PAPER

AdaFF: Adaptive Failure-Handling Framework for Composite Web Services

Yuna KIM[†a)], ***Student Member***, **Wan Yeon LEE**[††], **Kyong Hoon KIM**[†††], ***Members***, ***and*** **Jong KIM**[†], ***Nonmember***

SUMMARY In this paper, we propose a novel Web service composition framework which dynamically accommodates various failure recovery requirements. In the proposed framework called *Adaptive Failure-handling Framework (AdaFF)*, failure-handling submodules are prepared during the design of a composite service, and some of them are systematically selected and automatically combined with the composite Web service at service instantiation in accordance with the requirement of individual users. In contrast, existing frameworks cannot adapt the failure-handling behaviors to user's requirements. AdaFF rapidly delivers a composite service supporting the requirement-matched failure handling without manual development, and contributes to a flexible composite Web service design in that service architects never care about failure handling or variable requirements of users. For proof of concept, we implement a prototype system of the AdaFF, which automatically generates a composite service instance with Web Services Business Process Execution Language (WS-BPEL) according to the users' requirement specified in XML format and executes the generated instance on the ActiveBPEL engine.

key words: *composite Web service, adaptive failure-handling, dynamic workflow generation, WS-BPEL*

1. Introduction

A Web service is a programmable module published with a standard interface description that defines the access methods of the service [1]. In order to produce a new value-added service for a special purpose, multiple Web services that are published by different participants are combined into a composite Web service [2]–[4]. With the support of composite Web services, users enjoy rich functionality and convenience and enterprises save the burden of development by outsourcing most component services to other companies. To efficiently support composite Web services, many researchers have investigated on effective composition of component Web services [5]–[7], collaboration of numerous participants [8], [9], security enforcement of separate participants [10], [11], and failure handling [12]–[17]. This paper focuses on the enhancement of failure handling mechanisms.

Most previous Web service failure-handling mechanisms considered fast restart [18], consistent service results [13], [19], reliable completion when failures unexpectedly happen [14], [17], and coordinated recovery of concurrent errors [12], [20]. However, the previous failure-handling mechanisms have rarely considered the notion of *adaptive failure-handling*, i.e., the capability to dynamically provide different recovery flow against the same failure in accordance with the requirement of individual users.

The adaptation concept has been brought to adaptive service composition for producing a context-aware composite service with the best-matching component services [21], [22], but it does not support adaptively changing the recovery flow when a failure occurs. Adaptive failure-handling is needed in such environments where users may require their own recovery strategy on a composite service even after completing the composition with given components. In this paper, we propose a new failure-handling framework, which dynamically generates the requirement-matched failure recovery strategies and systematically combines the strategies into an established composite service at every request to accommodate various failure handling requirements of individual users.

The main contribution of this study is to propose a novel service composition framework with adaptive failure handling, called *Adaptive Failure-handling Framework (AdaFF)*. The AdaFF is aimed at generating a composite Web service which handles failures that are encountered during the execution of a composite service. AdaFF receives the user requirement on the failure handling at request time, and automatically generates the failure recovery flow matching with the requirement. The generated recovery flow is systematically combined with the normal operation flow to produce an on-demand workflow for each requester. To the best of our knowledge, the proposed framework is the first systematic approach to reflect the arbitrary failure-handling requirements of users by changing a failure recovery flow of composite services dynamically.

Another contribution of this paper is to implement a prototype system which verifies the concept of AdaFF working completely and gives a development guideline. The prototype system receives the respective failure-handling requirement of users via an XML schema, generates a workflow containing the requirement-matched failure recovery flow with Web Services Business Process Execution Language (WS-BPEL)[*], and executes the generated workflow on the ActiveBPEL engine.

Manuscript received April 28, 2009.
Manuscript revised October 18, 2009.

[†]The authors are with the Dept. of Computer Science and Engineering, Pohang University of Science and Technology, San 31 Hyoja Dong, Pohang, 790–784, South Korea.
[††]The author is with the Dept. of Computer Engineering, Hallym University, Chunchon 200–702, South Korea.
[†††]The author is with the Dept. of Information Science, Gyeongsang National University, Jinju 660–701, South Korea.
a) E-mail: existion@postech.ac.kr

DOI: 10.1587/transinf.E93.D.834

[*]http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf

The remainder of this paper is organized as follows. Section 2 presents an example of the problem that motivates this work, and Sect. 3 briefly surveys related previous work. Section 4 describes preliminaries of the proposed framework, which is described with its structure and formal expression in Sect. 5. Section 6 describes implementation details of the prototype system, and we evaluate the prototype with a travel package booking scenario in Sect. 7. Finally Sect. 8 gives concluding remarks.

2. Motivating Example

A composite service is implemented by means of a *workflow*, which contains both normal operation flow and failure recovery flow of component services. For example, a composite service for a travel package booking is a workflow of flight scheduling, hotel reservation, and car rental services (Fig. 1 (a)). In this example, two users request different recovery flows against the same failure of a hotel reservation:

- $Flow_1$: to repeat the request procedure (H) until the success of the reservation, and to release the other two reservations (F and C) at failure even after a limited number of repeats. This user requests either a complete or no reservation (Fig. 1 (b)).
- $Flow_2$: to stop the request procedure (H) and subsequently to proceed to the next procedure (P) with partial reservation results from F and C. This user requests all possible partial reservations (Fig. 1 (c)).

Even though users request different recovery flows, existing Web-service failure-handling frameworks allow only an identical recovery flow against the same failure [12], [16], [23], [24]. Such requirement-unmatched recovery flow results in 1) users cannot receive a worthy result of the service in time, 2) reserved resources are released unintentionally, or 3) service providers perform wasteful operations.

If $Flow_1$ is provided to the user who demands $Flow_2$, the continuous failure of hotel reservation ultimately prevents the user from obtaining a partial but worthy result and leads the service provider to perform wasteful unintended operations of repeat and release. The successfully reserved flight and car are released unintentionally, although the success would not be guaranteed in the next try. On the other hand, if $Flow_2$ is provided to the user who demands $Flow_1$, the instantaneous failure of hotel reservation produces an incomplete and worthless result. Reserved resources which are worthless to the user are not available to other users until the release of them. The user would have received a complete and worthy result if the hotel reservation request had been retried a couple of times in the first place.

Timely completion of composite services without unintended release of reserved resources becomes more important as reservation services of limited resources from crowded requests increase, e.g., online registration of popular lectures, high-demanded flight reservation, or blood reservation in a hospital transfusion service.

Static generation of all possible recovery flows per a composite service enables users to select the requirement-matched failure recovery flow without delay. However, it is difficult to determine an entire set of failure recovery flows when designing a composite service, because the failure recovery flows depend on component services and several failure recovery strategies. Moreover, most of them might not be used. Consequently, in this paper, we propose a new failure-handling framework, which dynamically generates the failure recovery flow that matches the user's requirements and systematically combines the flow into the workflow at every request to accommodate various failure handling requirements of individual users.

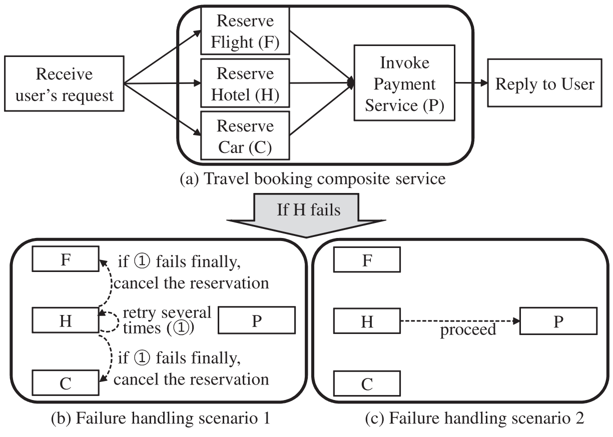

Receive user's request
Reserve Flight (F)
Reserve Hotel (H)
Reserve Car (C)
Invoke Payment Service (P)
Reply to User
(a) Travel booking composite service
If H fails
F
if ① fails finally, cancel the reservation
H
retry several times (①)
P
if ① fails finally, cancel the reservation
C
(b) Failure handling scenario 1
F
H
proceed
P
C
(c) Failure handling scenario 2

Fig. 1 A composite Web service for travel booking, with two different scenarios on failure handling. Solid arrows represent normal operation flows, while dashed arrows do failure recovery flows.

3. Related Work

While adaptive service compositions have been considered for efficient generation of normal operation flow [21], [22], the notion of adaptive failure-handling has never been considered for efficient generation of failure recovery flow. The adaptive service composition aims at searching for the best set of component services based on user's context, and supporting the low-cost modification of the services at runtime. This mechanism cannot be applied directly to failure-handling because the working principles and requirements of failure recovery flow are different from those of normal operation flow.

In existing failure-handling mechanisms of composite Web services, like WS-BPEL, Web Services Composition Action Language (WSCAL) [12], and Fung et al.'s work [23], failure recovery flow is implemented dependently on normal operation flow in a composite service. WS-BPEL is a process-oriented composition language used to implement normal-operation flow of component services and its dependent failure-recovery flow in an XML-based grammar [3], [25]. WSCAL is an XML-based language that defines how participants coordinate to deal with a failure occurring at a given normal flow. Fung et al. [23] proposed how to identify what failure is critical to successful termination of a normal-flow process developed using WS-BPEL, and then how to modify the process for recovery of the failure. This modification involves change of both failure recovery and normal operation flows. In these three mechanisms, the failure recovery flow cannot be easily detached from a composite Web service, so the entire workflow needs to be redesigned in order to change failure recovery flow.

There were attempts to implement failure recovery flows independently of normal operation flows, like in THROWS architecture [16] and WSTx framework [24]. In THROWS architecture, failure recovery flow is determined by selecting a set of component services. In order to change the failure recovery flow, only the selection needs to be modified. WSTx middleware framework specifies what outcome is expected for success of the service. In order to change the failure recovery flow, the specification of outcome conditions needs to be modified. In these two methods, it is possible to modify only the failure recovery flow while keeping the normal operation flow intact. However, they limit failure recovery flow to only one strategy, compensation, so the modification of failure recovery flow means only a change of either services that should be compensated or the compensation sequence.

Our framework exploits a method that implements failure recovery flow separately from normal execution flow, because this separation is efficient for changing the recovery flow at runtime. Furthermore, the best failure recovery flow in a composite service is generated automatically in order to satisfy the user's requirements. The framework supports diverse failure recovery strategies, including service retry, service replacement, failure negligence, as well as compensation based on failure dependency.

4. Preliminaries

To facilitate the understanding of the proposed framework, this section gives preliminary background knowledge such as the operational procedure of composite service, the types of failures occurring in composite services, and the failure recovery techniques used in the proposed framework.

4.1 Composite Service

A composite service is implemented by exploiting existing *individual* services as building blocks, and the individual services can be accessed through SOAP using the XML messages whose format is specified in the services' WSDL documents. SOAP is a protocol for exchanging XML-based messages over HTTP, and WSDL is an XML-based interface description language used to describe Web services.

A process to build a composite Web service with existing individual services is called *service composition*, which is performed through two procedures: *workflow generation* and *service binding* [26]. Workflow generation constructs an execution order of the basic functions and specifies data communications between them. Service binding selects one of individual Web services that perform the matched functions within the workflow. The generated composite service is executed and managed by a middleware program, called *orchestration engine*.

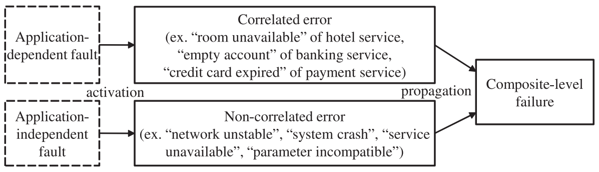

Application-dependent fault
Application-independent fault
activation
Correlated error (ex. "room unavailable" of hotel service, "empty account" of banking service, "credit card expired" of payment service)
Non-correlated error (ex. "network unstable", "system crash", "service unavailable", "parameter incompatible")
propagation
Composite-level failure

Fig. 2 Origins of composite-level failure.

4.2 Composite-Level Failure

We recognize all events deviating the delivered output of a composite service from its expected output as failures, whereas previous methods [27], [28] consider only the events in individual services. Our composite-level failure model is a superset of the individual-level failure model. In other words, even successful termination of individual services can cause a composite-level failure if the outcome of the individual service does not match the expected goal or quality of service.

We classify the origins of the composite-level failures (Fig. 2) into *application-dependent faults* and *application-independent faults*, and manipulate them differently. Both of the faults are activated when the output is not matched with the expected goal of the composite service, and the activation is identified as an error — *correlated* or *non-correlated*. Correlated error is deliberately exposed by the component services so that a composite service should handle this error with regard to other embedded components, while non-correlated error is unintendedly exposed and can be handled independently of the composite service.

4.3 Failure Recovery

There are two techniques for failure recovery: backward recovery [29], [30] and forward recovery [12], [31]. Backward recovery rolls back to the former correct state of services prior to its execution like a compensation. Forward recovery transforms services into any correct state like exception handling. These two recoveries complement each other in that the backward recovery cannot exclude the repeated failures and the forward recovery cannot guarantee the exact previous state. Hence, a combination of these two recovery techniques is needed for supporting various user requirements.

5. Proposed Framework

In the proposed service composition framework, a normal operation flow of abstract function units is designed in a business process without a failure recovery flow. After a request comes from a user with his/her requirement on failure recovery, a failure recovery flow is generated into the business process so that the actual recovery behavior of the process satisfies the user's requirement. In order to make the

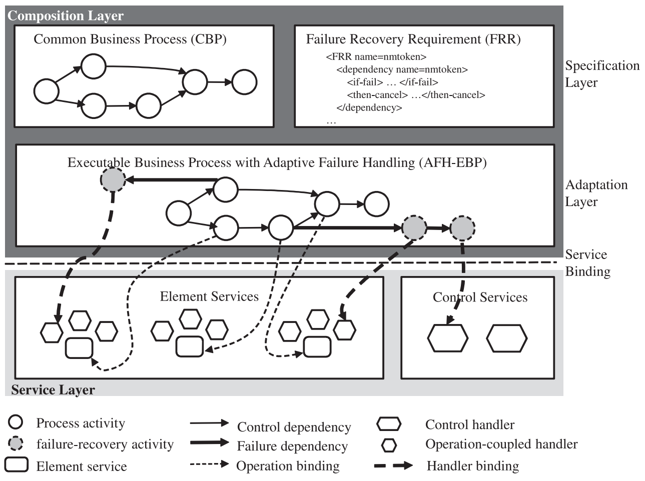


Fig. 3 Web Services composition framework for adaptive failure handling.

process executable, all of the abstract functions in the process are bound to real Web services which actually perform the matched functions.

The proposed framework consists of two layers, *composition layer* and *service layer*, and a *service binding* that links these two layers (Fig. 3). Primitive services in the service layer are composed into a composite service in the composition layer via the service binding. In this section, we will describe all the components of the framework in detail.

5.1 Composition Layer

The composition layer is divided into two sub-layers: *specification layer* and *adaptation layer*. The specification layer includes *Common Business Process (CBP)* and *Failure Recovery Requirement (FRR)*. CBP is designed by the developer of a composite service, and FRR is specified by a user of the composite service. The adaptation layer includes *Executable Business Process with Adaptive Failure Handling (AFH-EBP)*, which is dynamically generated with the CBP and the respective FRR, and provided to the user.

CBP is a normal operation flow of business activities, each of which represents an abstract unit of atomic function in a business process. The normal flow represents control dependency and data dependency between activities. Completion of execution of an activity is followed by the execution of another activity, which is called the *control dependency* between the activities. The outcome of an activity is consumed by another activity, which is called *data dependency* between the activities.

FRR is a requirement of failure recovery for the particular CBP. FRR can be expressed as an XML schema as shown in Fig. 4, which is effective in specifying the requirements by items with a formal structure and in parsing them due to the property of XML. It manifests three items related to failure recovery.

```
1  <FRR name = "qname">
2     <bestTry activity = activityID/>*
3     <ignoreFailure activity = activityID/>*
4     <dependency name = "ncname">
5         <if-fail> activityID </if-fail>
6         <then-cancel> activityID </then-cancel>
7     </dependency>*
8  </FRR>
```

Fig. 4 XML syntax of failure recovery requirement (FRR). Character "∗" appended to the end of the elements means that the elements may appear zero or more times.

- **Failure resistance** means that a failure of an activity is followed by retrying a semantically identical service until successful execution. If users demand a complete execution of an activity, they specify the name of the activity in the element <*bestTry*> in line 2.
- **Failure negligence** means that a failure of an activity is ignored and the process execution proceeds to the subsequent flow. If users are willing to endure a failure of an activity, they specify the name of the activity in the element <*ignoreFailure*> in line 3.
- **Failure dependency** means that a failure of an activity is followed by having another activity to roll back to the original state prior to the execution. If users demand to invalidate *activity*$_1$'s completion upon *activity*$_2$'s failure, they specify the names of the two activities in the element <*dependency*> in line 4: *activity*$_2$ in <*if-fail*> and *activity*$_1$ in <*then-cancel*>.

Users can express a variety of recovery strategies with combination of the above three items. For example, by combining *failure negligence* and *failure resistance*, an activity is tried several times until its success and then its failure is finally ignored. By combining several *failure dependencies*, an activity's failure affects a group of activities. XML schema of FRR designed in our prototype will be presented in Sect. 6.

AFH-EBP is an executable business process representing both normal operation flow and failure recovery flow of component services. It is produced through two steps: a failure recovery flow associated with a user's FRR is added into a particular CBP chosen by the user, and then all abstract activities are bound to Web services which perform the matched function.

Let us explain how to transform a CBP to an AFH-EBP from perspective of an activity in Fig. 5. In CBP, activities are linked with each other via *in_flow* and *out_flow*, which convey data and control between activities as shown in Fig. 5 (a). In AFH-EBP, as shown in Fig. 5 (b), *failure-recovery activities (FRA)*, *in_hflow*, and *out_hflow* are added for a failure recovery flow associated with a user's FRR. Each *FRA* represents an abstract unit of atomic function required for failure recovery, and each *in_hflow* and *out_hflow* represents control/data dependency involving *FRAs*. In the second step, activities of CBP are bound to component services through *in_link* and *out_link*. Through the links, the composite service invokes the component services and receives the result from them. *in_link* conveys the result data

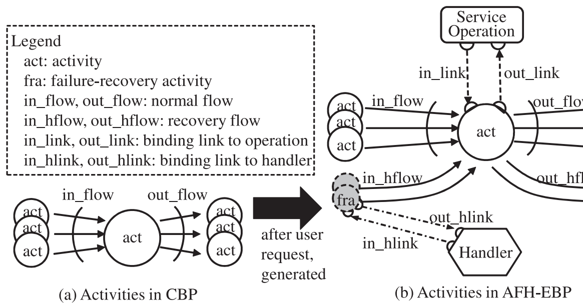

Legend
act: activity
fra: failure-recovery activity
in_flow, out_flow: normal flow
in_hflow, out_hflow: recovery flow
in_link, out_link: binding link to operation
in_hlink, out_hlink: binding link to handler
in_flow
out_flow
act
after user request, generated
(a) Activities in CBP
Service Operation
in_link
out_link
in_hflow
out_hflow
fra
out_hlink
in_hlink
Handler
(b) Activities in AFH-EBP

Fig. 5 Modeling of activities in *CBP* and *AFH-EBP*.

Table 1 Formal expression of CBP and AFH-EBP.

CBP $B = (A, F)$
- $A = \{A_1, A_2, ..., A_n\}$ is a set of activities A_i.
- A_i = (*functional_description, semantic_description*)
- $F = \{F_1, F_2, ..., F_m\}$ is a set of flows F_i.
- $F_i = (A_p, A_q, Parm_i, Cond_i)$ is a flow from A_p to A_q with the parameter $Parm_i$, being activated when $Cond_i$ is satisfied.

AFH-EBP $E(B) = (B, Bind, FRA, FRF, FRBind)$
- B is the associated CBP including A and F.
- $Bind = \{(A_i, OP_i) \mid A_i \in A$ is bound to i-th operation OP_i of services$\}$
- $FRA = \{FRA_1, FRA_2, ..., FRA_k\}$ is a set of failure-recovery activities FRA_i.
- $FRF = \{FRF_1, FRF_2, ..., FRF_k\}$ is a set of failure-recovery flows FRF_i.
- $FRF_\mathrm{i} = ([FRA|A]_p, [FRA|A]_q, HParm_i, HCond_i)$ is a failure-recovery flow from FRA_p or A_p to FRA_q or A_q with the parameter $HParm_i$, being activated when $HCond_i$ is satisfied.
- $FRBind = \{(FRA_i, H_i) \mid$ a failure-recovery activity FRA_i is bound to a failure handler $H_i\}$, where H_i is selected from either $H(OP_j)$ or $H(B)$.
- $H(OP_i) = [H(OP_i, restore) \mid H(OP_i, abort)]$ is a handler coupled with OP_i.
- $H(OP_i, restore)$ is a restoration handler coupled with OP_i, which rolls back to the previous state as before executing OP_i.
- $H(OP_i, abort)$ is an abortion handler coupled with OP_i, which stops on-going processing OP_i.
- $H(B) = [H(rebind) \mid H(retry) \mid H(ignore)]$ is a control handler that can be associated with any activities.
- $H(rebind, A_i)$ is a rebinding handler, which finds an alternative service for the activity A_i and binds it to the A_i.
- $H(retry, A_i)$ is a retrying handler, which tries the activity A_i again.
- $H(ignore, A_i)$ is an ignoring handler, which ignores the result of activity A_i.

of the service operation to the activity in CBP and *out_link* conveys the input data required for execution of the service operation from the activity in CBP. FRAs are also bound to failure handlers through *in_hlink* and *out_hlink*. The component services and the failure handlers are provided in the service layer.

5.2 Service Layer

The service layer is divided into two parts: *Element Services* and *Control Services*. Element services provide basic business functions that can be executed autonomously in an independent administrative domain. Each element service is published with a WSDL document which describes its own operations, possible application-dependent faults, and operation-coupled handlers that restore to the former state of the service. The handlers are needed to be directly given by the provider of the component services for backward recovery, because such service-state sensitive recovery is impossible without detailed information about working inside the operations. Unlike the element services, the control services provide control handlers that perform failure handling regardless of the service state. The handlers for forward recovery do not need any detailed information about working inside the operations of component services.

Element services and failure handlers are bound to normal activities and failure-recovery activities in an AFH-EBP, repsectively. Specifically, for *failure negligence* in FRR, an *ignoring handler* is bound to an FRA, which enforces the succeeding activity to ignore the outcome of the failed preceding activity. For *failure resistance* in FRR, a *rebinding handler* and a *retrying handler* are bound to FRAs in succession. Rebinding handler finds an alternative service for the preceding activity instead of the failed service and binds it to the activity. Retrying handler invokes the service again with the same inputs. For *failure dependency* in FRR, an operation-coupled handler is bound to an FRA, which restores to the former state of the coupled service operation.

5.3 Formal Expression of CBP and AFH-EBP

We present formal expressions of CBP and AFH-EBP as shown in Table 1. CBP is defined as an abstract process with constructs of BPEL (Business Process Execution Language), which is the most widely adopted standard for specifying execution of business process. Such BPEL constructs are easily modeled with a graphical design tool, and facilitate the generation of an AFH-EBP instance. The formal expressions will be used for describing an algorithm that generates AFH-EBP (Algorithm 1). In CBP, *functional_description* and *semantic_description* of each activity represent which services can be matched with the activity functionally and semantically, respectively.

6. Implementation

In this section, we present an implementation of the prototype that provides the composite service which adapts its failure handling behavior to the individual failure recovery requirement of users. This implementation is done as a proof-of-concept of our proposed framework.

The prototype is implemented on web/application server, Apache Tomcat 5.5, and Java SDK 1.4. We select WS-BPEL as the business process execution language, which is the most widely adopted standard. Our prototype can be divided roughly into two parts, *Preprocessor* and *Process Orchestrator* (Fig. 6). Preprocessor is involved in generation of an AFH-EBP instance with a CBP and an FRR at user's request, and the Process Orchestrator is involved in execution of the AFH-EBP instance generated by the Preprocessor.

6.1 Preprocessor

The Preprocessor consists of *Client Interface*, *FRR Interpreter*, and *AFH-EBP Generator*. First of all, through *Client Interface*, a user explores abstract information about CBPs

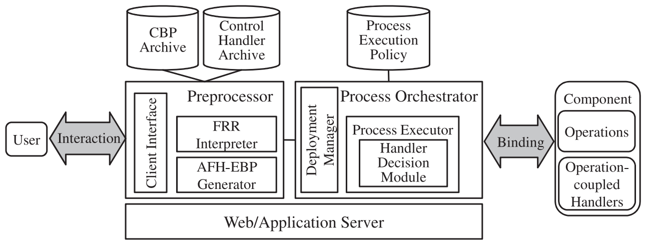


Fig. 6 A prototype of a composite service providing system for adaptive failure handling under the proposed framework.

Algorithm 1 AFH-EBP generation with CBP and FRR

```
PROCEDURE AFH-EBP_GENERATE (cbp B, afh-ebp E)
 /* CBP B = (A, F), AFH-EBP E = (B, Bind, FRA, FRF, FRBind)
    $A_i$.try_num: the number of tries of an activity $A_i$
    max_retry: the maximum number of times to retry activities */
$k \leftarrow 0$ /* index of failure-recovery activities */
for $\forall A_i \in A$ do
   if there exists bestTry[@activity=$A_i$.name] then /*$A_i$.name: the name of an activity $A_i$ */
         Add $FRA_{k+1}$ and $FRA_{k+2}$ to FRA.
         Add ($A_i$, $FRA_{k+1}$, $A_i$.ServiceInfo, failure occurs && $A_i$.try_num $\leq$ max_retry) to FRF.
         Add ($FRA_{k+1}$, H(rebind, $A_i$)) to FRBind.
         Add ($FRA_{k+1}$, $FRA_{k+2}$, $A_i$.input_parameters, binding success) to FRF.
         Add ($FRA_{k+2}$, H(retry, $A_i$)) to FRBind.
         $k \leftarrow k + 2$
   endif
   if there exists ignore[@activity=$A_i$.name] then
         ADD_RECOVERY_FLOW($A_i$, E, k).
         Add ($FRA_k$, H(ignore, $A_i$)) to FRBind.
   endif
   for $\forall A_j \in$ dependency[source=$A_i$.name] do
         ADD_RECOVERY_FLOW($A_i$, E, k).
         Add ($FRA_k$, H($OP_j$)) to FRBind . /* assert ($A_j$, $OP_j$) $\in$ Bind. */
   endfor
endfor
PROCEDURE ADD_RECOVERY_FLOW (src_act SA, afh-ebp E, int num_fra)
Add $FRA_{++num\_fra}$ to FRA.
if there exists bestTry[@activity=SA.name] then
    Add (SA, $FRA_{num\_fra}$, null, failure occurs && SA.try_num > max_retry) to FRF.
else
    Add (SA, $FRA_{num\_fra}$, null, failure occurs) to FRF.
endif
```

stored in *CBP Archive*, and selects one of them complying with the demands. Then the component activities are listed with checkboxes to select failure handling strategies. When the user pushes the submit button after selecting strategies, an FRR document is automatically produced in XML format and given to *Preprocessor* in the prototype system.

FRR Interpreter processes XPath[†] query expressions in the FRR and is implemented using Java XPath library. It receives an XPath query from the AFH-EBP Generator and gives the query result back. The query expressions can be seen from the statements marked with an underline in Algorithm 1. Note that the schema of FRR can be extended to WS-Policy for manifesting the requirements more explicitly.

AFH-EBP Generator dynamically generates an AFH-EBP instance, in the format of WS-BPEL, with the FRR given by a user and the associated CBP. It implements the pseudo-algorithm of *AFH-EBP-Generate* (Algorithm 1). In the algorithm, A_i`.try_num` is a variable initialized to zero and increased by one at every trial for each activity within an instance, and `max_retry` is an invariable threshold value. The algorithm works on every activity in the CBP by checking whether the activity is declared in each item of the FRR.

- Activity A_i declared in <*bestTry*> is required to be linked to two newly created FRAs in serial. The fore FRA is bound to a rebinding handler that rebinds an alternative service and the rear FRA is bound to a retrying handler that invokes the service with the same input as for activity A_i.
- Activity A_i declared in <*ignoreFailure*> is required to be linked to a newly created FRA, which is bound to an ignoring handler that makes the succeeding activities ignore the outcome of the activity A_i.
- *If-fail* activity A_i in <*dependency*> is required to be linked to a newly created FRA. The FRA is bound to an operation-coupled handler provided by the service to which the *then-cancel* activity A_j is bound. The handler carries out the restoration or the abortion of the destination activity A_j.

Rebinding handler, retrying handler, and ignoring handler are maintained in the *Control Handler Archive*. All the handlers are developed by the composite service provider itself or imported from remote registries. In this algorithm, we do not present how to select appropriate services to business activities of CBP from Web service registries, because this is not our focus. Such an effort is found in [4]–[7]. Our focus is not on adaptivity in service selection, but on adaptivity in failure recovery flow, i.e. which handlers would be invoked in which order upon a certain failure according to the requirement of individual users.

Finally the Preprocessor obtains an AFH-EBP instance in the format of WS-BPEL. Failure-recovery activities and failure-recovery flows are expressed by <*compensationHandler*> and <*faultHandler*> in the WS-BPEL standard specification.

6.2 Process Orchestrator

Process Orchestrator consists of *Deployment Manager* and *Process Executor*. *Deployment Manager* deploys the AFH-EBP instance passed from the Preprocessor to an ActiveBPEL server, by making a business process archive (.bpr) file. This archive file includes: a BPEL of AFH-EBP process (.bpel) generated by the Preprocessor; a process deployment descriptor (.pdd) file that describes partner link details, persistence and versioning information, process directives and indexed properties, and other details.

Process Executor starts executing the AFH-EBP instance according to a process specification. It is implemented using ActiveBPEL engine 2.0, which is an open-source business process execution engine. Into this engine, we added *Handler Decision Module*, which chooses the right type of operation-coupled handlers based on the running state of them. The operation-coupled handler has two types: restoration handler and abortion handler. When the AFH-EBP invokes an operation-coupled handler of a service for compensating the service, the Handler Decision

[†]http://www.w3.org/TR/xpath20/

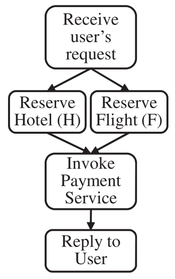
Module chooses a restoration handler if the service has been already committed, or chooses an abortion handler if the service is on execution. It is assumed that the execution engine works normally only when the input WS-BPEL specification has no error, and cannot deal with any exceptional situation of the engine itself.

In addition, the Process Executor refers to the *Process Execution Policy*, which maintains policies to be applied during execution of business processes, such as maximum threshold number of times that services can be retried. Such policies can be configured before running the Process Orchestrator.

7. Evaluation

We evaluated our prototype in two phases: generation of failure-recovery flow according to the FRR in *AFH-EBP Generator* and execution of AFH-EBP instances in *Process Executor*. The evaluation was performed under the environment such that two users request different FRRs on the same travel booking service and both application-dependent failures and application-independent failures are exposed during execution of the service.

The example travel booking service performs hotel and flight reservation according to the itinerary of a user and then sends the result to the user, as shown in Fig. 7 (a). *User A* offers an FRR (Fig. 7 (b)), which has the following semantics: i) any flight should be reserved, and ii) if no flight is available, hotel reservation is unnecessary. *User B* offers an FRR (Fig. 7 (c)), which has the following semantics: do not care about the failure of hotel reservation.



(a) CBP for travel agency

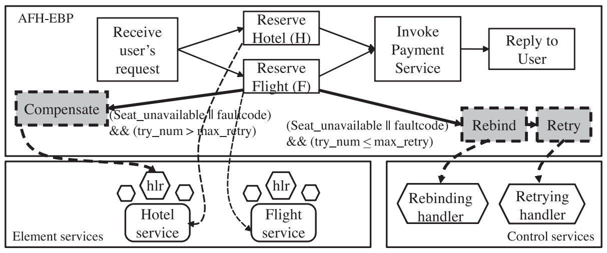
```
<FRR name = TravelAgencyUserA>
  <bestTry activity = Reserve flight/>
  <dependency name = FHDependency>
    <if-fail> Reserve flight</if-fail>
    <then-cancel> Reserve hotel </then-cancel>
  </dependency>
</FRR>
```

(b) FRR of *user A*

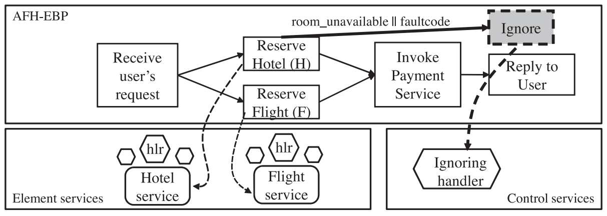
```
<FRR name = TravelAgencyUserB>
  <ignoreFailure activity = Reserve hotel/>
</FRR>
```

(c) FRR of *user B*

Fig. 7 Examples of one CBP and two FRRs.



(a) AFH-EBP instance for *user A*



(b) AFH-EBP instance for *user B*

Fig. 8 Examples of two AFH-EBPs generated from the CBP of travel agency and two users' FRRs as given by Fig. 7.

7.1 Generation of AFH-EBP Instances

AFH-EBP Generator generates two different AFH-EBP instances according to each FRR, as shown in Fig. 8.

1) AFH-EBP instance for *user A*: Due to <*bestTry*> declared in the FRR, if an application-dependent failure named *seat unavailable* or an application-independent failure named *faultcode*[†] occurs in the activity *F* and the number of tries is less than or equal to the maximum threshold then a *Rebind* activity finds an alternative flight service and binds the activity *F* to the service by using *rebinding handler*. Subsequently a *Retry* activity executes the activity *F* again by using *retrying handler*. Due to <*dependency*> declared in the FRR of *user A*, if the failure occurs and the number of retries is more than the maximum threshold, the *Compensate* activity compensates the *Hotel Service* by using the operation-coupled handler of the *Hotel Service*.

2) AFH-EBP instance for *user B*: Due to <*ignoreFailure*> declared in the FRR, if an application-dependent failure named *room unavailable* or an application-independent failure named *faultcode* occurs in the activity *H*, an *Ignore* activity forces the *Payment* activity to ignore the input from the activity *H* by using *ignoring handler*.

7.2 Execution of AFH-EBP Instances

Process Executor executes two AFH-EBP instances under diverse situations as follows:

1. Without any failure in all component services, both instances are successfully terminated with the complete reservation results on hotel and flight and the payment receipt.
2. During execution of the first instance (Fig. 8 (a)), a flight service was down. *Process Executor* detects the 'service unavailable' failure after timeout. This failure activates the failure recovery flow. The rebinding handler finds an alternative service from a service community and rebinds the service to the flight reservation activity. The retrying handler invokes the service with the same inputs. As a result, the instance gives a user the payment receipt of hotel and flight reservations.
3. During execution of the first instance (Fig. 8 (a)), a flight service was down, all alternative services were

[†]*faultcode* is designated by the composite service provider.

set to expose 'seat unavailable,' and `max_retry` was set to one, for evaluating the service compensation. After the maximum retires, 'seat unavailable' failure occurs as expected, and the restoration handler restores the state of the hotel service as before its execution. As a result, any reservations are not made and the user gets the result of no reservation.

4. During execution of the second instance (Fig. 8 (b)), a hotel service was set to expose 'room unavailable.' The ignoring handler consumes the failure and transfers the empty reservation result to the payment service. The payment service charges the cost of flight reservation, and the user finally gets only the flight reservation with the payment receipt.

7.3 Performance Overhead

In this system, a penalty on service completion time is incurred at service instantiation after user's request, because failure-recovery flow is dynamically generated according to the requirement of each user at every request. This penalty is not incurred in previous systems where all service instances provide the identical failure-recovery flow against the same failure. However, the penalty can be overcome during execution of service instances, with our requirement-matched failure handling. For example, for *user B* as in Fig. 7 (c), our system immediately gives the partial and worthy result, only the flight reservation data, to the user in the first try, even if a hotel service fails. On the contrary, the previous systems repeat the request to the hotel service several times, and then release the flight reservation. After all, the user should retry another service to get the desired result. Accordingly, the requirement-matched failure handling eliminates the need for another try, which can reduce the completion time for users to obtain the desired result.

In addition, the flow generation process at service instantiation can be accelerated by preparing the default failure-recovery flow and adding or subtracting the differential flow, and optimizing the XML processing associated with FRR interpretation. The runtime overhead of orchestrating the failure-recovery flow can be reduced by preparing the failure handlers in *Control Handler Archive* and collecting the alternative services of the same semantics in service community for fast handling.

8. Concluding Remarks

In this paper we proposed a new Web service composition framework which supports the dynamic adaptation of failure handling behavior of a composite service to the failure recovery requirement of each user. Adaptive failure handling reduces the handling cost by eliminating unnecessary handling of the failure that is acceptable to users. The proposed framework also contributes to a flexible service design by which an architect of composite services would never care about failure handling or versatile requirements of users. We also implemented a prototype based on our framework for verifying the adaptive failure handling. Our experiences are useful for those interested in developing their composite Web services. In future work, we will investigate a way of adjusting the failure handling behavior according to the failure's origin and the service environment at runtime.

Acknowledgments

This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2010-C1090-1031-0009).

References

[1] C. Ferris and J. Farrell, "What are web services?," Commun. ACM, vol.46, no.6, p.31, 2003.
[2] M.N. Huhns and M.P. Singh, "Service-oriented computing: key concepts and principles," IEEE Internet Comput., vol.9, no.1, pp.75–81, 2005.
[3] C. Peltz, "Web services orchestration and choreography," IEEE Computer, vol.36, no.10, pp.46–52, Oct. 2003.
[4] N. Milanovic and M. Malek, "Current solutions for web service composition," IEEE Internet Comput., vol.8, no.6, pp.51–59, 2004.
[5] B. Benatallah, Q.Z. Sheng, and M. Dumas, "The self-serv environment for web services composition," IEEE Internet Comput., vol.7, no.1, pp.40–48, 2003.
[6] R. Akkiraju, B. Srivastava, A.A. Ivan, R. Goodwin, and T. Syeda-Mahmood, "Semaplan: Combining planning with semantic matching to achieve web service composition," IEEE Intl. Conf. on WS, pp.37–44, 2006.
[7] P.P.W. Chan and M.R. Lyu, "Dynamic web service composition: A new approach in building reliable web service," 22nd Intl. Conf. on AINA, pp.20–25, 2008.
[8] U. Yildiz and C. Godart, "Information flow control with decentralized service compositions," IEEE Intl. Conf. on WS, pp.9–17, 2007.
[9] M. Stollberg, U. Keller, and D. Fensel, "Partner and service discovery for collaboration establishment with semantic web services," IEEE Intl. Conf. on WS, pp.473–480, 2005.
[10] M. Pietro, C. Bruno, S. Swaminathan, and B. Elisa, "Xacml policy integration algorithms," ACM Trans. Info. Sys. Sec., vol.11, no.1, pp.1–29, 2008.
[11] L. Peng and C. Zhong, "An access control model for web services in business process," IEEE/WIC/ACM Intl. Conf. on Web Intelligence, pp.292–298, 2004.
[12] F. Tartanoglu, V. Issarny, A.B. Romanovsky, and N. Lévy, "Coordinated forward error recovery for composite web services," 22nd IEEE Intl. Symp. on RDS, pp.167–176, 2003.
[13] S. Choi, H. Kim, H. Jang, J. Kim, S.M. Kim, J. Song, and Y.J. Lee, "A framework for ensuring consistency of web services transactions," Info. Softw. Tech., vol.50, pp.684–696, 2008.
[14] P.P.W. Chan, M.R. Lyu, and M. Malek, "Reliable web services: methodology, experiment and modeling," IEEE Intl. Conf. on WS, pp.679–686, 2007.
[15] L. Liu, Y. Meng, B. Zhou, and Q. Wu, "A fault-tolerant web services architecture," LNCS, vol.3842, pp.664–671, 2006.
[16] N.B. Lakhal, T. Kobayashi, and H. Yokota, "THROWS: An architecture for highly available distributed execution of web services compositions," 14th IEEE Intl. Work. on RIDE, pp.103–110, 2004.
[17] E. Alwagait and S. Ghandeharizadeh, "DeW: A dependable web services framework," 14th IEEE Intl. Work. on RIDE, pp.111–118, 2004.

[18] C.L. Fang, D. Liang, F. Lin, and C.C. Lin, "Fault tolerant web service," J. Syst. Archit., vol.53, no.1, pp.21–38, 2007.
[19] P. Greenfield, D. Kuo, S. Nepal, and A. Fekete, "Consistency for web services applications," 31st Intl. Conf. on VLDB, pp.1199–1203, 2005.
[20] A. Romanovsky, P. Periorellis, and A.F. Zorzo, "Structuring integrated web applications for fault tolerance," 6th Intl. Symp. on ADS, pp.99–106, 2003.
[21] A. Charfi and M. Mezini, "AO4BPEL: An aspect-oriented extension to bpel," WWW, vol.10, no.3, 2007.
[22] Z. Maamar, "On coordinating personalized composite web services," Inf. Softw. Tech., vol.48, pp.540–548, 2006.
[23] C.K. Fung, P.C.K. Hung, and D.H. Folger, "Achieving survivability in business process execution language for web services with exception-flows," 2nd IEEE Intl, Conf. on EEE, pp.68–74, March 2005.
[24] T. Mikalsen, S. Tai, and I. Rouvellou, "Transactional attitudes: Reliable composition of autonomous web services," IEEE Intl. Work. on DMS, 2002.
[25] L. Chen, B. Wassermann, W. Emmerich, and H. Foster, "Web service orchestration with BPEL," 28th Intl. Conf. on Softw. Eng., pp.1071–1072, 2006.
[26] L.J. Zhang, J. Zhang, and H. Cai, Services computing, Tsinghua University Press and Springer-Verlag, 2007.
[27] A. Avižienis, J.C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE Trans. Dependable Secure Comput., vol.1, no.1, pp.11–33, 2004.
[28] S. Brüning, S. Weißleder, and M. Malek, "A fault taxonomy for service-oriented architecture," 10th IEEE Intl. Symp. on HASE, pp.367–368, 2007.
[29] B. Kiepuszewski, R. Muhlberger, and M.E. Orlowska, "Flowback: Providing backward recovery for workflow management systems," ACM SIGMOD, pp.555–557, 1998.
[30] C. Liu, M. Orlowska, X. Lin, and X. Zhou, "Improving backward recovery in workflow systems," 7th Intl. Conf. on DSAA, pp.276–283, 2001.
[31] A. Mili, "Towards a theory of forward error recovery," IEEE Trans. Softw. Eng., vol.11, no.8, pp.735–748, 1985.

Yuna Kim received the B.S. and M.S. degrees in computer science and engineering from POSTECH, Korea, in 2001 and 2003, respectively. From 2003 to 2004, she was a researcher at POSTECH PIRL and worked for ebXML system development. She is currently working toward the Ph.D. at the same university. Her current research interests include services computing, dependable computing, and security.

Wan Yeon Lee received his B.S., M.S. and Ph.D. in Computer Science and Engineering from POSTECH in 1994, 1996 and 2000, respectively. He is currently an Associate Professor in the Department of Computer Engineering, Hallym University, Chunchon, South Korea. From 2000 to 2003, he was a research engineer in LG Electronics and worked for the standardization of Next GenerationMobile Network of 3GPP Group. From 2006 to 2007, he was a visiting professor in the School of Electrical Engineering, Purdue University, West Lafayette, USA. His research interest includes embedded system, real-time system, mobile computing, computer security, and parallel computing.

Kyong Hoon Kim received his B.S., M.S., and Ph.D. degrees in Computer Science and Engineering from POSTECH, Korea, in 1998, 2000, and 2005, respectively. Since 2007, he has been a full-time lecturer at the Department of Information Science, Gyeongsang National University, Jinju, Korea. From 2005 to 2007, he was a post-doctoral research fellow at GRIDS lab in the Department of Computer Science and Software Engineering, the University of Melbourne. His research interest includes real-time systems, Grid computing, and security.

Jong Kim received the B.S. in electronic engineering from Hanyang university, Korea, in 1981, the M.S. degree in computer science from the Korean Advanced Institute of Science and Technology, Korea, in 1983, and the Ph.D. degree in computer engineering from Pennsylvania State University in 1991. He is currently a professor in the Department of Computer Science and Engineering, Pohang University of Science and Technology, Pohang, Korea. From 1991 to 1992, he was a research fellow in the Real-Time Computing Laboratory of the Department of Electrical Engineering and Computer Science, University of Michigan. His major areas of interest are fault-tolerant computing, parallel and distributed computing, and security.