VxBPEL_ODE: A Variability Enhanced Service Composition Engine

Chang-Ai Sun[1,*], Pan Wang[1], Xin Zhang[1], and Marco Aiello[2]

[1] School of Computer and Communication Engineering, University of Science and Technology Beijing, 100083 Beijing, China
casun@ustb.edu.cn
[2] Johann Bernoulli Institute, University of Groningen, 9747 AG, Groningen, The Netherlands
m.aiello@rug.nl

Abstract. Service compositions have become a powerful development paradigm to create distributed applications out of autonomous Web services. Since such applications are often deployed and executed in open and dynamic environments, variability management is a crucial enabling technique. To address the adaptation issue of service compositions, we proposed VxBPEL, an extension of BPEL for supporting variability, and a variability-based adaptive service composition approach which employs VxBPEL for variability implementation. In this paper, we present a VxBPEL engine for supporting the execution of VxBPEL service compositions. The engine is called VxBPEL_ODE and is implemented by extending a widely recognized open source BPEL engine, Apache ODE. We discuss key issues of developing VxBPEL_ODE, and three real-life service compositions are employed to evaluate and compare its performance with another VxBPEL engine we developed in our previous work. VxBPEL_ODE, together with analysis, design, and run-time management tools for VxBPEL, constitutes a comprehensive supporting platform for variability-based adaptive service compositions.

Keywords: Service Oriented Architecture, Service Compositions, Variability Management, VxBPEL.

1 Introduction

Service Oriented Architecture (SOA) is a mainstream paradigm for application development in the context of the Internet and highly-scalable systems [1]. Since individual services usually provide limited functionalities and are unable to meet in isolation the actual demand, service compositions are a powerful mechanism for integration of data and applications in the context of distributed, dynamic, heterogeneous environments. A service composition can be seen as a process in which multiple individual services participate and are coordinated to support complex business goals. Service compositions are able to support rapid business re-engineering and optimization. Business Process Execution Language (BPEL) is a process-oriented executable service composition language which can be used to construct loosely coupled systems by orchestrating a bundle of Web services [2]. Business processes implemented by such service compositions are expected to be flexible enough to cater for frequent and rapidly-changing requirements [3]. For instance, a service under composition may become unavailable or fail to provide the expected functionalities. Consequently, flexibility of service compositions is highly desirable. Unfortunately, the standard version of BPEL is limited in supporting such an expectation [4].

To address the adaptation issue of service compositions, we have proposed a variability-based adaptive service composition approach [5]. In the proposed approach, the changes are treated as the first-class objects, and VxBPEL [6], an extension to BPEL, is used for specifying variation. VxBPEL provides a set of constructs for supporting variation design, such as variation points and variants [6][7]. Since these constructs are not of standard BPEL elements and cannot be executed by standard BPEL engines, we developed a VxBPEL engine called VxBPEL_ActiveBPEL [8] by extending ActiveBPEL [10], a well recognized BPEL engine. To support the analysis and maintenance of variation, we developed a tool, ValySec [9], which can be used to automatically extract variation definitions from VxBPEL service compositions and visualize variation configurations. With the proposed approach and tools, designers can implement adaptive service compositions by identifying possible changes within service compositions and specifying them with alternatives.

In this paper, we present a VxBPEL engine, VxBPEL_ODE, to further provide an integrated supporting platform for variability-based adaptive service compositions. As a mainstream application development platform, Eclipse provides two open source plug-ins for BPEL, namely BPEL Designer and Apache ODE [11]. In our previous work [5], we developed a visual VxBPEL design tool called VxBPEL Designer by extending BPEL Designer. Since both VxBPEL_ODE and VxBPEL Designer are implemented as Eclipse plug-ins, VxBPEL_ODE can be seamlessly integrated with VxBPEL Designer, which facilitates the provision of an integrated supporting platform for VxBPEL-based adaptive service compositions. Furthermore, VxBPEL_ODE is significantly different in architecture from VxBPEL_ActiveBPEL that relies on ActiveBPEL, which is no longer available as open source. In this paper, we examine key issues relevant to development of VxBPEL_ODE and compare the performance of VxBPEL_ODE with that of VxBPEL_ActiveBPEL using three VxBPEL service composition cases.

The rest of the paper is organized as follows. Section 2 introduces the underlying concepts of VxBPEL. Section 3 discusses the design and implementation of VxBPEL_ODE. Section 4 validates VxBPEL_ODE and compares its performance with VxBPEL_ActiveBPEL through case studies. Section 5 describes related work. The conclusion is reported in Section 6.

2 Background

BPEL [2] is a process-oriented, executable composition language which describes control flows between activities in a business process. Activities are basic interaction units, and divided into basic and structural activities. Basic activities are an atomic execution step, such as *assign* and *invoke*. Structural activities are compositions of basic activities and/or structural activities. A standard version of BPEL process specification is fixed, which means that all activities and their relationships must be exactly defined before deployment and run-time changes are not allowed after deployment.

Variability is the ability of a software system to extend, change, customize or configure itself in any particular environment [4]. The core concepts of the variability include *variation points*, *variants*, and *dependency*. The *variation point* is the part of the software system that may change. Typically, a *variation point* occurs when multiple design options are to be chosen. After one option at a variation point is selected, a so called *variant* is obtained; the collection of choices at each variation point is referred to as a variation *configuration*. *Dependency* specifies constraints between the different variations corresponding to the variant.

In order to introduce variability into service compositions, we proposed VxBPEL which extends BPEL for modeling variability [4]. VxBPEL employs the COVAMOF variability framework [12] and provides constructs such as *variants*, *variation points*, and their *configurations* and *constraints*. A *variant* is defined using the tag <vxbpel:Variant>. The associated name is used to identify a variant. The element enclosed by the tag <vxbpel:VPBPELCode> is used to specify the choice contained within the variant, which corresponds to original BPEL activity, such as an *invoke* activity. A *variation point* is defined using the tag <vxbpel:VariationPoint>. It is identified by its unique name and may contain one or more variants enclosed by the tag <vxbpel:Variants>. The tag <vxbepx:ConfigurableVairationPoint> is used to specify the selection of variants associated with variation points, and the tag <vxbpel:RequiredConfiguration> specifies the realization between the higher variation point and the lower variants. *Constraints* defined in the tag <vxbpex:Constraint> provide a powerful mechanism for defining variant dependencies. To further facilitate its adoption, an integrated supporting platform is expected. VxBPEL_ODE, a core component of such a platform, is our proposal described next.

3 Design and Implementation

The variability-based adaptive service composition approach consists of two steps: (1) BPEL is first used to specify service compositions, and (2) variability constructs defined in VxBPEL are then used to define and configure variations of the BPEL service compositions. The resulting service compositions are referred to as *VxBPEL processes*. Obviously, VxBPEL processes cannot be executed by the existing BPEL engines at run-time, since they contain non-standard BPEL elements. We next discuss how to develop VxBPEL_ODE by extending Apache ODE.

3.1 Design

The key issue of developing a VxBPEL engine is how to treat newly introduced variability elements in a BPEL process (i.e. a VxBPEL interpreter is needed) and how to enforce them with the existing BPEL engine (i.e. the interaction between the BPEL engine and the VxBPEL interpreter). A general rule of thumb we followed for designing VxBPEL_ODE is to leverage the implementation of Apache ODE to the maximum without significantly modifying its architecture. Fig. 1 shows the architecture of VxBPEL_ODE. *VxBPEL Compiler* and *Configuration Management* are newly introduced components in order to enable the execution of VxBPEL elements. VxBPEL_ODE first invokes *ODE BPEL Compiler* and *VxBPEL Compiler* to compile the VxBPEL process. The former is in charge of the compilation of all standard BPEL elements, while the latter is in charge of the compilation of VxBPEL elements, mapping variants to BPEL activities, and storing association of variation points and variants. *Configuration Management* is responsible for the interactions between *VxBPEL Complier* and *ODE BPEL Compiler* and selecting the corresponding variants for the serialization. The compiled VxBPEL representation (i.e. *Compiled Process Definitions*) is an object model similar in structure to the underlying BPEL process document, and all variability elements are resolved after the compilation process. Finally, *ODE BPEL Runtime* is used for the execution of compiled processes. We next discuss each major component individually.

- *VxBPEL Compiler* is responsible for preprocessing VxBPEL-specific elements. First, it creates an object model for all VxBPEL elements similar in structure to the object model for BPEL elements. Second, it maps all variants of a variation point to BPEL activities. Third, it stores the association of variation points and variants.

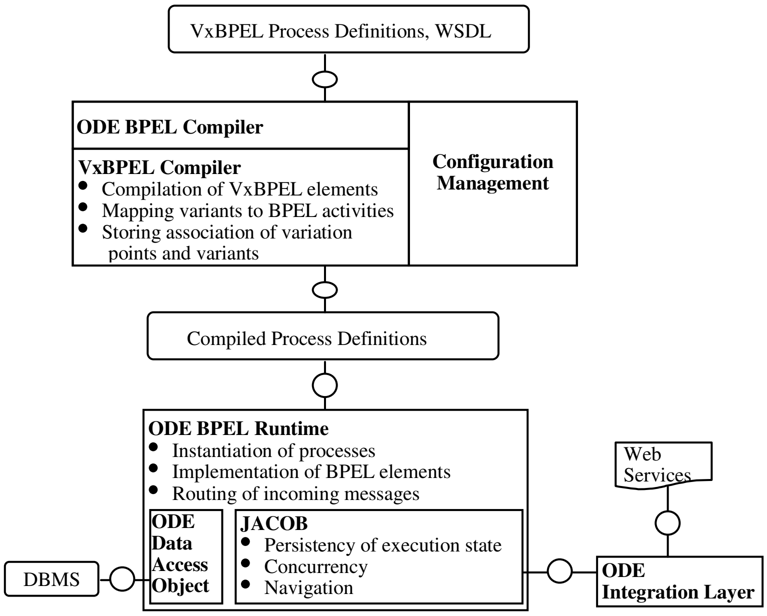
VxBPEL Process Definitions, WSDL
ODE BPEL Compiler
VxBPEL Compiler
Compilation of VxBPEL elements
Mapping variants to BPEL activities
Storing association of variation points and variants
Configuration Management
Compiled Process Definitions
ODE BPEL Runtime
Instantiation of processes
Implementation of BPEL elements
Routing of incoming messages
ODE Data Access Object
JACOB
Persistency of execution state
Concurrency
Navigation
DBMS
Web Services
ODE Integration Layer

Fig. 1. Architecture of VxBPEL_ODE

- *ODE BPEL Compiler* is responsible for the conversion of standard BPEL elements into a compiled representation.
- *Configuration Management* is used to manage variants associated with a variation point. It analyzes variation configurations specified in ConfigurableVairationPoint and provides the operations for changing the variability configuration at run-time. During the serialization, a specific variant of a variation point is selected depending on the variability configuration and its corresponding object is created. Through the treatment, all VxBPEL-specific elements are resolved. The compiled representation has resolved the various named references present in the VxBPEL, internalized the required WSDL and type information, and generated various constructs.
- *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.
- *ODE Data Access Objects (ODE DAOs)* mediates the interaction between the ODE BPEL Runtime and an underlying database management system DBMS. ODE DAOs normally provides interfaces for tracking active instances, routing message, referring to the values of BPEL variables for each instance, referring to the values of BPEL partner links for each instance, and serializing process execution states.
- *JACOB* provides an application concurrency mechanism, including a transparent treatment of process interrupt and persistency of execution state.
- *ODE Integration Layer* provides an execution environment by providing communication channels to interact with Web services, thread scheduling mechanisms, and the lifecycle management for ODE BPEL Runtime.

We next examine interactions among components of the engine using a UML collaboration diagram, as shown in Fig. 2. Note that the name of each component is described by texts above the box. The execution process is described as follows.

Phase 1. When the engine is started, it first configures the process deployment directory through the container in which the engine runs, and then activates *DeploymentPoller* in ODE Integration Layer to check whether a VxBPEL process is deployed (i.e. Step 1). If detected, an empty file will be created. After that, the engine employs a utility tool to monitor this directory. If a file in this directory is updated, *DeploymentPoller* will re-deploy the process; if the deployed process is removed, *DeploymentPoller* is responsible for removing this process, accordingly.

Phase 2. The engine invokes the "*deploy ( )*" method provided by *ProcessStoreImpl* in ODE Data Access Objects to deploy the VxBPEL process (i.e. Step 2). During the deployment, major activities include parsing the VxBPEL file, creating an object model for the execution, and serializing all relevant objects in a binary file. These activities are implemented by *DeploymentUnitDir* in ODE Data Access Objects, which encapsulates the "*Compile (File bpelFile)*" method provided by *BpelC* in ODE BPEL Compiler. *DeploymentUnitDir* is in charge of a deployment unit, which corresponds to a process directory. The "*Compile (File bpelFile)*" method is responsible for the following tasks (i.e. Steps 3~11):

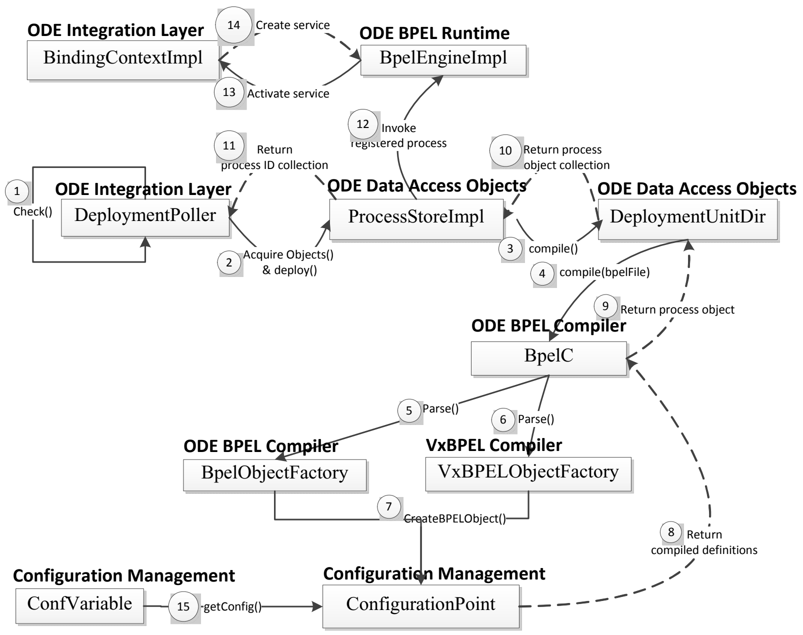

Fig. 2. The running process of VxBPEL_ODE

(1) *Parsing the VxBPEL file (i.e. Steps 3~6)*: This is implemented by the "*parse()*" methods provided by *BpelObjectFactory* in ODE BPEL Compiler and *VxBPELObjectFactory* in VxBPEL Compiler. This method parses the XML file and generates Java objects for process elements. During the paring phase, it generates objects for standard BPEL elements, and records the information about variation definitions and configurations. *ConfigurationPoint* in Configuration Management is responsible for parsing the VxBPEL configuration elements and providing interfaces for the configuration modifications.

(2) *Creating the binary file* (i.e. Steps 7~11): Objects generated during the parsing phase contains process and variation attributes. When the variation configuration is met for the first time, a variant specified by the *defaultVariant* configuration is selected. When the variation configuration is switched to a variation configuration scheme, the engine will invoke a method provided by *ConfigurationPoint* to select the specified variants. This variant selection process repeats until all variation points are handled. After the treatment, only BPEL process objects remain because variants associated with a variation point are of standard BPEL elements. The engine allows for changing the variation configurations at run-time, which is implemented by methods provided by *ConfigurationPoint* and *BpelC*. The variation configuration parameter is transferred to *ConfVariable* in Configuration Management, which maintains the current variation configuration (i.e. Step 15). Finally, all process relevant objects are serialized into a binary file.

Phase 3. The *BpelEngineImpl* in ODE BPEL Runtime executes the registered process (i.e. Steps 12~14). If an object is involved in the registered process, its relevant information is extracted from the binary file. For instance, when a Web service is invoked, the engine will call interfaces provided by *BpelBindContexImpl* in ODE Integration Layer to create the bound service for execution.

3.2 Implementation

We have implemented VxBPEL_ODE using the Java language. Two key issues related to the implementation are described next.

(a) *Compilation and deployment of VxBPEL elements*: As mentioned above, the engine first needs to deploy a process before its execution. An important task of the deployment is to compile the VxBPEL file and create an object model suitable for execution. This compilation process involves BPEL and VxBPEL elements that are distinguished by their name spaces. The former are handled by *ODE BPEL Compiler*, while the latter are handled by *VxBPEL Compiler*. To deliver an isomorphic implementation of the engine, we referred to the implementation of *ODE BPEL Compiler* during the implementation of *VxBPEL Compiler*.

(b) *Implementation of run-time variation configuration*: To support the implementation of variability, the engine first stores the variation definitions, and then selects variants based on the variation configuration during the serialization of process activities. The variation configuration is implemented by the component *ConfigurationPoint*. It records the name of the current VxBPEL process and the information about variation configurations, and selects the variants for execution at the serialization phase. Furthermore, variation configuration of the VxBPEL process can be changed at run-time via a method provided by *ConfigurationPoint*.

4 Evaluation

We report on an evaluation of VxBPEL_ODE using three VxBPEL processes. The goal of the evaluation is three-fold: (i) testing VxBPEL_ODE; (ii) evaluating the overhead of variability management by comparing BPEL and VxBPEL implementations for the same business scenarios, and (iii) comparing the performance of VxBPEL_ODE with that of VxBPEL_ActiveBPEL using the same VxBPEL processes for the same business scenarios.

4.1 Subject Programs

The *travel agency* process is a typical scenario describing how a travel request is processed [5]. When constructed from distributed services, the process often involves the composition of *travel agency*, *hotel*, *flight,* and *banking services*. These services can be provided by independent organizations. Once the system receives a request, the customer first chooses a travel agency, then selects the desired flights as well as hotels, the system asks the consumer to pay the package via an online banking service.

The *car estimation* process [5] describes a common scenario how a car repair is negotiated depending on the situation of the car and its repair cost. This process consists of five services, namely *Initial Estimate*, *Exterior Estimate*, *Interior Estimate*, *Powertrain Estimate*, and *Final Estimate*. Before the repair, a company needs to know the situation of the car, and the customer then decides to accept the reparation or not after being informed of the costs. Once the customer submits a request for estimation, the process first invokes an *Initial Estimate* service. In case of a simple estimation, the process invokes *Final Estimate* and notifies the cost to the customer; in case of a complex situation, the request is transferred to *Exterior Estimate*, *Interior Estimate*, and *Powertrain Estimate* services for a deep estimation, and the process then invokes the *Final Estimate* service and returns the cost.

The *smart shelf* process represents a complex shopping situation of shelf management [5], where one monitors quantity as well as quality of goods. Consumers first send a shopping request which includes the name and quantity of goods to be purchased. After receiving the request, the system first creates a formal request which is composed of *service time*, *quantity of goods*, *category of goods* and *period of goods* and then checks the *quantity*, *location* and *period* of goods. Next, the system checks the quantity, location, and period of goods, and executes various routine procedures based on the checking results. Finally, the system responds to the consumer with a confirmation message if *quantity*, *location*, and *period* are all qualified; otherwise, the process sends a failure message to the consumers and cancels the ordering list.

4.2 Evaluation Procedure

The following procedure is used for evaluation.

1. *Implementing business processes using VxBPEL*: The variability-based adaptive service composition method proposed in [5] guided us implement three subject programs using VxBPEL. We treat the locations that changes may happen as variation points, and possible choices are identified as variants. Furthermore, constraints are used to specify variation dependencies, which result in consistent and valid process variants (i.e. business scenarios) at run-time. Three VxBPEL programs are derived, as summarized in Table 1.

Table 1. A summary of subject programs

Name	**VxBPEL Lines of Code**[a]	**Number of Web Services**
travel agency	603	8
car estimation	607	7
smart shelf	1146	13

[a]. Measured in lines of XML code.

2. *Deploying and managing VxBPEL processes using MX4B*: Variation definitions and configurations defined in the VxBPEL process improve the adaptation of service compositions. Variation configuration switching is very complex and time-consuming. To facilitate this process, MX4B was developed to provide an efficient way for the deployment, configuration switching, and maintenance of VxBPEL

service compositions [5]. In this experiment, we employed MX4B for the following tasks: (i) deploying VxBPEL process artifacts, (ii) managing VxBPEL processes, and (iii) switching variation configurations at run-time.

3. *Evaluating and comparing performance*: We evaluate two aspects of the performance: (1) for the same business scenarios, we compare the execution time of the BPEL process running on Apache ODE with that of the VxBPEL process running on VxBPEL_ODE; (2) for the same VxBPEL process, we compare overhead of VxBPEL_ODE with that of VxBPEL_ActiveBPEL. The experimental setting is described in Table 2.

Table 2. Experimental setting

CPU	2.40*4 GHz
Memory	8 GBytes
Hard Disk	750GBytes
Operating System	Win7 with 64bit

4.3 Results and Analysis

We summarize the evaluation results and provide an analysis of comparative comparisons next.

(a) *Feasibility*: Through the above evaluation procedure, we observe that VxBPEL_ODE is able to provide an executable context for VxBPEL service compositions. Furthermore, the engine showed a good scalability, namely it is able to process a simple service composition of one Web service, as well as a complex one of 13 Web services, as shown in the case of the *smart shelf* process.

(b) *Performance*: Tables 3, 4 and 5 summarize the performance evaluation results in *ms* of Apache ODE, VxBPEL_ODE, and VxBPEL_ActiveBPEL for three subject programs, respectively. Column “Scenario Name” lists the name of business scenarios implemented using BPEL or VxBPEL; Column “Number of WS” shows the number of Web service involved in the current scenario; Column “A: BPEL by Apache ODE” shows the time cost of BPEL processes using Apache ODE; Column “B: VxBPEL by VxBPEL_ODE” shows the time cost of VxBPEL processes using VxBPEL_ODE; Column “C: VxBPEL by VxBPEL_ActiveBPEL” shows the time cost of VxBPEL processes using VxBPEL_ActiveBPEL; Column “B/A (%)” shows the ratio of Column “B: VxBPEL by VxBPEL_ODE” against Column “A: BPEL by Apache ODE”; Column “B/C (%)” shows the ratio of Column “B: VxBPEL by VxBPEL_ODE” against Column “C: VxBPEL by VxBPEL_ActiveBPEL”. Note that the time cost in Column “A: BPEL by Apache ODE” is the sum of deployment time and execution time of BPEL processes, while the time cost in Columns “B: VxBPEL by VxBPEL_ODE” and “C: VxBPEL by VxBPEL_ActiveBPEL” is the sum of variation switching time and execution time.
From Tables 3-5, we have the following observations:

- For all scenarios in three subjects, the performance of VxBPEL processes is very close to that of BPEL processes. In the case of *travel agency* and *car estimation*, the performance of all process scenarios implemented using BPEL is slightly higher than that implemented using VxBPEL; In case of *smart shelf*, the performance of most process scenarios except the "insufficient" scenario implemented using BPEL is slightly higher than that implemented using VxBPEL. This indicates that variability management does not introduce extra performance overhead. Furthermore, VxBPEL_ODE does not evidently decrease the performance of Apache ODE although some extensions are added. An insight investigation shows that most of time overhead of VxBPEL_ODE is dedicated to processing of standard BPEL elements.

- The performance of VxBPEL_ODE is slightly lower than that of VxBPEL_ActiveBPEL. In the case of *travel agency*, the performance of the former is comparable to that of the latter; In the case of *car estimation*, the performance of the former is higher than the one of the latter for the *simple* scenario, while lower for the *normal* or *expert* scenarios; in the case of *smart shelf*, the performance of the former is lower than that of the latter by around 50%. More interestingly, when the number of involved Web services in a scenario is small, the performance of the former is higher than that of the latter, while the situations change with the increasing number of involved Web services. This performance difference is mainly due to the fact that Apache ODE and ActiveBPEL have different architectures. Apache ODE introduces the serialization of the object models, and integrates with open source components, while ActiveBPEL adopts a coherent architecture and design patterns that significantly improves its performance.

Table 3. Performance evaluation results for the travel agency program

Scenario Name	**Number of WS**	**A:BPEL by Apache ODE** *(ms)*	**B: VxBPEL by VxBPEL_ODE** *(ms)*	**B/A** (%)	**C: VxBPEL by VxBPEL_Active BPEL** *(ms)*	**B/C** (%)
A	3	1508	1611	**106.83**	1435	**112.26**
B	3	1548	1608	**103.88**	1547	**103.94**
C	3	1564	1577	**100.83**	1660	**95.00**
D	3	1501	1607	**107.06**	1604	**100.19**
E	3	1475	1584	**107.39**	1440	**110.00**
F	3	1534	1569	**102.28**	1563	**100.38**

5 Related Work

One category of approaches has turned to Aspect Oriented Programming (AOP) technique [13]. AdaptiveBPEL [14] is a service composition framework which leverages AOP technique to handle various concerns that are separately specified in BPEL processes. The adaptation process is driven by policies, namely a policy mediator is used to negotiate a composite policy and oversee the aspects weaving to enforce the negotiated policy, and a run-time aspect weaving middleware is integrated on top of a BPEL engine. AOBPEL [15] is an aspect-oriented extension to BPEL, which provides a solution for the modularization of crosscutting concerns and supporting dynamic changes in BPEL. AOBPEL specifies extra concerns associated with business processes as aspects, and provides generic aspect constructs. These aspect-oriented approaches can enhance the adaptation of BPEL processes via aspects. However, aspects split up the process logic over different files, which may cause it a difficult task to comprehend the variation, especially when service compositions are complex.

Table 4. Performance evaluation results for the car estimation program

Scenario Name	**Number of WS**	**A: BPEL by Apache ODE** *(ms)*	**B: VxBPEL by VxBPEL_ODE** *(ms)*	**B/A** (%)	**C: VxBPEL by VxBPEL_Active BPEL** *(ms)*	**B/C** (%)
simple	1	441	464	**105.21**	1354	**34.27**
normal	5	2433	2514	**103.33**	1610	**156.15**
expert	5	2505	2524	**100.76**	1561	**161.69**

Table 5. Performance evaluation results for the smart shelf program

Scenario Name	**Number of WS**	**A: BPEL by Apache ODE** *(ms)*	**B: VxBPEL by VxBPEL_ODE** *(ms)*	**B/A** (%)	**C: VxBPEL by VxBPEL_Active BPEL** *(ms)*	**B/C** (%)
default	6	2539	2735	**107.71**	1850	**147.83**
location	7	2753	2803	**101.81**	1845	**151.92**
status	7	2791	2843	**101.86**	1995	**142.50**
locationstatus	8	2768	2897	**104.66**	1844	**157.10**
sufficient	9	2806	2836	**101.07**	2100	**135.05**
insufficient	9	2710	2464	**90.92**	1753	**140.56**
warelocation	10	2752	2777	**100.90**	1872	**148.34**
warestatus	10	2765	2929	**105.93**	1983	**147.71**
warelocation status	11	2866	2932	**102.30**	1980	**148.08**

The other category of approaches is based on the proxy (or broker) mechanism. Trap/BPEL and its predecessors [16] are a family of extensions to BPEL for enhancing robust service compositions through static, dynamic, and generic proxies, respectively. Events such as faults and timeouts during the invocation of partner Web services at run-time are monitored, and the adapted process is augmented with a proxy that replaces failed services with predefined or newly discovered alternatives. wsBus [17] is a framework which is capable of realizing QoS adaptation of service compositions by means of the concept of virtual endpoints. A virtual endpoint is used to select appropriate services based on the attached policy for execution at run-time. All requests are sent to this virtual endpoint and redirected to the real service. The selection of services based on the monitoring data and QoS metrics. SCENE [18] is a service composition execution environment that supports dynamic changes disciplined through rules. The implementation of adaptation is based on the proxy mechanism, which is used to bind the discovered services to the proxy associated with each activity in the BPEL specifications. These proxy-based approaches implicitly implement the adaptation of processes at the messaging/event layer. Since the changes are not treated as first-class objects and variation dependencies are not clearly handled, it may result in variation configuration and maintenance a difficult task.

Unlike the existing efforts, our efforts have been made to achieve the adaptation of BPEL processes through variability management [4-9]. We extended BPEL to provide a set of variability constructs for explicitly specifying variation of service compositions, and the engine presented in this paper provides an executable environment for service composition with variability design. This engine, together with analysis, design, and run-time management tools for VxBPEL, forms an integrated and comprehensive platform, which not only makes the variability-based adaptive service composition approach viable, but also improves its efficiency.

6 Conclusion and Future Work

We have presented a VxBPEL engine, VxBPEL_ODE, by extending an open source BPEL engine, Apache ODE, to enable the execution of VxBPEL processes. VxBPEL_ODE is a core component of an integrated supporting platform which facilitates the adoption of the variability-based adaptive service composition approach. In this paper, we examined the key issues of the design and implementation of such an engine, and validated its effectiveness through case studies. Furthermore, we evaluated the performance of VxBPEL_ODE and compared it with that of VxBPEL_ActiveBPEL. From the experimental results, we observe that VxBPEL_ODE shows a comparable performance of VxBPEL_ActiveBPEL while benefits an integrated design and execution environment for VxBPEL processes.

For future work, we plan to extend VxBPEL to support unplanned changes of service compositions at run-time via the dynamic binding technique, which will accordingly require the further extension of VxBPEL_ODE for binding abstract services with concrete services searched at run-time. We are also interested in examining the variability-based adaptive service composition approach in the development of social network analysis tools that desire the adaptation ability.

Acknowledgment. This research is supported by the National Natural Science Foundation of China (Grant No. 61370061), the Beijing Natural Science Foundation of China (Grant No. 4112037), the Fundamental Research Funds for the Central Universities (Grant No. FRF-SD-12-015A), and the Beijing Municipal Training Program for Excellent Talents (Grant No. 2012D009006000002).

References

1. Papazoglou, M., Traverso, P., Dustdar, S., Leymann, F.: Service-oriented computing: a research roadmap. International Journal on Cooperative Information Systems 17(2), 223–255 (2008)
2. OASIS. Web services business process execution language version 2.0 (2007), `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
3. Aiello, M., Bulanov, P., Groefsema, H.: Requirements and Tools for Variability Management. In: Proceedings of REFS 2010, pp. 245–250. IEEE Computer Society (2010)
4. Koning, M., Sun, C., Sinnema, M., Avgeriou, P.: VxBPEL: Supporting variability for Web services in BPEL. Information and Software Technology 51(2), 258–269 (2009)
5. Sun, C., Wang, K., Xue, T., Aiello, M.: Variability-Based Adaptive Service Compositions (submitted for publication, 2014)

6. Sun, C., Aiello, M.: Towards variable service compositions using VxBPEL. In: Mei, H. (ed.) ICSR 2008. LNCS, vol. 5030, pp. 257–261. Springer, Heidelberg (2008)
7. Sun, C., Rossing, R., Sinnema, M., Aiello, M.: Modeling and managing the variability of Web service-based systems. Journal of Systems and Software 83(3), 502–516 (2010)
8. Sun, C., Xue, T., Hu, C.: Vxbpelengine: A change-driven adaptive service composition engine. Chinese Journal of Computers 36(12), 2441–2454 (2013)
9. Sun, C., Xue, T., Aiello, M.: ValySeC: A Variability Analysis Tool for Service Compositions Using VxBPEL. In: Proceedings of APSCC 2010, pp. 307–314 (2010)
10. ActiveBPEL, Active Endpoints (2007), http://www.activebpel.org
11. Apache, Apache ODE (2006), http://ode.apache.org/
12. Sinnema, M., Deelstra, S., Nijhuis, J., Dannenberg, R.B.: COVAMOF: a framework for modeling variability in software product families. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 197–213. Springer, Heidelberg (2004)
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
14. Erradi, A., Maheshwari, P.: AdaptiveBPEL: a Policy-Driven Middleware for Flexible Web Services Compositions. In: Proceedings of International Workshop on Middleware for Web Services (MWS 2005), pp. 5–12 (2005)
15. Charfi, A., Mezini, M.: AO4BPEL: An Aspect-Oriented Extension to BPEL. World Wide Web Journal 10(3), 309–344 (2007)
16. Ezenwoye, O., Sadjadi, S.M.: TRAP/BPEL-A Framework for Dynamic Adaptation of Composite Services. Proceedings of WEBIST (1), 216–221 (2007)
17. Erradi, A., Maheshwari, P.: wsBus: QoS-aware middleware for reliable web services interaction. In: Proceedings of EEE 2005, pp. 634–639. IEEE Computer Society (2005)
18. Colombo, M., Di Nitto, E., Mauri, M.: SCENE: a service composition execution environment supporting dynamic changes disciplined through rules. In: Dan, A., Lamersdorf, W. (eds.) ICSOC 2006. LNCS, vol. 4294, pp. 191–202. Springer, Heidelberg (2006)