AdaFF: Adaptive Failure-handling Framework for Composite Web Services

Yuna KIM†, 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 combined with the composite Web service at runtime 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 adaptive 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 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 [3], [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 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 strategies against the same failure in accordance with the requirement of individual users. 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.

†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.
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, called Adaptive Failure-handling Framework (AdaFF), which adaptively 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. 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 at runtime.

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)ootnote{http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf}, and executes the generated workflow on the ActiveBPEL engine.

The remainder of this paper is organized as follows. Section 2 presents an example of a problem that motivates this work, and Sect. 3 briefly surveys related previous work. Section 4 describes a model of services and failures considered in 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 Section 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 failure-handling frameworks support only an identical recovery flow against the same failure. 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 ultimately continuous failure of hotel reservation 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.

Changing the failure recovery flow requires the redevelopment of the whole workflow of composite service. At every request with a user’s requirement, the development of the workflow containing the requirement-matched failure recovery flow delays the service delivery. Static generation of all possible recovery flows per a composite service enables users to select their favorite one immediately. However, it is difficult to determine an entire set of failure recovery flows when designing a composite service, because the user’s behavior and requirements are quite diverse and constantly changing. Moreover, it is inefficient in that most of them might not be used. Consequently, in this paper, we propose a new failure-handling framework, which dynamically generates the requirement-matched failure recovery flow and systematically combines the flow into the workflow at every request to accommodate various failure handling requirements of individual users.

3. Related Work

While adaptive service compositions have been sufficiently 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 mechanisms try to search for the best set of component Web services based on user’s context, and support the low-cost modification of the services at runtime. These mechanisms 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, failure recovery flow is implemented either together with normal operation flow or as a sub-module being combined independently of normal operation flow. In WS-BPEL, Web Services Composition Action Language (WSCAL) [12], and Fung et al. [23], failure recovery flow is inseparably intermingled with normal operation flow in a composite service. WS-BPEL is a process-oriented composition language used to implement a workflow of composite Web services in an XML-based grammar [3], [24]. WSCAL is an XML-based language that defines how participants belonging to a composite service are coordinated to deal with a failure. Fung et al. [23] proposed how to identify a failure critical for successful termination of a 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. In order to change failure recovery flow, the entire workflow needs to be redesigned.

There were attempts to implement failure recovery flows independently of normal operation flows, like in THROWS architecture [16] and WSTx framework [25]. In THROWS architecture, a composite Web service constructs a hierarchy of component services. In this architecture, failure recovery flow is determined by selecting a set of component services. In order to change the failure recovery flow, only the selection of embedded component services needs to be modified. WSTx middleware framework specifies which component services are involved in the composite service and what outcome is expected for success of the composite 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 support only one recovery strategy, compensation, so the modification of failure recovery flow means merely 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. System Model

4.1 Service Model

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.

4.2 Failure Model

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 unintentionally exposed and can be handled independently of the composite service.

4.3 Failure Recovery Model

There are two error recovery techniques: backward error recovery [29], [30] and forward error 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 errors and the forward recovery cannot guarantee the exact previous state.

Our framework adopts a combination of the two recoveries in order to support various user requirements. For backward recovery, our framework requires operation-coupled handlers of individual services. The handlers restore
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 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 through 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.

- **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 `<ignore>` in line 3.

- **Failure dependency** means that a failure of an activity is followed by having another activity roll back to the original state prior to the execution. If users demand to invalidate activity1’s completion upon activity2’s failure, they specify the names of the two activities in the element `<dependency>` in line 4: activity2 in `<source>` and activity1 in `<destination>`.

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 Section 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 from perspective of an activity. As shown in Fig. 5(b), failure-related activities (FRA) and in(out)flow 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(out)flow represents control/data dependency between activities including FRAs. In addition, activities of CBP are bound to component services through an in(out)link, which is referred to as service binding. Through the links, the composite service invokes the component services and receives the result from them. in_link conveys the result data of the service.
operation to the activity in CBP and \textit{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 \textit{in(out)\_hlink}, which is referred to as handler binding. \textit{in(out)\_hlink} is similar to \textit{in(out)\_link} but connects FRA to failure handler. 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: \textit{Element Services} and \textit{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 (Sect. 4.2) that restore to the former state of the service. Unlike the element services, the control services provide control handlers (Sect. 4.2) that perform failure handling regardless of the service state.

Element services and failure handlers are bound to normal activities and failure-related activities in an AFH-
EBP, respectively. 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, and the expressions will be used for describing an algorithm that generates AFH-EBP (Algorithm 1).

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 exploits ActiveBPEL engine 2.0† as an open-source process execution engine 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 the Client Interface, users explore abstract information about CBPs stored in the CBP Archive, select one of them complying with their demands, and request it with FRRs associated with the CBP. CBPs are designed by a business

†http://www.activebpel.org
process architect in advance. Users can specify their requirements by simply checking some checkboxes through the
Client Interface, which produces FRRs in XML format.

FRR Interpreter processes XPath\(^1\) 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). 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 \(<\text{bestTry}>\) 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 \(<\text{ignore}>\) 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 \( A_i \).
- Source activity \( A_i \) in \(<\text{dependency}>\) 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 destination activity \( A_j \) is bound. The handler
  carries out the restoration or the abortion of the destination activity \( A_j \).

We do not present in this algorithm how to find 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].

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. Finally the
Preprocessor obtains an AFH-EBP instance in the format of WS-BPEL. Failure-related activities and failure-related
flows are expressed by \(<\text{compensationHandler}>\) and \(<\text{faultHandler}>\) 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
by adding Handler Decision Module to an ActiveBPEL engine. Handler Decision Module chooses the right type
of operation-coupled handler at runtime. The operation-coupled handler has two types: restoration handler and
abortion handler. Restoration handler restores the state of the service as before its execution, and the abortion

\(^1\)http://www.w3.org/TR/xpath20/
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, max_retry: the number of times to retry activities */
1 /* index of failure-related activities */

k ← 0
for ∀ A_i ∈ A do /* A_i.name: the name of an activity A */
if there exists bestTry[activity=A_i.name] then
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 ≤ max_retry) to FRF.
Add (FRA_{k+1}, H(rebind, A_i)) to FBind.
Add (FRA_{k+1}, FRA_{k+2}, A_i.input_parameters, binding success) to FRF.
Add (FRA_{k+1}, H(retry, A_i)) to FBind.
k ← 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 FBind.
endif
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

endfor

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 number of times that services can be retried. Such policies can be configured before running the Process Orchestrator.

7. Evaluation

We evaluate our prototype with a travel booking scenario. The scenario shows how two different AFH-EBP instances are generated from a single CBP with reference to two different FRRs.

There is a CBP that 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). We consider two users; User A offers an FRR as shown in 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 as shown in Fig. 7(c), which has the following semantics: do not care about the failure of hotel reservation.

According to Algorithm 1, two different AFH-EBP instances are generated for user A and user B, as shown in Fig. 8(a) and Fig. 8(b), respectively. In both instances, Reserve Hotel and Reserve Flight activities are bound to Hotel Service and Flight Service, respectively.

1) AFH-EBP instance for user A: Due to <bestTry> declared in the FRR of user A, if an application-dependent failure named seat unavailable or an application-independent failure named faultcode† occurs in the

†faultcode is designated by the composite service provider.
Receive user's request

Reserve Flight

Invoke Payment Service

Reply to User

Fig. 7 Examples of one CBP and two FRRs.

Reserve Flight activity 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 Reserve Flight activity to the service by using rebinding handler. Subsequently a Retry activity executes the Reserve Flight activity 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 <ignore> declared in the FRR of user B, if an application-dependent failure named room unavailable or an application-independent failure named faultcode occurs in the Reserve Hotel activity, an Ignore activity forces the Payment activity to ignore the input from the Reserve Hotel activity by using ignoring handler.

The prototype successfully generates AEH-EBP instances for the two users with different FRRs at runtime, and the instances can handle failures according to the requirements.
8. Concluding Remarks

In this paper we proposed a new 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 failure to a level that is acceptable to users. As much handling as users can endure failures. The framework also contributes to a flexible service design by which an architect of composite services would never care about failure handling or variable 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 at runtime.

References