Integrated Constraint Violation Handling for Dynamic Service Composition

MingXue Wang, Kosala Yapa Bandara and Claus Pahl
Dublin City University, Ireland
[mwang|kyapa|cpahl]@computing.dcu.ie

Abstract

Dynamic service composition is suitable for on-demand business requests. For autonomic computing, service composition needs to deal with runtime environment faults, but also with business constraint violations which result from business requirements. We propose an approach for integrated handling of business constraint violations and runtime environment faults for dynamic service composition. We introduce a loosely coupled implementation architecture to maintain the platform-independent nature.

1 Introduction

Service Oriented Architecture (SOA) is an important solution for Enterprise Application Integration. Business components are exposed as platform-independent Web services, which are then orchestrated using the Business Process Execution Language (WS-BPEL or BPEL for short). Dynamic service composition is a promising direction for *on-demand* business requests [7, 2].

Faults can be caused by various reasons during the execution of a software system. However, there is no standard approach that addresses fault tolerance for the Web services environment [6, 9]. BPEL offers a fault handling mechanism which can catch runtime faults. As a description and execution language, BPEL does not provide any remedial strategies for faults unless predefined by developers at design time. Some efforts have been made regarding reliable service composition [9, 13, 10, 6, 15, 5]. However, firstly, most do not address the problem of dynamic service composition. Concrete fault information can not be expected at design time, i.e., remedial strategies are required to be dynamically selected based on runtime fault analysis. Secondly, current work does not take into account that, in business environments, business process developers frequently add and change business rules on top process data flows to address the requirements of successful semantic exchanges [8, 14, 2]. Therefore, business constraint violations handling is a must besides runtime environment fault handling.

BPEL is a de-facto standard for service composition. Considering the platform-independent nature of SOA [9], we need an engine-indepdent integrated handling of business constraint violations and runtime environment faults.

We introduce an approach for the integrated handling of business constraint violations and runtime environment faults for dynamic service composition with BPEL with:

1. an intelligent mechanism for dynamic remedial strategy selection. A fault taxonomy mapped to remedial strategies is basis of the remedial analysis. The comprehensive taxonomy captures both business constraint violations and technical runtime environment faults.
2. an instrumentation template for violation and fault handling allows dynamically selected remedial strategies to be applied within process execution. A BPEL solution of this instrumentation offers fault handling implementation loosely coupled to a BPEL engine.

Our prototype evaluation shows the instrumented BPEL processes have no additional overhead compared to the original business processes on default execution.

This paper is structured as follows. Section 2 introduces fault monitoring with BPEL fault handlers. Section 3 illustrates our remedial knowledge base, which has been developed to focus on autonomic Web service composition. In Section 4, we introduce our architecture and core components. Section 5 details the process instrumentation with the violations and faults handling template. After an evaluation, we discuss related work and present some conclusions.

2 Constraint violation and fault monitoring

Fault monitoring is responsible for the fault detection and fault data collection to provide input for a subsequent fault analysis. For services, a fault is an abnormal condition which may lead to a service process failure [4].

BPEL provides the capability to catch and manage faults using fault handlers (<*catch*> and <*catchAll*>). Fault handlers can be attached to the entire process or a smaller execution scope (<*scope*>). If the process or scope completes normally, fault handlers are ignored, but if a fault situation occurs, the fault is propagated to the fault handler. The fault handler takes over the process execution by executing additional activities inside the fault handler.

Using BPEL's fault handlers for monitoring can avoid overheads on additional monitoring processes and BPEL engine-dependent monitoring components. We weave context model-based constraint services (as pre- and post-condition) into BPEL processes for each business services [2]. The context model captures both business and technical level information for semantic service composition. Constraint services validate context information for each service-invoking instance. Constraint violations as faults are thrown by the constraint services. This way, we simply transform business constraint violations into service faults, which can be monitored by default BPEL fault handlers.

3 Fault analysis

The fault analysis takes fault data as input and determines a suitable remedial strategy for the fault instance. The basis of the fault analysis is predefined fault remedial knowledge. It defines various remedial strategies for different fault scenarios. There are three steps for defining the fault remedial knowledge: defining a fault taxonomy, defining remedial strategies, and matching each fault category with remedial strategies.

3.1 A taxonomy of fault categories

In order to suggest and apply a suitable remedial strategy, a key factor are the types of fault to look for. Thus, we categorize possible faults that can occur for dynamically composed services [4, 1, 15]. In order to deal with business constraint violations, a fault taxonomy needs to capture both business and technical faults. Our fault taxonomy is derived from a context model used for our constraint validation services [2]. Root fault categories are: Functional, Quality of Service, Domain, and Platform Context.

Functional context fault: describes the violation of operational features of Web services. It is grouped into Syntax, Effect and Protocol faults.

1. **Syntax fault:** includes violation of input/output parameters that define the operations' messages and the data types for the parameters for invoking the service.
2. **Effect fault:** includes faults in terms of pre-conditions and post-conditions, i.e. functional failure during an operation execution.
3. **Protocol fault:** refers to faults related to the consistent exchange of message between services involved in a service composition to achieve their goals.

Quality of Service context fault (QoS): violation of end-to-end quality in service compositions, includes single and compound services. It is grouped into four categories.

1. **QoS runtime fault:** violation of properties related to the execution of a service. This includes Performance, Reliability and Availability violations.
2. **Financial/business fault:** violation relates to the financial context which allows the assessment of a service from a financial or business perspective. This includes Cost, Reputation and Regulatory violations.
3. **Security fault:** violation of security requirements. This includes Integrity, Authentication, Non-repudiation and Confidentiality violations.
4. **Trust fault:** violation refers to failed establishment of trust relationships between client and providers.

Domain context fault: refers to application domains that need specific requirements to be met for services.

1. **Semantic fault:** violations related to the semantic framework (i.e. concepts and their properties) in terms of vocabularies, taxonomies or ontologies.
2. **Linguistic fault:** violation related to the language used to express queries, functionality and responses.
3. **Measures and standard fault:** violation relates to locally used standards for measurements, currencies, etc.

Platform Context fault: violation relates to the technical environment a service is executed in (includes classical technical platform faults).

1. **Device fault:** refers violation with the computer/hardware platform on which the service is provided.
2. **Connectivity fault:** refers violation with the network infrastructure used by the service to communicate.

3.2 Defining remedial strategies

Potential remedial strategies - such as retry or replace - have been introduced in [15, 9, 6, 1, 13]. In dynamic service composition, remedies are selected and applied dynamically. These strategies need address process impact, resource consuming, and additional component requirements to support remedial knowledge definition. Naturally, some strategies are context-dependent. Therefore, we only discuss a framework here.

We categorise remedial strategies into goal-preserving and non-goal preserving strategies. Goal preserving strategies aim to recovery from faults immediately; the business goal would be completed with a continued process execution after fault recovery. In contrast, non-goal preserving strategies do not attempy recovery. They provide additional actions to assist possible future recovery.

3.2.1 Goal-preserving strategies

In BPEL, an invoke activity calls a business activity performed by a Web service. This can be monitored with a fault handler using the scope attachment. We can identify four goal-preserving strategies for fault handling as follows:

Ignore: does not take any action on a fault. It ignores specified faults that do not affect the overall business goal.

Retry: retries the fault causing service. Maximum retry times and interval before each retry can be defined.

Replace: replaces faulty service by alternative service with same capabilities.

Recompose: discards the faulty process and establishes an alternative process with the same goal.

Ignore and Retry are lower-level recoveries, which keep the original process workflow. Applying them requires less time resources. In higher-level recovery (Replace, Recompose), an additional component is needed for discovering alternatives, requiring more resources. In general, lower-level goal-preserving strategies should be applied first, as they have less impact. The following example allows one Retry opportunity before applying Ignore:

```
<preConditionViolationRemedy>
  <sequence>
     <retry><max>1</max><waitingTime>P0Y0M0DT0H0M1.0S</waitingTime></retry>
     <ignore><value>true </value><log>level_1 </log></ignore>
  </sequence>
</preConditionViolationRemedy>
```

There are two ways to provide alternative replacements. Firstly, alternative services are pre-assigned to remedial strategies. The Replace strategy can be applied instantly. Secondly, alternative services are dynamically discovered based on functional and non-functional properties. Recompose is different, as in dynamic composition, we presume service processes are only discovered at runtime. However, depending on the business goal and size of service registry, Recompose can be time consuming. Hence, we have developed a selective process repository to minimize time [12]. The process repository saves composed services and processes with a categorized fault ratio. Alternative processes can be retrieved and selected from the process repository.

Replace is a passive replication technique; the backup service is only called after a primary service fault. A parallel strategy in introduced in [6]. Several alternative services are invoked in parallel for one invocation. The first response received is chosen for ongoing process execution. A disadvantage is that all alternative services need to be discovered dynamically at composition time. It also causes large overheads on computation and network resources to execute alternative services. Moreover, it could cause business goal violations on state update, e.g. a bill is paid twice. The advantage is that only the best performing service is picked, and does not need to be replaced. We obtain a similar result and avoid its disadvantages by selecting alternative services for replacement. With multiple alternative services, the services' fault ratio and, if performance is the issue, response times in the process repository is used for alternative services sequencing. The service response time is calculated from the time window between end of pre-condition and start of post-condition constraint services.

Replace and Recompose might call for compensation or rollback. Compensation would be a pre-condition of these remedial strategies in many cases. Deploying an alternative process, the system needs to clear up partially executed faulty processes (rollback), i.e. the process execution needs transactional behaviour. This is difficult as no common protocol exists for Web services [11, 3, 5]. BPEL compensationHandler enable to define an activity at the scope or process level whose execution reverses previously executed application logic. However, there is no automatic restoration of data during compensation. The application might define its own compensation behaviour. We assume for state-updating services that there is at least one service that can rollback its effect and does not depend on any state for execution. For Replace, compensation may also be required for post-condition faults before an alternative service is retried.

3.2.2 Non-goal preserving strategies

Non-goal preserving strategies do not impact on process execution and can be combined with other strategies includes goal-preserving strategies. We define three non-goal preserving strategies. **Log** records the captured fault. It could be applied at different levels, e.g. Level-1 logs fault source and fault message. Level-2 logs data transmission of fault sources as well. This data is saved in a fault log database. **Alert** ensures that relevant stakeholders will be notified. **Suspend** suspends the faulty service or process until future investigation, if the fault element exceeds an acceptable fault ratio. A composition component would normally avoid suspended processes. The purpose is to isolate the fault elements to avoid possible repeat faults.

3.3 Fault categories & remedial strategies

Matching fault categories with remedial strategies needs to consider different levels of data. From low to high, there are default remedial data, services and process-specific remedial data and application-specific remedial data.

Default remedial data comes from an analysis of fault categories. It is the proposed solution for all fault categories (Table 1). Retry is suitable for most remote faults where post-condition constraints are violated. For instance, a missingOutput faults might result from a temporary unavailable service. Retry is not suitable for pre-condition constraint violations. Replace and Recompose are suitable for all fault categories. Recompose would be last option as it is the most time and resource consuming. The following is a pre-condition remedial strategy for securityFaults:

```
<securityFault>
   <preConditionViolationRemedy>  <sequence>
          <ignore><value>false </value><ignore>
          <retry><max>0</max><waitingTime>P0Y0M0DT0H0M0.0S</waitingTime></retry>
          <replace><value>any</value></replace>
          <recompose><value>true </value><log>level_1 </log></recompose>
    </sequence>  </preConditionViolationRemedy>
   <postConditionViolationRemedy >...</postConditionViolationRemedy>
</securityFault>
```

	Pre-cond constraint violation	Post-cond constraint violation
Ignore	All fault categories	All fault categories
Retry	Not suitable	Functional context fault; Platform context fault
Replace	All fault categories	All fault categories
Recompose	All fault categories	All fault categories

Table 1. Default remedial data

Service-specific remedial data is defined according to service descriptions for specific services only. Services with side-effects need compensation. Services can have fault and compensation handlers as child elements.

```
<service>
   <serviceReference>
      <endpointUrl>http://localhost:8080/.../BankpaymentService</endpointUrl>
      <operation>Bankpayment</operation>
    </serviceReference>
   <faults>
      <securityFault >...</securityFault>
       ...
    </faults>
   <compensation>
      <serviceReference>
         <endpointUrl>http://localhost:8080/.../BankrefundService</endpointUrl>
         <operation>Bankrefund</operation>
       </serviceReference>
    </compensation>
</service>
```

Process-specific remedial data is defined according to business goals and application domains. It needs to comply with application requirements and organisational policies. In processes involving financially sensitive data, security-level mismatch faults are not acceptable; some processes would mark minor security faults as ignorable. Organisations might define their own trusted alternative service as a Replace remedy. Processes could have high-level faults and services as child elements.

```
<process>
   <processReference>
      <onDemandRequest>Gas—BillPayment </onDemandRequest>
    </processReference>
   <services>
      <service>
         <serviceReference >...</serviceReference>
         <faults >...</faults>
       </service>
        ...
     </services>
    <faults >...</faults>
</process>
```

For a fault instance, the system searches for remedial strategies from high to low level. Higher levels are customizations of lower level data. For example, remedial strategies of securityFault for a business service is appeared in both service- and process-specific level, only the remedies of process-specific level is valided.

4 Architecture and core components

We divide our fault tolerant architecture into three layers (Fig. 1): process execution layer, composition and fault-tolerance layer and database layer. A BPEL engine is responsible for the process execution layer. The three databases of the database layer have been discussed in the pervious section. The four components in the composition and fault-tolerance layer are core of the archicture. They interact with the instrumented BPEL process, thus our approach is completely BPEL engine independent. We discuss the core components here.

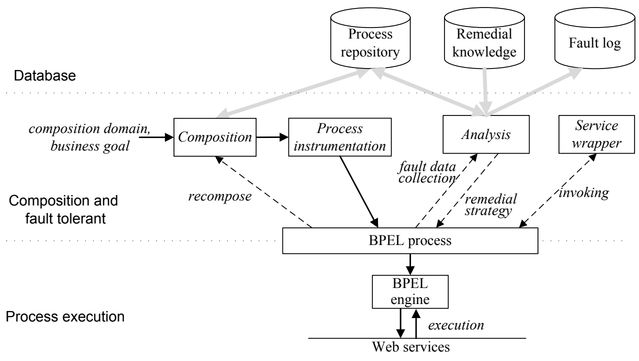

Figure 1. Three layered system architecture

The **composition component** composes services to service processes based on business goals. This requires a services ontology to provide the composition domain. Service processes are saved in an indexed process repository for possible future reuse, such as the Recompose remedy. The process repository also supplies a suspended-list of services and processes for composition to filter invalid processes. To enable recomposition, the composition component is exposed as a Web service *recompose()* with a *processReference* as input. *ProcessReference* contains the process name as business goal and the process index which differentiates multiple processes for the same business goal.

The **process instrumentation component** converts business processes to BPEL processes, which includes the generation of deployment descriptors. An important step of the conversion is instrumenting the fault monitor and handling mechanism within the BPEL process.

The **analysis component** utilizes the remedial knowledge we defined earlier to provide remedial strategies for any fault instance. It also updates the fault ratio of services and processes in the process repository and updates the fault log if the Log strategy is required. All non-goal preserving remedial strategies could be implemented by the analysis component. Its Web service interface *analyse()* has 5 inputs (Fig. 2). *faultData* is a fault variable or constraint violation collected by the BPEL fault handlers. *processReference* denotes the current BPEL process. *invokingServiceReference* is an instance of *ServiceReference* that identifies a Web service. A *ServiceReference* contains a service endpointUrl and an invoking operation. *RequestData* and *responseData* record fault service data transmission for the Log strategy. *analyse()*'s output *analyseResult* is a complexType, which we detail in Section 5.

The **service wrapper component** is a dynamic service invoker. It wraps actual business services into a unified service interface *genericOperation()* (Fig. 3). The *genericOperation()* has two input parts. *requestData* is input of the

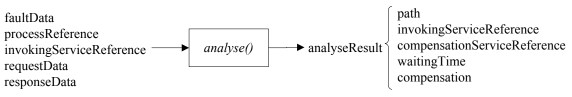

Figure 2. *analyse()* **service data flow**

business service; *invokingServcieReference* is the identity of the business service. *responseData* returned by *genericOperation()* is output of the business service. The purpose of this wrapper is to provide a dynamic binding partner link. In BPEL, partner links define how the process interaction with other business processes and services. Dynamic binding partner links allow that business services endpoints are selected and assigned to the partner link through configuration or runtime input. The limitation is these business services must have the same interface. Our wrapper component achieves dynamic binding without this limitation.

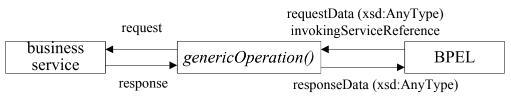

Figure 3. *genericOperation()* **service data flow**

5 Instrumentation for violation handling

The purpose of process instrumentation is the fault monitoring and handling capability of BPEL processes, i.e. dynamically selected remedies are able to act on process executions when faults occur. The instrumentation is based on an intrumentation template for violations and faults for each service-invoking activity (Fig. 4). Since we use constraint services for pre- and post-activity validation, two constraint services are bound to each invoking service.

We describe the violation and fault handling template in two parts. The first part is the <*repeatUntil*> container in the top half of Fig. 4. It achieves fault monitoring and supports the Ignore, Retry, and Replace remedial strategies. The second part is for the Recompose strategy; see bottom half of Fig. 4. Since non-goal preserving strategies do not impact the processes, they will not be considered by the violation handling template.

5.1 Ignore, Retry, and Replace

Five variables for each invoking activity define the violation handling context and determine the handler execution: *invokingServiceReference* provides the current invoking activity service reference, e.g. *billPay()*. *composition* denotes if compensation of the current invocation is needed for recomposition. It has the default value true. *compensationServiceReference* names the compensation service of the current invocation - the initial value is empty. *waitingTime* defines the waiting duration for the Retry strategy

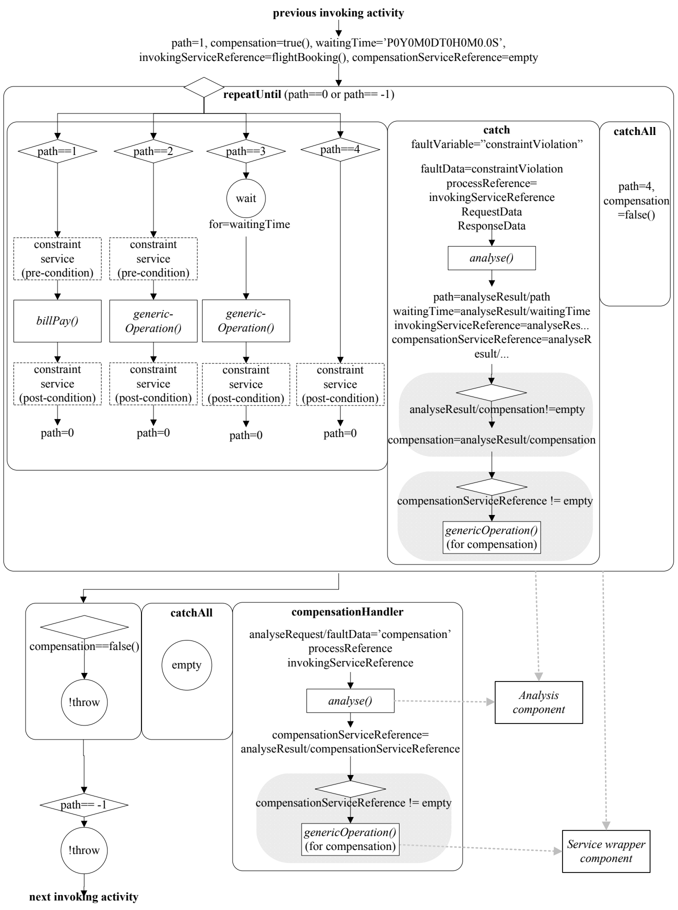
previous invoking activity
path=1, compensation=true(), waitingTime='P0Y0M0DT0H0M0.0S',
invokingServiceReference=flightBooking(), compensationServiceReference=empty
repeatUntil (path==0 or path== -1)
path==1
path==2
path==3
path==4
wait
for=waitingTime
constraint service (pre-condition)
billPay()
generic-Operation()
constraint service (post-condition)
path=0
catch
faultVariable="constraintViolation"
faultData=constraintViolation
processReference=
invokingServiceReference
RequestData
ResponseData
analyse()
path=analyseResult/path
waitingTime=analyseResult/waitingTime
invokingServiceReference=analyseRes...
compensationServiceReference=analyseResult/...
analyseResult/compensation!=empty
compensation=analyseResult/compensation
compensationServiceReference != empty
genericOperation()
(for compensation)
catchAll
path=4,
compensation
=false()
compensation==false()
!throw
catchAll
empty
compensationHandler
analyseRequest/faultData='compensation'
processReference
invokingServiceReference
analyse()
compensationServiceReference=
analyseResult/compensationServiceReference
compensationServiceReference != empty
genericOperation()
(for compensation)
Analysis component
Service wrapper component
path== -1
!throw
next invoking activity

Figure 4. Violation handling instrumentation template

with an initial value 0. The execution *path* is by default initialised as the uninstrumented original execution path: pre-condition constraint service, invoking service, and post-condition constraint service.

In addition to context constraints, paths are the second key concept in the template. In a fault-free scenario, only the default path is executed. Otherwise, the fault is caught by attached fault handlers and the *analyse()* service inside the handlers determines the subsequent actions, including selection of a new execution path. In the template, the <*repeatUntil*> container is important. It only ends when a path is executed successfully (*path=0*) or *analyse()* decides to recompose the current process (*path=-1*). We distinguish pre-and post-condition-based constraint violations.

For faults caused by **pre-condition constraint violation**, the fault handler passes the fault variable thrown by constraint service (*constaintViolation*) to the *analyse()* service. *analyse()* takes *processReference*, *invokingServiceReference* and other additional variables.

1. If the remedial knowledge suggests to Ignore the fault,

analyse() returns *path=3, compensation=true(), compensationServiceReference=empty, waitingTime=0*, and keeps *invokingServiceReference=billPay()*. The <*repeatUntil*> forces the process to execute path 3. The *billPay()* is executed by the wrapper *genericOperation()* and the post-condition is validated.

2. If the Replace strategy is applied, the *analyse()* sets *path=2*, *compensation=true(), compensationServiceReference=empty, waitingTime=0* and assigns the *invokingServiceReference* to an alternative service. The second path is similar to first path, except *genericOperation()* replaces the original business service *billPay()*. This allows the alternative service to be executed through the wrapper component.
3. If *analyse()* suggests the recomposition strategy, it keeps *invokingServiceReference*, sets *compensation=true*, *compensationServiceReference=empty*, *waitingTime=0*, and *path=-1* to end <*repeatUntil*>. We discuss this in the next subsection.

For faults coming from the **invoking service activity**, i.e. either *billPay()* or *genericOperation()*, a non-constraint violation fault is caught by the <*catchAll*> fault handler. The fault handler assigns *path=4*, which means the post-condition constraint service deals with the fault and throws a constraint violation fault for *analyse()*, i.e. a syntax constraint violation is expected thrown from the constraint service for *faultData*. Variable *compensation* is also set to *false*, as no compensation is required for this invocation activity during recomposition.

For faults caused by **post-condition constraint violation** we distinguish the four strategy cases:

1. In the case of Ignore, *analyse()* keeps *invokingServiceReference*, sets *compensation=empty*, *compensationServiceReference=empty*, *waitingTime=0*, and *path=0* to end <*repeatUntil*>. The compensation variable keep its previous value, i.e. it remembers if a fault came from the invoking service activity.
2. For the Retry strategy, *analyse()* sets *path=3*, *compensation=true*, a *waitingTime* for the <*wait*> activity to give a interval before retry execution, *compensationServiceReference=empty* and keeps the previous *invokingServiceReference*.
3. For Replace, *analyse()* sets *path=2*, *compensation=true*, *waitingTime=0* and an alternative service for *invokingServiceReference*. In addition, Replace for post-condition faults also needs to check if compensation is required. If *analysis()* returns a *compensationServiceReference*, *genericOperation()* within the fault handler executes the compensation service.
4. For Recompose, *path=-1*, *compensation=empty*, *compensationServiceReference=empty*, *waitingTime=0* is set and *invokingServiceReference* and *compensation* are kept.

In case of faults with alternative replacement services, the same strategy as above is applied again.

5.2 Recompose

We continue with the second *scope* (Fig. 4 with <*compensationHandler*> attachment). The second scope is responsible for compensation of the whole process scope, i.e. for Recompose. If variable *compensation==false*, a throw activity throws a defined fault. An attached <*catchAll*> handler catches the fault and does nothing. The purpose is to mark this scope as a faulty scope. The BPEL compensationHandler attachment can only be triggered by a succcessful scope for process scope compensation. In that case, such as Ignore with a post-condition fault, a compensation handler attached to the scope within <*repeatUnit*> will not be triggered as a fault occurred. Thus, we create this scope for invocation activity compensation. A *compensation* variable decides whether to trigger it.

If *analyse()* decides to Recompose (*path=-1*), a <*throw*> activity throws a fault. The process scope catches this fault and starts scope composition before calling *recompose()* (Fig. 5). All fault-free compensation scopes are executed in backward order. If any invocation activity requires compensation, *analyse()* provides a rollback service, which is executed through *genericOperation()*.

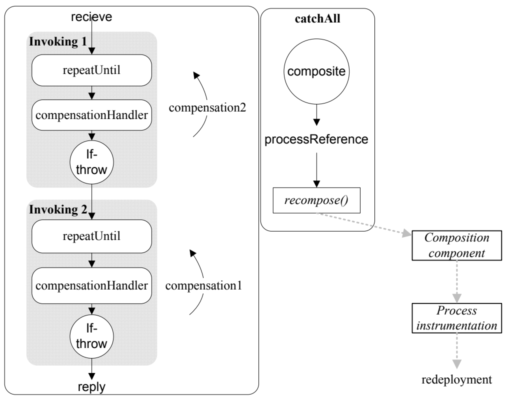

Figure 5. The structure of a process scope

5.3 Template for normal processes

In case of normal service processes which without pre- and post-condition constraint validation services, our violation handling template is also applicable for technical platform faults with a small modification (Fig. 6). The difference is inside the <*repeatUntil*> container. We need fault handlers for each BPEL fault to be caught. *analyse()* only sets *compensation=false* for the Ignore strategy. A suitable remedial knowledge database needs to be provided. A minor limitation of our template is it does not deal with faults during compensation service execution due to the complexity of service transaction handling. However, backups for compensation are usually not provided by the designer.

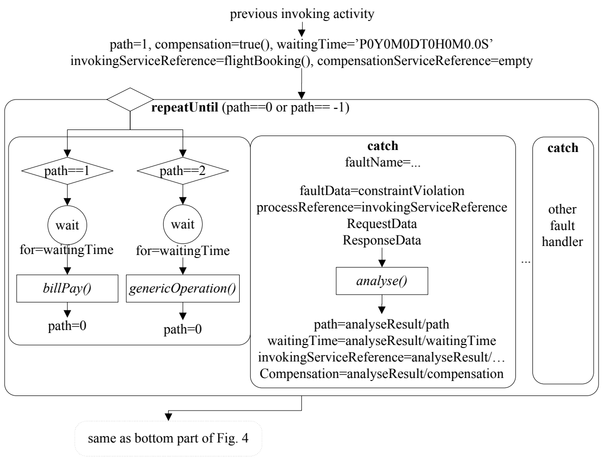
previous invoking activity
path=1, compensation=true(), waitingTime='P0Y0M0DT0H0M0.0S'
invokingServiceReference=flightBooking(), compensationServiceReference=empty
repeatUntil (path==0 or path== -1)
path==1
path==2
wait
for=waitingTime
wait
for=waitingTime
billPay()
genericOperation()
path=0
path=0
catch
faultName=...
faultData=constraintViolation
processReference=invokingServiceReference
RequestData
ResponseData
analyse()
path=analyseResult/path
waitingTime=analyseResult/waitingTime
invokingServiceReference=analyseResult/…
Compensation=analyseResult/compensation
...
catch
other fault handler
same as bottom part of Fig. 4

Figure 6. Template for normal processes

6 Evaluation

Objectives and Strategies. We focus on effectiveness and performance as the central criteria. Effectiveness and correctness are essential. We need to ensure correct remedial strategies are selected for different fault categories based on the remedial knowledge. Selected remedial strategies must be executed correctly in order for business goals to be achieved. We used a test driven approach to evaluate our system within a utility bill payment application.

Violation handling performance differs for each process instance. It depends on the number of faults during process execution, the size of remedial knowledge, the remedial strategies applied, and the waiting time of retrials. Constraint validation services also cause overheads. Our process instrumentation makes business processes more complex. A concern is if the violation and fault handling template causes unacceptable overheads compared to the original process execution.

Application and Test Cases. In total 35 test cases were designed for the bill payment process to test the correctness of the remedial strategy implementation, which involves four business services (requestBill(); billPay(); updateRecords(); infoProvider()). The last three services include state update actions, i.e. can be used to test compensation. We also developed three alternative services for each service to test replacement remedies for faults with alternative services. The following table shows one of the test cases.

In the test case, we created a pre-condition measurement constraint violation for the billPay service, which can be replaced by billPayAlt1. We also disabled the billPayAlt1 to

Input	`<ns2:CompanyBillPayment xmlns:ns2="http://businessService/">` `<customer>...</customer></ns2: CompanyBillPayment>`
Constraint violation and fault instances	Pre-condition <*measureFault*> on *billPay* service; Turn off *billPayAlt1* service to create a runtime fault; Remedial strategy: Replace with any
Expected service invoking sequence	*requestBill; billPay; analyse; genericOperation(billPayAlt1); analyse; genericOperation(billPayAlt2); updateRecords; infoProvider*
Expected output	`<ns2: CompanyBillPaymentResponse xmlns:ns2="http://busi...">` `<return>...</return></ns2: CompanyBillPaymentResponse>`

simulate a runtime fault, which can be recovered by the billPayAlt2 replacement. Each test case is designed for a different scenario. In another, similar test case, we changed the measurement violation as post-condition of a billPay fault; then we expect the billPay compensation service to be executed as well (i.e. the correct amount is debited from a customer account). Other complex test cases cover e.g. violations of last step, which need to rollback previous actions before Recompose. These cases cover all remedial strategies with various designed constraint violation and runtime fault scenarios.

We created nine pairs of processes to evaluate performance. Each process contains between 2 and 10 business services. We instrumented each process, which creates nine pairs of processes to compare the performance. Each process was executed six times using the ActiveBPEL engine v5.0.02 and we took the average execution time.

Results. Based on an analysis of process outputs, the fault log database, and the BPEL engine execution log, we verified all test cases are effective, i.e have correctly implemented the remedial strategies.

For performance, following table shows execution time (ms) of each pair of processes in our experiment.

No. services	2	3	4	5	6	7	8	9	10
Original	1340	1992	2563	3256	3851	4488	5199	5779	6378
Instrumented	1352	2052	2675	3293	3876	4522	5319	5820	6560

Discussion. With all test cases successful, we can demonstrate our approach provides a reliable violation and fault handling for dynamic service composition. It, of course, depends on alternative services and processes being supplied for the replacement remedies.

The performance evaluation results show that the instrumented processes does not introduce any significant overhead (in average less than 1% over all cases). The instrumented processes do not delay business processes execution time unless a violation or fault needs to be handled.

7 Related work

Work on reliability for Web service composition can be classified into two categories. The first category does not address the problem of dynamic service composition. Software developers are required to embed concrete remedial strategies into service processes before deployment. [3] requires developers to define monitoring and/or recovery actions for each business process in a Web Service constraint language. A supervision framework was designed for individual BPEL engines to supervise the constraint rules. [6] and [10] defined some BPEL fault-tolerant patterns, which allow single recovery strategies to be translated into BPEL processes according predefined recovery actions. However, they cannot handle recovery strategy selection at runtime.

The second category provides dynamic fault handling. Sometimes, this is achieved through engine-dependent implementations. The execution engines is modified to retain the original simple business process workflows. [13] develops the VieDAME Monitor component which extends a BPEL engine with different engine adaptors. The monitor component uses a *try-catch* structure to handle service faults. [15] achieves runtime recovery through an extended ActiveBPEL engine. In contrast, some approaches try to maintain engine independence. [5] uses a job submission description language (JSDL) to describe the business processes as job flow. Each job unit has to be translated to a BPEL flow for execution. A job flow manages jobs execution and can resubmit a job for Retry and Replace strategies. This introduces significant complexity through the job flow manager. It is also restricted to non-transactional behaviour and, thus, does not support any compensation.

In addition, as discussed, the above work essentially focusses on runtime environment faults. Business constraint violations are not taken into account, as we do.

8 Conclusions

In this paper, we have introduced an approach for business constraint violation and runtime environment fault handling for dynamic service composition. We have developed an intelligent remedial knowledge base for dynamic remedial strategy selection. We provided an instrumentation template for constraint violation and runtime fault handling to enable an engine-independent implementation. We demonstrated the effectiveness and the limited overhead through instrumented process execution.

Our approach is lightweight in that we do not require engine extension and do not cause overheads during normal execution. The remedial strategy selection for both business and technical constraint violation monitoring and analysis is integration into a normal fault handling mechanism.

As software quality aspects such as performance are of central importance for dynamic service composition, we intended to run more complex test scenarios to measure our prototype violation and fault handling performance in the future. We also plan to study and apply advanced learning based mining technique to optimize service and process selection to influence handling performance.

Acknowledgment

The authors would like to thank the Science Foundation Ireland for their support for the CASCAR project.

References

[1] D. Ardagna, C. Cappiello, M. Fugini, E. Mussi, B. Pernici, and P. Plebani. Faults and recovery actions for self-healing web services. In *15th Int. World Wide Web Conf.*, 2006.

[2] K. Y. Bandara, M. Wang, and C. Pahl. Dynamic integration of context mdel cnstraints in web service processes. In *IASTED Intl. Conf. on Software Engineering*, 2009.

[3] L. Baresi and S. Guinea. A dynamic and reactive approach to the supervision of bpel processes. In *1st India Software Engineering Conference*, 2008.

[4] K. M. Chan, J. Bishop, J. Steyn, L. Baresi, and S. Guinea. A fault taxonomy for web service composition. In *3rd Intl. Workshop on Engineering Service Oriented Applications (WESOA)*, 2007.

[5] G. Dasgupta, O. Ezenwoye, L. Fong, S. Kalayci, S. M. Sadjadi, and B. Viswanathan. Design of a fault-tolerant job-flow manager for grid environments using standard technologies, job-flow patterns, and a transparent proxy. In *Intl. Conf. on Software Engineering and Knowledge Engineering*, 2008.

[6] G. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2006.

[7] K. Fujii and T. Suda. Semantics-based dynamic service composition. *IEEE Journal on Selected Areas in Communications*, 23(12), 2005.

[8] M. E. Kharbili and T. Keil. Bringing agility to business process management: Rules deployment in an soa. In *6th IEEE Euro. Conf. on Web Services, Business Track*, 2008.

[9] J. Lau, L. C. Lung, J. d. S.Fraga, and G. S. Veronese. Designing fault tolerant web services using bpel. In *7th IEEE/ACIS Intl. Conf. on Computer and Information Science*, 2008.

[10] A. Liu, Q. Li, L. Huang, and M. Xiao. A declarative approach to enhancing the reliability of bpel processes. In *2007 IEEE Intl. Conf. on Web Services*, 2007.

[11] T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous web services. In *Workshop on Dependable Middleware-based Systems*, 2002.

[12] C. Moore, M. Wang, and C. Pahl. An architecture for autonomic web servive process planning. In *3rd Workshop on Emerging Web Services Technology*, 2008.

[13] O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *17th Intl. World Wide Web Conference*, 2008.

[14] F. Rosenberg and S. Dustdar. Business rules integration in bpel - a service-oriented approach. In *7th IEEE Intl. Conf. on E-Commerce Technology*, 2005.

[15] S. Subramanian, P. Thiran, N. C. Narendra, G. K. Mostefaoui, and Z. Maamar. On the enhancement of bpel engines for self-healing composite web services. In *2008 Intl. Symp. on Applications and the Internet*, pages 33–39, 2008.