose an approach to supporting unplanned dynamic changes of service compositions by combining variability management and dynamic binding. The proposed approach introduces the concept of abstract proxy services in a variability-supporting service composition language, namely VxBPEL, and provides a mechanism to support variation design and dynamic binding for unplanned changes at run time. To realize the proposed approach, we have developed a service composition engine that supports abstract proxy services and their run-time replacement via service discovery or user intervention. Finally, a case study has been conducted to demonstrate the feasibility of the proposed approach and quantify the performance overhead resulting from runtime variability management and dynamic binding. The experimental results show that the proposed approach overcomes the limitation of imperative variability-based approaches in handling unplanned dynamic changes and further enhances the dynamic adaptability of VxBPEL-based service compositions.

INDEX TERMS Service oriented architectures, variability management, dynamic binding, adaptive service composition, business process execution language for Web services.

I. INTRODUCTION

Service-Oriented Architecture (SOA) [1] is a software development paradigm that is widely adopted both in academia and industry. Various organizations are increasingly providing their service following the Software-as-a-Service (SaaS) model [2]. A service is a software system designed to support interoperable machine-to-machine interactions over a network [3], which in most situations is isolated and only provides simple and basic functionality. Accordingly, these services are expected to be composed to meet complex business requirements. The process that coordinates a set of loosely coupled services (termed as *component services*) together is called *service composition* [4]. The component services may be implemented in different ways, such as SOAP Web Services [3], RESTful Web Services [5], and Apps [6]. The resulting composite service can act as a component service in another composite service of a larger granularity. As such, service compositions provide an efficient way for application development [7].

When a composite service is expected to provide continuous service, especially for a long-running business process, it must be adaptable sufficiently at run-time. (i) The operational environments of the component services are highly

The associate editor coordinating the review of this manuscript and approving it for publication was Resul Das.

2169-3536 © 2019 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

dynamic and unpredictable. The qualities of the component services in the composite service cannot be assured by the service providers. For instance, a Web service may temporarily fail or terminate, or its quality of service may suddenly deteriorate due to changes in network bandwidth. When one of the component services does not behave as expected, the composite service must be able to replace it with an alternative service. (ii) The business requirements may change frequently. The changes in the requirements need to be propagated to the component services dynamically. In order to quickly respond to all these changes, a composite service is expected to have adaptability [8].

Making a composite service adaptable at run-time becomes possible in the context of SOA due to the loosely coupled nature of the component services. On the one hand, SOA enables the vertical composition of services due to the convergence between business goals and technical services. For example, an operation of a Web service usually implements an independent business functionality. If the expected functionality changes at run-time, a replacement of the Web service deployed in a remote site can make sense. On the other hand, the current service composition techniques have not paid enough attention to the dynamic adaptability of service compositions. For instance, the Business Process Execution Language for Web Services (WS-BPEL) [9] is a widely recognized service composition language, which has significant limitations in terms of adaptability [10].

To address the adaptability of composite services, some research efforts are reported [11]. The key issue is how to handle the changes of the composite service. The existing approaches can be divided into two categories, namely, imperative and declarative [12]. The imperative method focuses on the procedure of constructing a process, and defining sequences of commands. For instance, Provop is a framework to capture all process variants in a single process model, which defines the optional elements to implement the changes of the basic process model [13]. The declarative method focuses on expressing the logic of a process without describing its control flow. For example, Pesic et al. [14] proposed an approach to support variability of workflow systems via constraint-based models. However, most of the existing approaches work at the instance level, i.e., the changes are considered on the basis of a specific service composition instance rather than using generic language abstractions. Therefore, the resulting service compositions suffer low understandability, maintainability, and evolvability [15].

In our previous work [10], [11], [15], [16], we have proposed to address the adaptability problem of composite services in terms of variability management. We have designed a variability-supporting service composition language, namely VxBPEL [10], which is an extension to the standard WS-BPEL [9]. With VxBPEL, one can treat the changes within the composite service as first-class objects. We have also developed an integrated supporting platform that enables the design, execution, deployment and run-time management of VxBEPL-based service compositions [17]. Unlike the existing approaches based on adaptation to process instances [18]–[20], our approach is a generic method based on a specification language. Accordingly, the resulting service compositions are easy to understand and maintain.

Although the previous approach is able to handle various changes within VxBPEL-based service compositions [11], it still has limitation in supporting *unplanned changes* at run-time, which is also shared by all other imperative methods. For instance, a predefined component service (as a variant) may become unavailable in a long running process. If this happens, the process variant defined using VxBPEL will consequently fail, since the current approach does not allow switching to a new alternative that may be discovered automatically or specified by the user, all at run-time.

To handle unplanned dynamic changes, we propose a new adaptive service composition approach in this paper. We introduce the concept of abstract proxy service in VxBPEL-based service compositions, and develop an enabling technique, including a service composition language and its supporting engine and tool. When an unplanned dynamic change occurs, our approach is able to find a substitute service, which could be newly discovered or specified by the user, to replace the original service at run-time, without terminating and restarting the whole application.

The main contributions of this work are as follows:

1) An adaptive service composition approach that enables unplanned dynamic changes of VxBPEL-based service compositions by combining variability management and dynamic binding. Our approach extends VxBPEL with the concept of *abstract proxy service* (*APS*) and provides a mechanism for integrating variability design and dynamic binding.
2) A process engine which was developed to support the extended VxBPEL by extending an open source WS-BPEL engine, namely, Apache ODE [21]. It supports the dynamic binding of Web services to process instances in two modes. One is done by run-time service discovery, and the other is done by the user. The dynamic binding is implemented based on the aspect-oriented programming technique.
3) A case study which was conducted to validate the proposed approach and quantify the performance overhead resulting from using the variability management and dynamic binding.

The rest of this paper is organized as follows. Section II presents our approach to supporting unplanned dynamic changes of service compositions. Section III discusses the design and implementation of the supporting processes engine. Section IV reports a case study that is used to evaluate the feasibility and performance of the proposed approach. Section V discusses related work and provides a comparison of them with our approach. Section VI concludes the paper and outlines future work.

II. ADAPTIVE SERVICE COMPOSITIONS SUPPORTING DYNAMIC UNPLANNED CHANGES

The standard WS-BPEL only provides basic constructs to describe interactions among activities in business processes. It is inadequate to accommodate frequent and rapid changes in service compositions. In this paper, we present a generic approach to supporting unplanned dynamic changes of service compositions through the combination of variability management and dynamic binding.

We first discuss the limitations of existing approaches, and then propose a new approach followed by the detailed discussions of its key issues.

A. LIMITATIONS OF EXISTING APPROACHES

Most of existing representative approaches are based on the proxy mechanism to support the adaptation of WS-BPEL service compositions. *TRAP/BPEL* [18] is a framework that aims to make an aggregate Web service continue to function even after one or more of its constituent Web services have failed. It monitors events such as faults and timeouts from within the adapted process. When faults occur, a generic proxy is used to replace the failed services with predefined or newly discovered alternatives. *wsBus* is another typical framework which is capable of realizing Quality of Service (QoS) adaptation of Web service compositions [19]. This framework introduces the concept of a virtual endpoint where a policy may be attached, and plays a role of broker which selects appropriate services for execution at run-time. Similarly, Ardagna and Pernici [20] proposed to select Web services to satisfy the predefined QoS constraints by using the linear planning technique. All these typical proxy-based approaches achieve adaptation of service compositions at the implementation layer and are for individual process instances at run-time.

The other category of adaptive service composition approaches is based on variability management. In our previous work, we investigated how to systematically deal with the adaptation issue of service composition from the perspective of variability management [10], [11], [15]–[17]. The proposed approach treats changes of service composition as first class citizens, and provides a systematic treatment for such changes, including a framework and a supporting platform [17]. The former includes a variability-supporting service composition language (i.e. VxBPEL), a variability modelling profile and execution process [11], while the latter includes an VxBPEL analysis tool (i.e. ValySec [16]), two versions of VxBPEL engine (i.e. VxBPEL_engine [22] and VxBPEL_ODE [23]), a visual VxBPEL designer tool (i.e. VxBPEL_Designer [17]), and a run-time VxBPEL management tool (i.e. MX4B [17]). This approach considers the adaptation issue of service composition at the specification layer, providing a language and its corresponding execution environment.

One may observe that both the above-mentioned categories of adaptive service composition approaches have their limitations. (i) The proxy-based approaches have such disadvantages as the resulting service compositions being difficult to maintain (due to changes being explicitly treated), and having low efficiency (once some changes happen, both the service composition and proxy have to be rewritten). (ii) The variability management-based approaches are not able to handle unplanned run-time changes since variants have to be predefined at the design phase, although it has such advantages as easy to understand and maintain. In this context, a question arises as follows: *is it possible to combine the two categories of approaches to complement each other?*

In this work, we introduce the proxy mechanism into the variability management-based adaptive service composition approach. The goal is to address the limitation of the VxBPEL-based service compositions in supporting unplanned dynamic changes. The proposed approach and the supporting platform is reported below.

B. OVERVIEW OF APPROACH

The proposed approach is illustrated in Fig. 1. The basic idea behind our approach is to introduce the proxy mechanism into variability management-based service compositions. An *abstract proxy service* (*APS*) is an abstract service at the design time, and has to be a concrete one at run-time. We introduce the concept of *abstract proxy services* (APS) into VxBPEL since the predefined variants at the design time are unable to cover all changes that may happen at run-time. Accordingly, the binding of an *APS* with a concrete service can be realized through either run-time service discovery or a request from the user. To enable the proposed approach, we extend VxBPEL with a new language construct `dybind` to support the declaration of *APS*, and develop a support platform to interpret *APS* and realize automatic run-time service discovery and user-requested service replacement.

We discuss below how the proposed approach works and its related key issues.

C. DECLARING APS

For the given application requirements, one can construct a service-based process by coordinating a set of functional services that are likely to be deployed and run on remote sites. This construction process is often known as service composition. To make service composition adaptive, one method is to introduce variability management [24]. As a representative method, we have proposed to design adaptive service compositions using VxBPEL [17]. VxBPEL supports the main concepts of variability design [10], including *variation points* and *variants*. A *variation point* declares a variable part in the service composition, which may contain multiple variants. A *variant* defines an alternative implementation within a variation point.

In VxBPEL, the syntax of `variant` is as follows:

```
<vxbpel:Variant name=''default''>
(WS-BPEL code or VariationPoint elements)
</vxbpel:Variant>
```

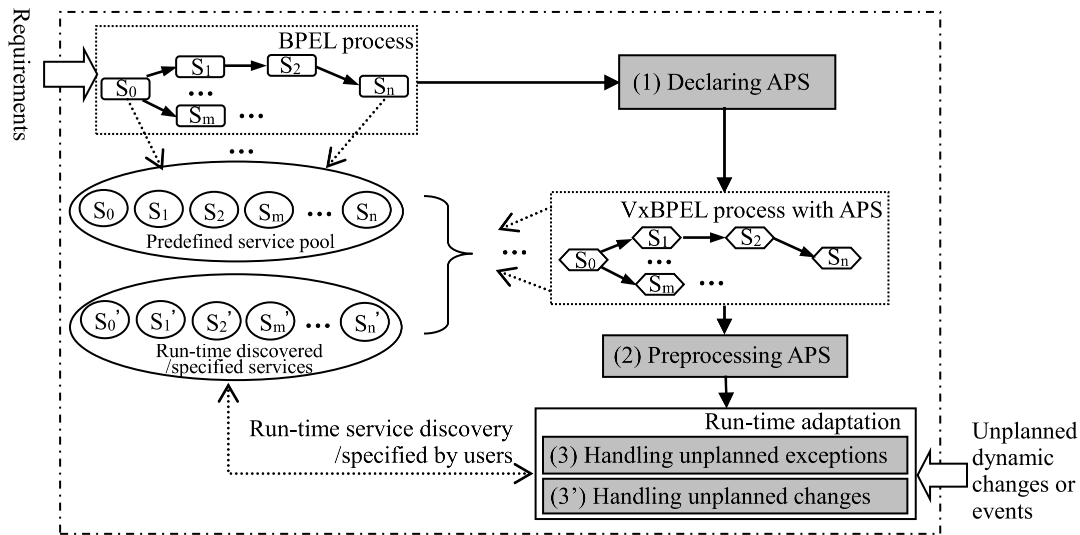

FIGURE 1. **Overview of approach.**

Within the `Variant` element, any valid WS-BPEL code can be inserted.

The syntax of `VariationPoint` is defined as follows:

```
<vxbpel:VariationPoint name=``...''>
<vxbpel:Variants>
<vxbpel:Variant name=``v1''>
/*...definition of variant v1...*/
</vxbpel:Variant>
<vxbpel:Variant name=``v2''>
/*...definition of variant v2...*/
</vxbpel:Variant>
...
</vxbpel:Variants>
</vxbpel:VariationPoint>
```

Although VxBPEL enhances WS-BPEL by providing a set of variability design constructs that can be used to specify the predefined variants within service compositions at design time, it is impossible to predict all possible changes that may happen at run-time. Therefore, VxBPEL is still not able to deal with unplanned changes at run-time.

To overcome this limitation, we further declare an *APS* as a variant of the variation point through the `dybind` construct. The syntax of `dybind` is as follows:

```
<vxbpel:dybind selector="..."/>
```

An *APS* must be declared within an `invoke` element. It indicates that the target service to be invoked can only be determined at run-time. As an illustration, the following VxBPEL code means that the target service (associated with the partner link *FoodPL*) will be finalized at run-time.

```
<bpel:invoke name="InvokeFood"
partnerLink="FoodPL"
operation="findFood"
inputVariable="FoodPLRequest"
outputVariable="FoodPLResponse">
<vxbpel:dybind selector="..."/>
</bpel:invoke>
```

Furthermore, we identify two classes of unplanned dynamic changes in service compositions. One is related to environment changes (*EC* in short), such as corruption or unavailability of a service being invoked by a service composition (i.e., due to maintenance or bad network conditions); the other is related to requirements changes (*RC*, in short), such as the functionality or Quality of Service of the current service being invoked is not satisfactory. The `dybind` construct with different `selector` attributes are provided to distinguish these two classes of unplanned changes. That is, the `selector` attribute having value "mkchange" means that the *APS* is used to respond to *RC*; otherwise, the `selector` attribute having any other value means that the *APS* is used to respond to *EC*, such as one service being invoked becomes unavailable. For both cases, we further provide interfaces for binding a concrete service to the *APS*.

D. PREPROCESSING APS

After the above treatment at the design time, we can now have VxBPEL-based service compositions with *APS* declarations. The resulting service compositions contain not only variability constructs such as variation points and variants, but also the `dybind` constructs for *APS*. Preprocessing the VxBPEL-based service compositions with *APS* declarations mainly include parsing the VxBPEL specification, creating an object model for the execution, and serializing all relevant objects in a binary file.

During the parsing phase, it generates objects for standard BPEL elements, and records the information about

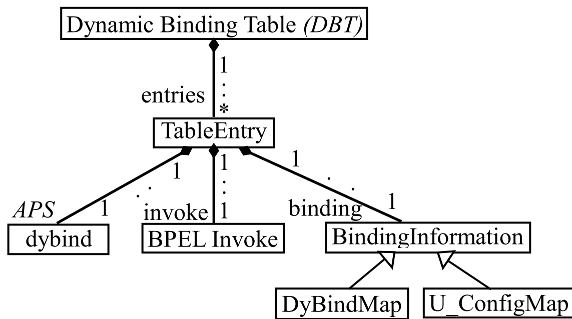

FIGURE 2. Conceptual model of dynamic binding table.

variation definitions and configurations. To handle the activities associated with *APSs* (namely, those containing the `dybind` elements), we need to create a *Dynamic Binding Table* (*DBT*) for each activity. A *DBT* contains a set of table entries, each of which corresponds to an *APS* and further captures its owner `invoke` element and service binding information. For dynamic service discovery, the binding information is stored in `DyBindMap`, while for the dynamic service replacement required by the user, the binding information is stored in `U_ConfigMap`. The conceptual model of the *DBT* is illustrated in Fig. 2. At the end of this phase, the expected resources and variability configuration schema are ready for execution.

E. RUN-TIME ADAPTATION

A normal VxBPEL-based service composition only contains a variety of variability configuration schema, and each schema specifies which variants should be selected for each variation point. The services to be invoked in all variants are predefined and thus they are not able to be changed at runtime. To execute VxBPEL-based service compositions with *APS* declarations, we have to extend the VxBPEL engine to interpret the new construct `dybind` related to *APS* and support the dynamic binding of *APS* to concrete services identified.

To implement the unplanned dynamic changes mentioned in Section II-C, the *APS* can be triggered in two modes. For *EC*, the *APS* is triggered in an automatic way when an exception happens with the owner `invoke` element (e.g., the target service to be invoked become unavailable), and a newly discovered service will be used for run-time service replacement of the *APS*; for *RC*, the *APS* is triggered in a manual way and a target service explicitly specified by users will be used for run-time service replacement of the *APS*. The binding process of *APS* is illustrated in Fig. 3. When the VxBPEL engine encounters an exception, it first decides whether the exception is related to run-time service replacement or not (1). If so, it then retrieves relevant information about the target service, such as port, service name, service type, which is either provided via service discovery (2) or specified by the user (2'). The target service is then bound into the business process (3 or 3'), and the engine continues the execution of the remaining process (4).

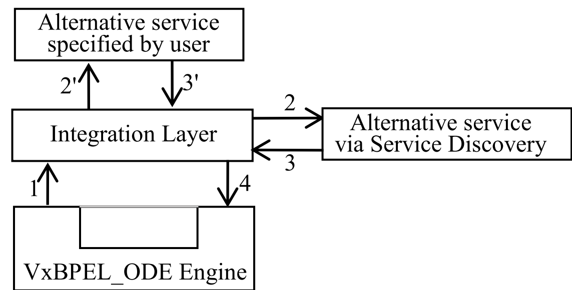

FIGURE 3. Concrete realization of *APS* within the *dybind* element.

For both modes of run-time service replacement, we further develop an integration tool which provides an infrastructure to enable both run-time service discovery and service replacement required by the user. More details about the extended VxBPEL engine and the integration tool will be discussed in Section III.

1) HANDLING UNPLANNED EXCEPTIONS

When a VxBPEL-based service composition is being executed, our approach will monitor every external service invocation starting from an `invoke` element. If an exception happens (for example, due to network failure), the extended engine (i.e., the VxBPEL_ODE engine in Fig. 3) will intercept it, and search the *APS* that is associated with the `invoke` element in the *DBT*. If such an entry does not exist, the extended engine will re-throw the exception, so that the original engine (i.e., the WS-BPEL ODE engine) can handle it. If such an entry exists in the *DBT* and the `selector` attribute is not "mkchange", the *Integration Layer* in the extended engine will trigger the *Service Discovery* process to find a new alternative service; otherwise, the process handling unplanned changes of requirements will be triggered (see the next subsection).

The *Service Discovery* process (i.e., the Alternative service via Service Discovery in Fig. 3) will first retrieve the information of the *APS* from its owner `invoke` element and the corresponding partner link definition. Then, it will query alternative services with the interface information from a UDDI proxy. If the UDDI proxy returns some candidate services and one of them will be selected as the alternative candidate service (either by default or by some selection condition).

Next, the original target service will be replaced with the selected alternative service. The extended engine will update the execution context of the invoked service, where the service binding information is stored, and then resend the service invocation message to the new target service. It is worth noting that there may be a mismatch between the old message and the new one. For instance, assume that the old service that becomes unavailable has the following information:

```
targetNamespace: http://food.five.
ustb.org;
Web service name: FoodSuggestService;
```

```
operation name: findFood;
port name: FoodSuggest;
port location: http://localhost:8080/...
/services/FoodSuggest.
```

while the alternative service includes the following information:

```
targetNamespace: http://foodc.five.
ustb.org;
Web service name: FoodCSuggestService;
operation name:  findFoodC;
port name: FoodCSuggest;
port location:http://localhost:8080/...
/services/FoodCSuggest.
```

The differences between the old and new messages may appear in targetNamespace, operation name, port location, and required parameters. In order to ensure that the alternative service can successfully receive an invocation message, the old invocation message is to be converted into the new one, which has been automated by means of parsing the WSDL file of the involved Web services. As such, the business process can run continuously during dynamic service replacement; otherwise, it has to be paused and then restarted. Note that Web services concerned in our approach are stateless, and repeated invocations of the same service are thus independent. Furthermore, it is expected to rollback the previous operations in case of an exception, which is not supported in the current treatment.

2) HANDLING UNPLANNED CHANGES OF REQUIREMENTS

Due to the dynamic changes in user requirements, a component service previously bound may need to be replaced by another after the VxBPEL-based service composition is deployed. Let us consider a travel agency application comprising of two component services, which are expected to provide accommodation and food services, respectively. The original requirements of the application are set to cheap hotels and fast-food restaurants. However, after the composition is deployed, the requirements evolve into cheap hotels and best restaurants. In this situation, the old service that is expected to provide the fast-food service has to be replaced by a new one that provides the best food service. In order to cater for such unplanned dynamic requirement changes, the process should be updated online without any interruption. Our approach supports this by the user actively triggering the *APS* through a management interface.

The treatment is similar to that for handling unplanned exceptions except that the alternative service is specified by the user. As discussed in Section II-E.1, our approach intercepts each external invocation to the `invoke` elements. when an invocation of an *APS* is encountered, the extended engine will look up the *DBT*. If this *APS* is associated with an `invoke` element in the *DBT* and its `selector` attribute is "mkchange", a process is trigged to expose a management interface to the user (i.e. Alternative service via user in Fig. 3), who is responsible for specifying a new alternative service (otherwise, a *Service Discovery* process will be triggered). After an alternative service is specified, our approach will forward the invocation message to the new target service and a similar message conversion is applied.

F. SUMMARY

We have presented a language-based generic approach to address unplanned dynamic changes in service compositions. VxBPEL, which supports variability in WS-BPEL service compositions, is further extended with a new construct, `dybind`. One can employ the extended VxBPEL to declare an *abstract service proxy* in service compositions. Such an *APS* introduces the opportunity of handling unplanned run-time changes of the resulting compositions, which has been extensively discussed above. To summarize, the proposed approach provides a systematic way to handle unplanned dynamic changes, including those related to environments and those related to requirements.

III. IMPLEMENTATION

In this section, we report on the implementation of an extended VxBPEL engine called VxBPEL_Dyn_ODE, which provides an enabling infrastructure for preprocessing the *APS* in the extended VxBPEL and supporting run-time adaptation via *APS*.

A. ARCHITECTURE OF VXBPEL_DYN_ODE

The implementation of VxBPEL_Dyn_ODE takes a two-phase interpreter style, which first compiles VxBPEL process definitions into a standard process object pool and then individually interprets the compiled objects at run-time. The run-time interpretation is further refined into three parts: core interpretation activities via an *ODE BPEL Runtime*, invocation of external Web services via an *ODE Integration Layer*, and dynamic service searching & binding of *APS* via a *Dynamic Binding Module*. The architecture of VxBPEL_Dyn_ODE is shown in Fig 4. VxBPEL_Dyn_ODE is developed by extending VxBPEL_ODE, a VxBPEL engine that was developed in our previous work [23]. Compared with VxBPEL_ODE, the *Dynamic Bind Module* is newly added and the *VxBPEL Compiler* is extended with the *preprocessing of APS*.

We first introduce each component of VxBPEL_Dyn_ODE individually and then discuss how these components interact, especially focusing on the interaction between *Dynamic Bind Module* and *ODE Integration Layer*.

- *Compilation* consists of the *ODE BPEL Compiler*, *VxBPEL Compiler*, and *Configuration Management*. The *ODE BPEL Compiler* is responsible for converting standard BPEL elements into a compiled representation. The *VxBPEL Compiler* is responsible for the compilation of VxBPEL-specific elements (such as *variants*, *variation points*, *APSs*, and so on). It first creates an object model for all VxBPEL elements similar in

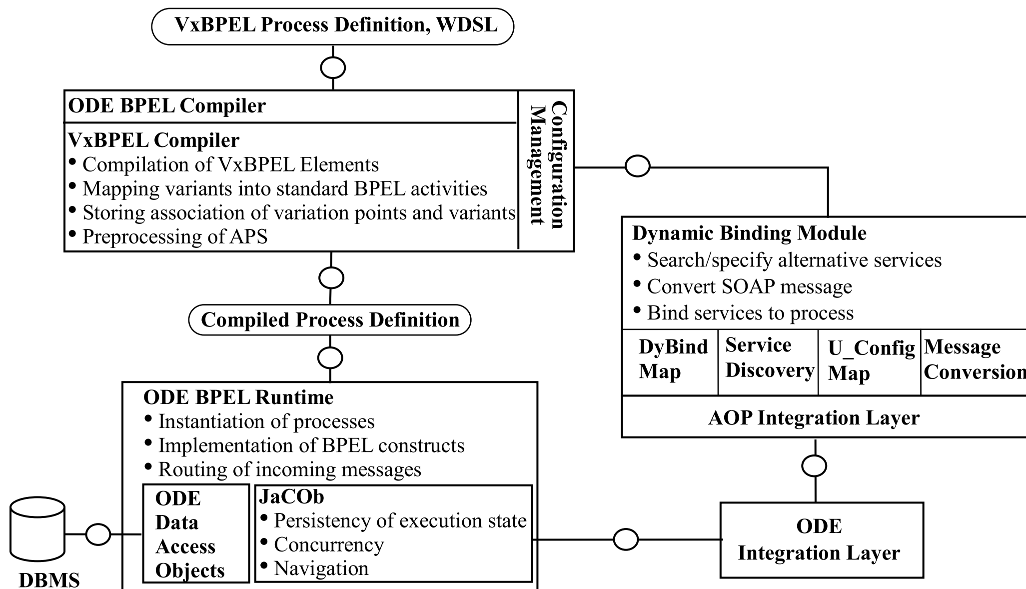

FIGURE 4. Architecture of VxBPEL_Dyn_ODE.

structure to the object model of BPEL elements, then maps all variants of a variation point to BPEL activities, and finally stores the association of variation points and variants. The *Configuration Management* manages variants associated with a variation point and maintains a current variation configuration. After the compilation, the generated objects contain process and variation attributes. The *Configuration Management* selects the specified variants based on a variation configuration scheme and the variant selection process repeats until all variation points are processed. After this process, only BPEL process objects remain because variants associated with a variation point become standard BPEL elements. Finally, all process relevant objects are serialized into a binary file.

– *ODE BPEL Runtime* is used to interpret the compiled process definitions, including creating a new process instance, implementing the various BPEL constructs, and delivering an incoming message to the appropriate process instance. Inside the *ODE BPEL Runtime*, the *ODE Data Access Objects (ODE DAOs)* mediates the interaction between the *ODE BPEL Runtime* and *DBMS*, and *JACOB* provides an application concurrency mechanism, including a transparent treatment of process interrupts and persistence of execution state.
– *DBMS* is an underlying database management system which stores information about active process instances, routing messages, values of BPEL variables for each instance, values of BPEL partner links for each instance, and process execution states.
– *ODE Integration Layer* provides an execution environment, including communication channels to interact with Web services or *AOP Integration Layer*, thread scheduling mechanisms, and the lifecycle management for the *ODE BPEL Runtime*.
– *Dynamic Binding Module* is responsible for communication with the *ODE Integration Layer*. It monitors all invocations related to the *APS* from the *ODE Integration Layer*, binds an alternative service returned via run-time service discovery or specified by the user, converts the invocation and response messages for the alternative service, and returns the control back to the *ODE Integration Layer*.

Fig. 5 depicts the main workflow of VxBPEL_Dyn_ODE. After the VxBPEL-based process is compiled, the *ODE BPEL Runtime* will take over the interpretation of the compiled process definition. When an *invoke* activity is encountered (Step 1), the *ODE BPEL Runtime* extracts the execution context from the *ODE DAOs* for the current object (Step 2) and transfer the invocation of an external Web service to the *ODE Integration Layer* (Step 3), which will be in charge of the communication with the *Dynamic Binding Module*. The *AOP Integration Layer* within the *Dynamic Binding Module* is responsible for catching the invocation exceptions (Step 4) and mediating the subsequent service run-time replacement flows. The *DyBindMap* checks whether there is an entry related to the *invoke* object in the *DBT* (Step 5) and returns the query result (Step 6). If the *invoke* object is not associated with an *APS*, the *AOP Integration Layer* will re-throw the exception. Otherwise, if the query result indicates that the `selector` is not set to "mkchange" (Step 7), the *DyBindMap* will invoke the *UDDIProxy* to search alternative services (Step 8 $\sim$ Step 9), create the execution context for the returned alternative service (Step 10), invoke the *Message Conversion* to convert the invocation and response SOAP messages for the alternative service (Step 11), and

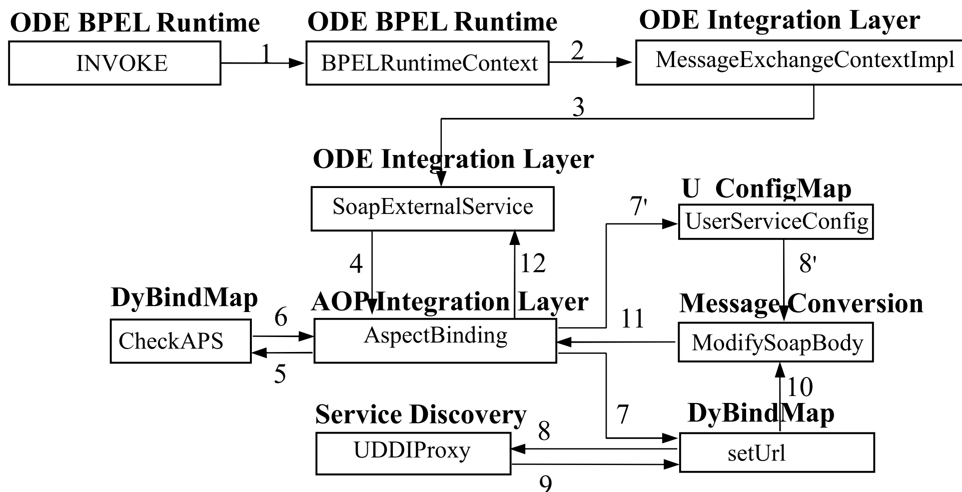

FIGURE 5. Interactions between *Dynamic Binding Module* and *ODE Integration Layer*.

return the control back to *ODE Integration Layer* (Step 12); if the `selector` is set to "mkchange", the *U_ConfigMap* will specify an alternative service and create the execution context for the alternative service (Step 7' ∼ Step 8'), invoke the *Message Conversion* to convert the invocation and response SOAP messages for the alternative service (Step 11), and return the control back to *ODE Integration Layer* (Step 12).

B. IMPLEMENTATION OF THE DYNAMIC BINDING MODULE

In order to enable run-time adaptation, we have to extend the previous VxBPEL engine, VxBPEL_ODE, with the capability of replacing a service with a newly discovered or specified one at run-time. As we discussed above, the *Dynamic Binding Module* is a newly added component which is mainly responsible for run-time service replacement. One key issue is how to effectively integrate the *Dynamic Binding Module* with the *ODE Integration Layer*. Furthermore, such an integration should demand as little modifications to the existing *ODE Integration Layer* as possible. Fortunately, AOP provides a perfect technical solution to cater for this need [25], and we accordingly adopt AspectJ [26], a popular aspect programming framework, to develop the *AOP Integration Layer*, which is the core component of *Dynamic Binding Module*.

When implementing the *AOP Integration Layer*, we design an *aspect* called *AspectBinding*, which contains a set of pointcuts and advices. Next, we explain individually how each pointcut and its associated advice (and its involved components) are implemented.

- Pointcut *invoke* is specified at *invoke ( )* to intercept a run-time exception when invoking an external partner, and the *invoke (x)* advice associated with this pointcut first checks whether the object that triggers the invocation exception has an entry in the *DBT* (i.e., CheckAPS in *DyBindMap*), and then decides to execute which kind of run-time service replacement: (i) If this object belongs to *DBT* and its associated `selector` attribute is not equal to "mkchange", the advice calls UDDIProxy in the *Service Discovery* to locate an alternative service. The implementation of UDDIProxy is based on UDDI4J,[1] which is a Java class library that provides an API to interact with a UDDI (Universal Description, Discovery and Integration) registry;
(ii) If this object belongs to *DBT* and its associated `selector` attribute is equal to "mkchange", the advice calls a JMX interface exposed by the *Configuration Management* for specifying an alternative service (*U_ConfigMap*). JMX is an extension to Java, allowing Java objects to expose certain functionality (possibly to external tools) [27];
(iii) Otherwise, this object does not belong to the *DBT*, and the advice re-throws the exception via "*proceed (x)*", which will be handled by the original *ODE BPEL Runtime*.
- Pointcut *setEPR* is specified at the invocation of *setUrl ()* within *invoke ()* to intercept the binding address, and its associated advice, *setEPR (url)*, and sets the binding address (i.e., *url*) of the alternative service that is specified via the *U_ConfigMap* or discovered via the *Service Discovery*, as discussed in the *invoke (x)* advice.
- Pointcut *createSoapRequest* is specified at the invocation of *createSoapRequest ()* within *invoke ()* to intercept its SOAP request (including the message context *mctx*, message *msg*, and operation name *opr*). The associated advice, *createSoapRequest (mctx, msg, opr)*, is responsible for the conversion of the SOAP request. In order to do that, this advice first derives the requested operation, name space, and relevant parameters from the intercepted request message *msg* through WSDL4J,[2] and then creates a new SOAP request for the alternative

[1] http://uddi4j.sourceforge.net/
[2] https://sourceforge.net/projects/wsdl4j/

service that is specified via the *U_ConfigMap* or discovered via the *Service Discovery*.
- Pointcut *reply* is specified at the invocation of *reply ()* within *invoke ()* to intercept its response message, and its associated advice, *extract (message, soapBody, bodyDef, msg, rpcWrapper)*, the converts the response message of the alternative service into the required form.

From the above discussion, one can observe that the presented implementation only introduces minor modifications to the *ODE Integration Layer* in order to support run-time service replacement. In particular, we only add a package to implement the *AspectBinding* aspect inside *org.apache.ode.axis2*, which is responsible for the implementation of the *ODE Integration Layer*. For *AspectBinding*, we specify a total of four pointcuts, and for each pointcut, one advice is specified. Among the four pointcuts, the first one intercepts the exception (i.e., AxisFault) thrown by the invocation of *invoke ()* inside *ODE Integration Layer* when invoking an external Web service, the other ones intercept necessary information for service binding and message conversion during the procedure of *invoke ()*, including binding address, request message, and response message. The advices associated with these pointcuts execute the actions following the interactions shown in Fig. 5.

IV. EVALUATION

We have conducted a case study to demonstrate the effectiveness of the proposed approach and quantify its performance overhead. We have used the *Travel Reservation System* [28] as our subject program, and all Web services in the system are implemented in Java. We first briefly describe the *Travel Reservation System* and the research questions, then evaluate the effectiveness and performance of our approach in handling unplanned exceptions and changes in user requirements, followed by a discuss of the answers to the research questions.

A. SUBJECT PROGRAM AND RESEARCH QUESTIONS

The *Travel Reservation System* [28] provides a series of travel services, including *Vehicle Reservation*, *Hotel Reservation*, *Ticket Reservation*, and *Insurance Purchase*. Through this system, customers are able to book tickets for their preferred transportation, and reserve hotel and tour tickets according to their travel plans. When doing these, they may also purchase insurance.

Following the main steps of the proposed approach in Section II, we have implemented a *Travel Reservation System* through VxBPEL-based service compositions. Note that there are likely multiple alternative services available for each type of service in the real business scenarios. For instance, a number of hotels may provide their services for *Hotel Reservation*. For simplicity of the demonstration, we only consider two variants (i.e., alternative services or business fragments) at each variation point during the implementation. Accordingly, Table 1 illustrates the variability configuration of the *Travel Reservation System*. Note that we assume the *Travel Reservation System* can provide customers with either *Flight* or *Train* reservation, either *Business* or *Economy* Hotel reservation, either *Scenic* or *Exploration* reservation, and either *Insurance A* or *Insurance B* service.

TABLE 1. Variability Configuration of *Travel Reservation System*.

Variation Point	Variant 1	Variant 2
Vehicle Reservation	Flight	Train
Hotel Reservation	Business	Economy
Ticket Reservation	Scenic	Exploration
Insurance Purchase	Insurance A	Insurance B

Now, let us consider the following common scenarios that could happen at run-time and the corresponding research questions for evaluation.

Scenario 1: After the system is deployed, a component service may become unavailable due to various environment-related exceptions. For instance, *Business* associated with *Hotel Reservation* becomes unavailable, and then any invocation of this service will result in a run-time exception. For this case, the system should find an alternative service that can provide the required accommodation service, say, *Chain Hotel*, to meet the customer's requirements in a timely manner. In order to do that, one normally has to re-design the system to replace *Business* with *Chain Hotel*, and re-deploy the system. Such a maintenance process usually takes some time and design efforts, which usually does not meet the user's expectations. In this regard, we are interested in evaluating whether our approach can effectively handle this case in a more effective manner, and one research question arises: *"Can the proposed approach support the run-time service replacement when a predefined service becomes unavailable due to environmental changes?"* ***(RQ1)***

Scenario 2: User requirements may also dynamically change after the system is deployed, and the system should be adaptable to these changes. For instance, a customer is not satisfied with the insurance services currently provided by the system, instead, she would like to specify her preferred insurance service through run-time search of a newly available insurance service, say, *Insurance C*. Again, to cater for the dynamic changes of user requirements, the system has to undergo a maintenance process, which includes the binding of the *Insurance Purchase* service with *Insurance C* and redeployment of the system. In this case, we are interested in evaluating whether our approach is able to implement the dynamic changes related to user requirements in a more effective manner, and the research question is as follows: *"Can the proposed approach support a newly discovered service specified by the user at run-time without redesign and redeployment of the system?"* ***(RQ2)***

If the unplanned dynamic changes related to either the environment or user requirements can be effectively handled by our approach, we are further interested in the performance overhead due to the dynamic binding introduced in our approach compared with the original VxBPEL process.

```
<vxbpel:Variant name="Business_Hotel">
  <bpel:invoke name="Invoke_Business_Hotel" partnerLink="Business_HotelPL"
    operation="provideHotel" inputVariable="Business_HotelPLRequest"
    outputVariable="Business_HotelPLResponse">
  <vxbpel:dybind selector="Yes"/></bpel:invoke>
</vxbpel:Variant>
```

FIGURE 6. VxBPEL fragment of *Business* invocation.

```
1  Service Invocation Exception!
2  14:49:12,640 ERROR [ExternalService] Error sending message <…provideHotel…>:
3  Transport error:404 Error:Not Found
4  org.apache.axis2.AxisFault:Transport error: 404 Error: Not Found
5                at org...
6  Start Dynamic Binding Mechanism!
7  Replace the Original Operation: provideHotel!
8  Reorchestration Process!
9  Chain_Hotel!
10 Start Modify URL!
11 URL: http://localhost:8080/TravelReservation/services/Chain_Hotel?wsdl!
12 Create Request Information of New Web Service: provideHotel!
13 <parameters><…></paramters>
14 End Dynamic Binding Mechanism!
```

FIGURE 7. An execution log sketch of Scenario 1.

Accordingly, a research question arises as follows: *"Does the proposed approach introduce a significant performance overhead?"* ***(RQ3)***

B. EVALUATION RESULTS

The above case study was used to answer the above research questions. The evaluation was carried out using a PC with a Windows 7 64-bit operating system, 2 CPUs and a memory of 4GB. We evaluate the effectiveness and performance of our approach when it was used to handle unplanned environment exceptions in *Scenario 1* and dynamic changes of user requirements in *Scenario 2*.

1) HANDLING UNPLANNED EXCEPTIONS

(I) Effectiveness evaluation. In Scenario 1, we simulate unplanned exceptions through a failed invocation to *Business* hotel, to evaluate the effectiveness of our approach. As discussed in Section II-C, we need to declare an *APS* within the *invoke* activity to cover the case that the associated service is unavailable at run-time. In this example, we declare an *APS* within the *invoke* activity of the *Business* hotel service. Fig. 6 illustrates the VxBPEL fragment of the *Business* hotel invocation, from which we observe that a `dybind` element is inserted into the `invoke` activity of an alternative (i.e., *Business* variant) of the *Hotel Reservation*, and the value of the `selector` attribute is set to any value other than "mkchange" (in this case, we set it to "*Yes*").

At run-time, we close the *Business* hotel service to simulate an unavailable exception and at the same time, we deploy *Chain* hotel as an alternative registered in UDDI. In this context, any invocation of *Business* in the original VxBPEL-based service composition will result in a run-time exception. However, when the above extended VxBPE-based service composition with *APS* is executed, it is expected that an *APS* will be triggered when the unavailable *Business* is invoked and a new target service, say, *Chain Hotel*, will be discovered from UDDI and dynamically bound for continual execution.

Fig. 7 shows the execution log sketch of Scenario 1. During the execution, VxBPEL_Dyn_ODE first catched a service invocation exception and reported it (Lines 1-5). Then, the engine searched the *APS* associated with the *Business* and checked the value of the `selector` attribute. In this case, the value is not "mkchange", and the dynamic binding mechanism was trigged (Line 6). The integration layer triggered the *Service Discovery* process to find an alternative service from UDDI. When the UDDI proxy returned a candidate service, the engine replaced the invalid service (Lines 7-11), and created the corresponding request message for the new target service (Lines 12-13). Then, the dynamic binding mechanism was closed (Line 14).

(II) Performance overhead evaluation. To evaluate the performance overhead of supporting unplanned exceptions, we measure the deployment and execution time of the original VxBPEL-based service composition and the extended VxBPEL-based service composition of the *Travel*

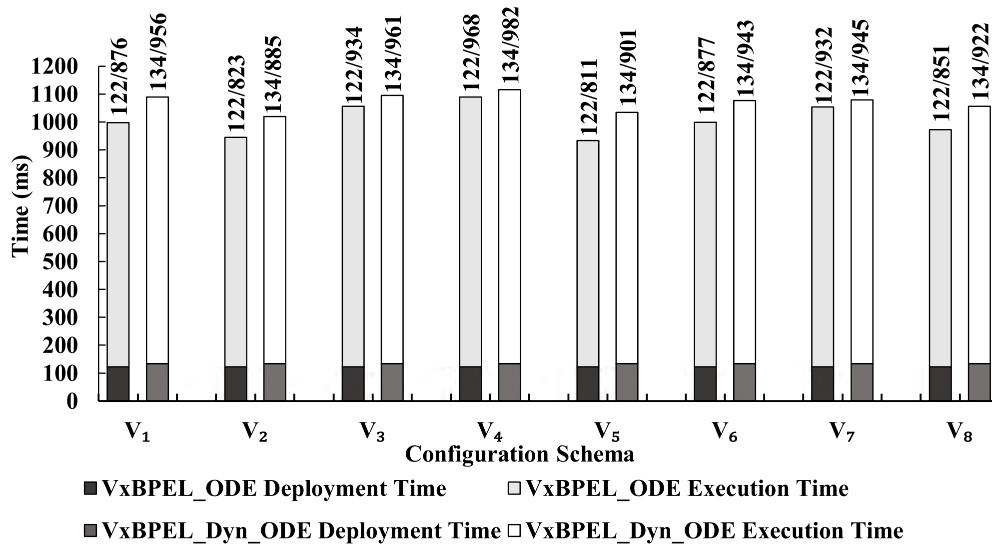

FIGURE 8. Deployment and execution time of the *Travel Reservation System* under different variation configurations.

TABLE 2. Variation Configuration Schema of *Travel Reservation System*.

Configuration	Vehicle Reservation	Hotel Reservation	Ticket Reservation	Insurance Purchase
V_1	Flight	Business	Scenic	Insurance A
V_2	Train	Business	Exploration	Insurance B
V_3	Flight	Business	Exploration	Insurance B
V_4	Flight	Business	Scenic	Insurance B
V_5	Flight	Business	Exploration	Insurance A
V_6	Train	Business	Exploration	Insurance A
V_7	Train	Business	Scenic	Insurance A
V_8	Train	Business	Scenic	Insurance B

Reservation System, respectively. Accordingly, VxBPEL_ODE and VxBPEL_Dyn_ODE are used to execute two versions of the *Travel Reservation System* with different variation configurations. Table 2 lists all sample configuration schema, labeled with V_i ($1 \leq i \leq 8$). For instance, V_1 represents that *Flight*, *Business*, *Scenic*, and *Insurance A* are chosen for each variation point of the *Travel Reservation System*. In this scenario, *Business* hotel is used to simulate unplanned exceptions and thus is fixed in all configuration schema.

Fig. 8 shows the performance comparison of the original VxBPEL-based service composition and the extended VxBPEL-based service composition of *Travel Reservation System*. Performance includes deployment and execution time for each configuration. Since both the VxBPEL service composition and the extended VxBPEL service composition contain variation design, the specifications for all variation configurations need to be deployed only once, while the specifications have to be run individually for each variation configuration scheme. From Fig. 8, we observe that for all variation configuration schema, there is no significant performance difference between the original VxBPEL-based service composition and the extended VxBPEL-based service composition. This further indicates that the performance overhead introduced by supporting unplanned exceptions is negligible.

```
<vxbpel:Variant name="Insurance_A">
  <bpel:invoke name="Invoke_Insurance_A" partnerLink="Insurance_APL"
    operation="provideInsurance" inputVariable="Insurance_APLRequest"
    outputVariable="Insurance_APLResponse">
  <vxbpel:dybind selector="mkchange"/></bpel:invoke>
</vxbpel:Variant>
```

FIGURE 9. VxBPEL fragment of *Insurance A* invocation.

2) HANDLING CHANGES IN USER REQUIREMENTS

(I) Effectiveness evaluation. At the design phase, it is difficult to predicate all possible requirements. That is, there is a possibility that new requirements emerge after service compositions have been developed and deployed. In Scenario 2, we simulate unplanned requirement changes at run-time. In particular, we assume that a customer is not satisfied with all the insurance services currently provided by the *Travel Reservation System*, and she intends to dynamically search a suitable insurance service (for instance, *Insurance C*) for *Insurance Purchase*. When the extended VxBPEL is used to implement such a service composition, we need to consider the uncertainty of selecting insurance services at the *Insurance Purchase*. Fig. 9 illustrates the VxBPEL fragment of the *Insurance A* invocation, where an *APS* is inserted into the invoke activity of an alternative (i.e., the *Insurance A* variant) of the *Insurance Purchase* and the value of the `selector` attribute is set as "mkchange".

At run-time, when the invoke activity of *Insurance A* is encountered, VxBPEL_Dyn_ODE will interrupt the

1 **Get Dynamic Binding Information:mkchange!**
2 UserConfig: type= DybindingSetting, id = 168
3 User Revise Config File, **Start Dynamic Binding Mechanism!**
4 **Replace the Original Operation: provideInsurance!**
5 Reorchestration Process!
6 **Insurance_C!**
7 **Start Modify URL!**
8 URL: http://localhost:8080/Insurance_C/services/Insurance_C?wsdl!
9 **Create Request Information of New Web Service: provideInsurance!**
10 <parameters><…></paramters>
11 **End Dynamic Binding Mechanism!**

FIGURE 10. An execution log sketch of Scenario 2.

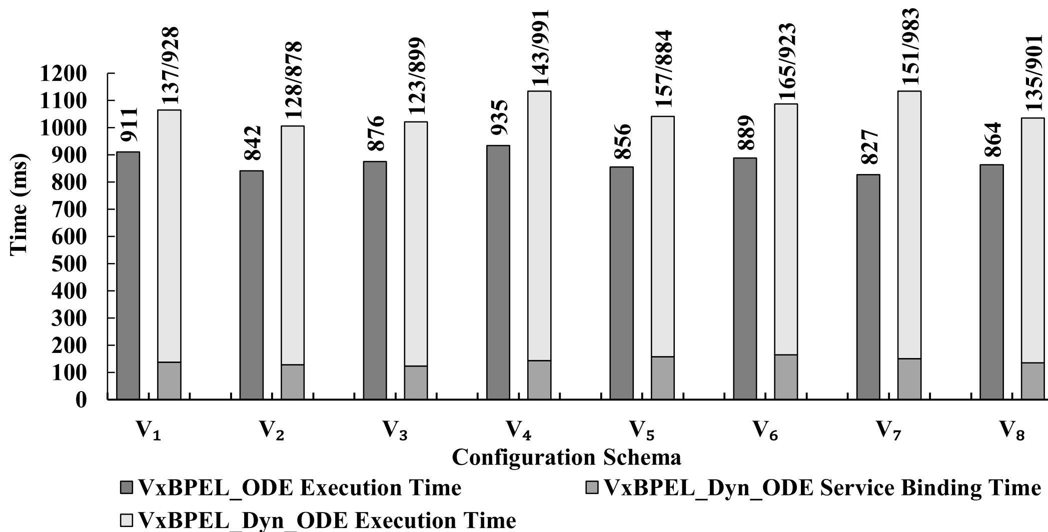

FIGURE 11. Service binding and execution time of the *Travel Reservation System* under different variation configurations.

execution and allow the customer to seek and specify a new insurance service. In this scenario, we use the management interface of JConsole to support such an dynamic service replacement process, and accordingly a newly discovered insurance service, namely, *Insurance C* is used to replace *Insurance A* in the *Travel Reservation System*.

Fig. 10 shows the execution log sketch of scenario 2. When the invoke activity of *Insurance A* was encountered and the value of the `selector` attribute inside the *APS* is "mkchange" (Line 1), the management interface was presented to the user to specify a new service as an alternative (Line 2). Then the dynamic binding mechanism was triggered to modify the service binding information and create the corresponding request message (Line 3-11). From the execution log, we observe that *Insurance A* has been replaced by *Insurance C* during the execution of the *Travel Reservation System*.

(II) Performance overhead evaluation. To evaluate the performance overhead of supporting unplanned requirement changes, we measure the execution time of the original VxBPEL service composition and the extended VxBPEL service composition of the *Travel Reservation System*, respectively. Again, VxBPEL_ODE and VxBPEL_Dyn_ODE are used to execute two versions of the *Travel Reservation System* with different variability configurations. Table 3 lists all sample configuration schema, labeled with V_i ($1 \leq i \leq 8$). In this scenario, we simulate an unplanned requirement change related to *Insurance A* with *Insurance Purchase*, while all the combinations of configuration schema with other three variation points are examined.

Fig. 11 shows the performance comparison of VxBPEL-based service composition and the extended VxBPEL-based service composition of the *Travel Reservation System*. Similar to Scenario 1, for all variation configurations, both the original VxBPEL-based and the extended VxBPEL-based service compositions need to be deployed only once, their deployment time is very close and thus is not included for performance comparison. However, the execution time (including binding time) for the two versions of service composition for the same configuration scheme

TABLE 3. Variability Configuration Schema of *Travel Reservation System*.

Configuration	Vehicle Reservation	Hotel Reservation	Ticket Reservation	Insurance Purchase
V_1	Flight	Business	Scenic	Insurance A
V_2	Train	Economy	Exploration	Insurance A
V_3	Flight	Economy	Exploration	Insurance A
V_4	Flight	Economy	Scenic	Insurance A
V_5	Flight	Business	Exploration	Insurance A
V_6	Train	Business	Exploration	Insurance A
V_7	Train	Business	Scenic	Insurance A
V_8	Train	Economy	Scenic	Insurance A

are somewhat different: the original VxBPEL-based service compositions do not involve newly discovered service binding (thus unable to handle such unplanned requirement changes), while the execution of the extended VxBPEL-based service compositions involves a dynamic service binding and thus introduces some extra performance overhead. From Fig. 11, we observe that the performance difference between original VxBPEL-based service compositions and the extended VxBPEL-based service compositions is not very large for all variation configuration schema. This further indicates that the performance overhead introduced by dynamic service binding to support unplanned requirement changes is acceptable.

C. ANSWERS TO RESEARCH QUESTIONS

The experimental results demonstrate the effectiveness and performance of the proposed approach in handling unplanned exceptions and changes in user requirements.

Answer to RQ1: The proposed approach supports the run-time service replacement when a predefined service becomes unexpectedly unavailable, through declaring an *APS* in the associated `invoke` element, catching possible run-time exceptions, and replacing the unavailable service with an alternative service from the UDDI dynamically.

Answer to RQ2: The proposed approach supports run-time requirement changes in case that some services currently provided in the service composition cannot meet the requirements, through declaring an *APS* in the associated `invoke` element, and allowing the user to specify newly discovered services for run-time service replacement without the need for service compositions redesign and redeployment.

Answer to RQ3: The deployment time of the original VxBPEL-based and extended VxBPEL-based service compositions is almost the same, while their execution time shows some difference, which can be negligible especially in the context of a long running business process. Therefore, the performance overhead introduced by supporting unplanned dynamic changes is in general acceptable. Furthermore, a service composition without using our approach normally needs to undergo a tedious and time-consuming redesign and redeployment process when unplanned changes occur at run-time. In summary, the proposed approach significantly enhances the dynamic adaptation capability of service compositions while introducing an acceptable performance overhead.

V. RELATED WORK

Service compositions are expected to be adaptable in case that they need to accommodate frequent requirement changes and environmental exceptions. Many efforts have been made to address the adaptation issue of service compositions. We introduce below several representative approaches and provide a comparison of them with our approach.

AOP-based adaptive service composition. This category of approaches applies AOP to improve the adaptation of service compositions. An early research work is AO4BPEL [29], which extends WS-BPEL with AOP, to enhance the self-adaption ability of WS-BPEL compositions. Newly introduced business rules are specified as aspects, which is first predefined and registered by the administrator and then is activated/deactivated at run-time to support dynamic changes for service compositions. VieDAME [30] is an AOP-based extension to the ActiveBPEL engine, which monitors and captures various QoS attributes of a running WS-BPEL process. These AOP-based approaches only support planned dynamic changes since dynamic service replacement is restricted to those predefined ones in a service repository. In contrast, our approach supports unplanned dynamic changes through dynamic service replacement with those predefined at design time or newly discovered/provided at-runtime. The potential service replacement in a service composition is supported by a variability-supporting language, while AOP is adopted to implement the integration of dynamic service discovery and the process engine.

Proxy-based adaptive service composition. Casati et al. [31] proposed eFlow to support adaptive and dynamic service composition. TRAP/BPEL [18] uses a generic proxy to encapsulate autonomic behavior through self-management policies. WS Binder [32] is a framework supporting the binding of an abstract service to a concrete one in a service composition, in order to achieve an optimal QoS according to functional and non-functional preferences and/or constraints. Hammas et al. [33] proposed to detect service dynamic changes via a monitoring module. When a failure or unavailability occurs, the failed services will be replaced with the corresponding backup services. However, unplanned dynamic changes are not supported since backup services have to be fixed at design time. Unlike the existing approaches where abstract services are declared with respect to service composition instances and their treatment has to be implemented specifically and individually, our approach supports abstract services with a generic language construct, and their treatment is embedded in the underlying engine (transparent to service composition designers).

Annotation-based adaptive service composition. This category of approaches achieves the adaptation capability of service compositions through policy annotation. Sheng et al. [34] proposed an adaptive service composition approach where service contexts and exceptions are

TABLE 4. Comparison summary of related approaches.

Approach	Level	Perspective	Tool support	Validation
our approach	specification	variability management and dynamic binding	Yes	Yes
[29]	instance	AOP	Yes	No
[30]	instance	AOP	Yes	Yes
[31]	instance	proxy	Yes	No
[18]	instance	generic proxy	No	Yes
[32]	instance	dynamic proxy	Yes	No
[33]	instance	backup services	No	No
[34]	instance	context annotation	Yes	Yes
[35]	instance	policy annotation	No	Yes
[36]	instance	context annotation	Yes	No
[37]	instance	late binding	Yes	Yes
[38]	instance	AI planning	Yes	Yes
[39]	instance	variability models	Yes	Yes
[40]	instance	feature model	Yes	Yes
[41]	instance	multi-objective optimization	Yes	Yes

configurable to accommodate the needs of different users. Unplanned dynamic changes are not supported since exception handling policies have to be specified at design time. Laleh et al. [35] addressed the adaptation of service compositions in terms of constraints, which not only consider internal constraints (usage restrictions imposed by service providers), but also external constraints (usage restrictions externally-defined at run-time). This approach only considers the constraints concerning usage restrictions of composite services, while dynamic changes from the environment and user requirements are not considered.

Context-aware adaptive service composition. This category of approaches adapts service compositions to various contexts. Wieland et al. [36] proposed an approach to runtime adaptation of situation-aware workflows. To simplify the modeling of situation-aware workflows, Képes et al. [37] proposed an approach to enabling the transformation of a traditional workflow model into a situation-aware workflow model. Bucchiarone et al. [38] proposed a planning-based approach to address the consistency issues of context-aware dynamic service composition and execution. To handle unplanned dynamic changes, these approaches require the prediction of all possible situations, which would be impractical.

Variability management-based adaptive service composition. This category of approaches addresses the adaptation issue by dealing with the variability of service compositions. Alférez et al. [39] supported dynamic adaptation of service compositions by modeling service features in terms of variability. Similarly, Nguyen et al. [40] proposed a feature-based framework to develop and provide customizable Web services. Both these approaches and our approach achieve the adaptation of service compositions through variability management. Unlike these approaches that focus on feature-level variability modeling of service compositions, our approach provides a general executable service composition language which combines variability management and the concept of abstract proxy, and supports unplanned dynamic changes through run-time service replacement.

Model-based adaptive service composition. Yau et al. [41] proposed a multi-objective optimization approach to address the QoS-oriented adaptation of service compositions. Unlike this approach which focuses on the QoS-oriented adaption of service compositions, our approach aims to improve the adaptation capability of service compositions in handling unplanned dynamic changes in the environment and user requirements.

A comparison of the above-mentioned approaches is summarized in Table 4. The "Level" column indicates whether the concerned approach works at a specification or instance level, the "Perspective" column refers to the solution used by the concerned approach, the "Tool support" column indicates the availability of supporting tools, and the "Validation" column indicates whether the concerned approach is validated. From Table 4, we can see that all existing approaches somehow support dynamic adaptation of service compositions at the *instance* level, whereas only our approach at the *specification* level. Besides, our approach is aided with tool support and validated with a case study.

VI. CONCLUSION

We have presented a novel adaptive service composition approach to supporting unplanned dynamic changes to service compositions by combining two typical adaptive service composition methods, namely, proxy-based and variability management-based methods. The proposed approach extends VxBPEL developed in our previous work with the concept of *abstract proxy service*. This extension is key to declaring those services that are dynamically discovered or specified by the user, to complement the predefined variants within VxBPEL-based service compositions. It overcomes the limitation shared by all existing imperative variability management-based adaptive service approaches. To exercise the proposed approach, we have further developed an extended VxBPEL engine to support the *abstract proxy services* and facilitate two ways of dynamic service bindings (namely, run-time service discovery and user specification). Finally, we have conducted a case study to validate the

effectiveness of the proposed approach and quantify the performance overhead resulting from variability management and dynamic binding.

The work presented in this paper advances the state of art of adaptive service compositions from the following perspectives. First, a language-based generic approach is proposed to address unplanned dynamic changes in service compositions, where a language construct is used to express the concept of abstract proxy service and a mechanism to enable the execution of abstract proxy services and the binding of newly discovered or specified services. The approach is further facilitated by a supporting platform and thus provides a systematic and practical solution rather than only a proof of concept. Second, the approach overcomes the limitations of variability management-based adaptive service composition approaches in supporting unplanned dynamic changes. In particular, the proposed approach enhances the dynamic adaptability of VxBPEL-based service compositions. Third, the approach overcomes the limitations of proxy-based adaptive service composition approaches while introducing the proxy mechanism, achieving low maintenance and efficiency.

For future work, we intend to evaluate our approach with more complex scenarios, and integrate with advanced service discovery and recommendation techniques to achieve ''smarter'' or more automated service compositions. We are also interested in exploring the possibility of adapting our approach to specific application domains such as Cyber-Physical Systems (CPS), or development paradigms such as DevOps.

REFERENCES

[1] C. Peltz, "Web services orchestration and choreography," *Computer*, vol. 36, no. 10, pp. 46–52, Oct. 2003.
[2] Software & Information Industry Association. (2001). *Software As A Service: Strategic Backgrounder*. [Online]. Available: http://www.siia.net/estore/pubs/SSB-01.pdf
[3] W3C. *Web Services Glossary*. Accessed: 2004. [Online]. Available: http://www.w3.org/TR/ws-gloss/
[4] Peltz. (2003). *Web Services Orchestration: A Review of Emerging Technologies*. [Online]. Available: http://xml.coverpages.org/HP-WSOrchestration.pdf
[5] R. T. Fielding. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. [Online]. Available: http://www.ics.uci.edu/simfielding/pubs/dissertation/rest_arch_style.htm
[6] Wikipedia. (2017). *Mobile App*. [Online]. Available: https://en.wikipedia.org/wiki/Mobile_app
[7] Z. Wu, Y. He, and D. Liu, "An approach to support semantic-enabled Web service," in *Proc. Joint Int. Comput. Conf. (JICC)*, 2005, pp. 177–182.
[8] Z. Xiao, D. Cao, C. You, and H. Mei, "Towards a constraint-based framework for dynamic business process adaptation," in *Proc. IEEE Int. Conf. Services Comput. (SCC)*, Jul. 2011, pp. 685–692.
[9] D. Jordan *et al.*, "Web services business process execution language version 2.0," *OASIS Standard*, vol. 11, no. 120, p. 5, 2007.
[10] M. Koning, C.-A. Sun, M. Sinnema, and P. Avgeriou, "VxBPEL: Supporting variability for Web services in BPEL," *Inf. Softw. Technol.*, vol. 51, no. 2, pp. 258–269, 2009.
[11] C. Sun, R. Rossing, M. Sinnema, P. Bulanov, and M. Aiello, "Modeling and managing the variability of Web service-based systems," *J. Syst. Softw.*, vol. 83, no. 3, pp. 502–516, 2010.
[12] M. Aiello, P. Bulanov, and H. Groefsema, "Requirements and tools for variability management," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf. Workshops (COMPSACW)*, Jul. 2010, pp. 245–250.
[13] A. Hallerbach, T. Bauer, and M. Reichert, "Managing process variants in the process lifecycle," in *Proc. 10th Int. Conf. Enterprise Inf. Syst.*, 2008, pp. 154–161.
[14] M. Pesic, M. H. Schonenberg, N. Sidorova, and W. M. P. van der Aalst, "Constraint-based workflow models: Change made easy," in *On the Move to Meaningful Internet Systems* (Lecture Notes in Computer Science). Berlin, Germany: Springer, 2007, pp. 77–94.
[15] C.-A. Sun and M. Aiello, "Towards variable service compositions using VxBPEL," in *Proc. 10th Int. Conf. Softw. Reuse, High Confidence Softw. Reuse Large Syst.* Berlin, Germany: Springer-Verlag, 2008, pp. 257–261.
[16] C.-A. Sun, T. Xue, and M. Aiello, "ValySeC: A variability analysis tool for service compositions using VxBPEL," in *Proc. 5th IEEE Asia–Pacific Services Comput. Conf.*, Dec. 2010, pp. 307–314.
[17] C.-A. Sun, Z. Wang, K. Wang, T. Xue, and M. Aiello, "Adaptive BPEL service compositions via variability management: A methodology and supporting platform," *Int. J. Web Services Res.*, vol. 16, no. 1, pp. 37–69, 2019.
[18] O. Ezenwoye and S. Sadjadi, "TRAP/BPEL: A framework for dynamic adaptation of composite services," in *Proc. WEBIST*, 2007, pp. 216–221.
[19] A. Erradi and P. Maheshwari, "wsBus: QoS-aware middleware for reliable Web services interaction," in *Proc. IEEE Int. Conf. e-Technol., e-Commerce e-Service*, Hong Kong, Mar./Apr. 2005, pp. 634–639.
[20] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Trans. Softw. Eng.*, vol. 33, no. 6, pp. 369–384, Jun. 2007.
[21] *Apache ODE*. Accessed: 2011. [Online]. Available: http://ode.apache.org/developerguide/architectural-overview.html
[22] C.-A. Sun, T.-H. Xue, and C.-J. Hu, "VxBPELEngine: A change-driven adaptive service composition engine," *Chin. J. Comput.*, vol. 36, no. 12, pp. 2441–2454, 2013.
[23] C.-A. Sun, P. Wang, X. Zhang, and M. Aiello, "VxBPEL_ODE: A variability enhanced service composition engine," in *Proc. APWeb Workshops*, in Lecture Notes in Computer Science, vol. 8710. Cham, Switzerland: Springer, 2014, pp. 69–81.
[24] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture Process and Organization for Business Success*. Boston, MA, USA: Press/Addison, 1997.
[25] G. Kiczales *et al.*, "Aspect-oriented programming," in *Proc. 11th Eur. Conf. Object-Oriented Program. (ECOOP)*, in Lecture Notes in Computer Science, vol. 1241. Berlin, Germany: Springer-Verlag, 1997, pp. 220–242.
[26] J. D. Gradecki and Nicholas Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, 1st ed. Hoboken, NJ, USA: Wiley, 2003.
[27] Sun Microsystems. (2016). *Java Management Extensions*. [Online]. Available: http://java.sun.com/products/JavaManagement/
[28] Y. Xing, F. Gu, and H. Mei, "Feature model driven Web services composition approach and its support tool," *J. Softw.*, vol. 18, no. 7, pp. 1582–1591, 2007.
[29] A. Charfi and M. Mezini, "Aspect-oriented Web service composition with AO4BPEL," in *Web Services*. Berlin, Germany: Springer, 2004, pp. 168–182.
[30] O. Moser, F. Rosenberg, and S. Dustdar, "Non-intrusive monitoring and service adaptation for WS-BPEL," in *Proc. World Wide Web Conf.*, 2008, pp. 815–824.
[31] F. Casati, S. Ilnicki, L.-J. Jin, V. Krishnamoorthy, and M.-C. Shan, "Adaptive and dynamic service composition in eFlow," in *Proc. CAISE*, in Lecture Notes in Computer Science, vol. 1789. Berlin, Germany: Springer, 2000, pp. 13–31.
[32] M. Di Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo, and E. Di Nitto, "WS Binder: A framework to enable dynamic binding of composite Web services," in *Proc. Int. Workshop Service-Oriented Softw. Eng.*, 2006, pp. 74–80.
[33] O. Hammas, S. B. Yahia, and S. B. Ahmed, "Adaptive Web service composition insuring global QoS optimization," in *Proc. IEEE Int. Symp. Netw., Comput. Commun.*, May 2015, pp. 1–6.
[34] Q. Z. Sheng, B. Benatallah, Z. Maamar, and A. H. H. Ngu, "Configurable composition and adaptive provisioning of Web services," *IEEE Trans. Services Comput.*, vol. 2, no. 1, pp. 34–49, Jan. 2009.
[35] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, "Constraint adaptation in Web service composition," in *Proc. IEEE 14th Int. Conf. Services Comput.*, Jun. 2017, pp. 156–163.
[36] M. Wieland, H. Schwarz, and U. Breitenbücher, and F. Leymann, "Towards situation-aware adaptive workflows: SitOPT—A general purpose situation-aware workflow management system," in *Proc. IEEE Int. Conf. Pervasive Comput. Commun. Workshops (PerCom Workshops)*, Mar. 2015, pp. 32–37.

[37] K. Képes, U. Breitenbücher, S. G. Sáez, J. Guth, F. Leymann, and M. Wieland, "Situation-aware execution and dynamic adaptation of traditional workflow models," in *Proc. 5th Eur. Conf. Service-Oriented Cloud Comput. (ESOCC)*, 2016, pp. 69–83.

[38] A. Bucchiarone, A. Marconi, M. Pistore, and H. Raik, "A context-aware framework for dynamic composition of process fragments in the Internet of services," *J. Internet Services Appl.*, vol. 8, no. 1, p. 6, 2017.

[39] G. H. Alférez, V. Pelechano, R. Mazo, C. Salinesi, and D. Diaz, "Dynamic adaptation of service compositions with variability models," *J. Syst. Softw.*, vol. 91, no. 5, pp. 24–47, 2014.

[40] T. Nguyen, A. Colman, and J. Han, "A feature-based framework for developing and provisioning customizable Web services," *IEEE Trans. Services Comput.*, vol. 9, no. 4, pp. 496–510, Jul./Aug. 2016.

[41] S. S. Yau *et al.*, "Toward development of adaptive service-based software systems," *IEEE Trans. Services Comput.*, vol. 2, no. 3, pp. 247–260, Jul. 2009.

CHANG-AI SUN received the bachelor's degree in computer science from the University of Science and Technology Beijing (USTB), China, and the Ph.D. degree in computer science from Beihang University. He was an Assistant Professor with Beijing Jiaotong University, China, and a Post-Doctoral Fellow with the Swinburne University of Technology, Australia, and the University of Groningen, The Netherlands. He is currently a Full Professor with the School of Computer and Communication Engineering, USTB. His research interests include service-oriented computing and software testing.

ZHEN WANG received the bachelor's degree in computer science from the University of Science and Technology Beijing, where she is currently pursuing the Ph.D. degree with the School of Computer and Communication Engineering. Her current research interests include service-oriented computing and software architecture.

ZAIXING ZHANG received the bachelor's degree in computer science from the University of Science and Technology Beijing, where he is currently pursuing the master's degree with the School of Computer and Communication Engineering. His current research interest includes service-oriented computing.

PAN WANG received the bachelor's degree in computer science from the University of Science and Technology Beijing, where he is currently pursuing the master's degree with the School of Computer and Communication Engineering. His current research interest includes service-oriented computing.

XIAO HE received the Ph.D. degree in computer science from Peking University, China. He was a Visiting Professor with the Johann Bernoulli Institute, University of Groningen, The Netherlands. He is currently an Associate Professor with the School of Computer and Communication Engineering, University of Science and Technology Beijing. His main research interests include model transformation languages, model generation, metamodeling, and domain-specific modeling.

JUN HAN received the Ph.D. degree in computer science from the University of Queensland, Australia. Since 2003, he has been a Full Professor of software engineering with the Swinburne University of Technology, Australia. He has published more than 250 peer-reviewed articles. His current research interests include service and cloud systems engineering, adaptive and context-aware software systems, and software architecture and quality.