Constraint Integration and Violation Handling for BPEL Processes

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

Abstract

Autonomic, i.e. dynamic and fault-tolerant Web service composition is a requirement resulting from recent developments such as on-demand services. In the context of planning-based service composition, multi-agent planning and dynamic error handling are still unresolved problems. Recently, business rule and constraint management has been looked at for enterprise SOA to add business flexibility. This paper proposes a constraint integration and violation handling technique for dynamic service composition. Higher degrees of reliability and fault-tolerance, but also performance for autonomously composed WS-BPEL processes are the objectives.

1. Introduction

Service-oriented architecture is a paradigm for software development. Web services are platform-independent, reusable components of business processes. The Business Process Execution Language for Web Services (WS-BPEL) has become the de-facto standard for service-based workflow description and execution. However, service composition is a complex task and is a challenge if composition problems have to be dealt with dynamically. Recent developments such as on-demand service composition are examples for the need to address dynamic fault-tolerance compositions. Therefore, building composite Web services with an autonomic tool is critical [1,2].

There have been some advances based on achievements in the artificial intelligence context, such as Semantic Web-based planning and reasoning. Business rule management has been adapted from expert systems and other AI sub-disciplines to enterprise SOA [3]. Rules or constraints, which restrict the states and transitions the process can go through to satisfy specified goals, add flexibility, but also and more importantly, reliability and fault-tolerance if possible constraint violations can be dealt with dynamically. Constraint integration and violation handling for dynamic service compositions is our focus. Due to the distributed and heterogeneous nature of Web service compositions, we cannot assume compositions to be stable. In order to provide true autonomy, both business level constraints as well as technical runtime failures have to be dealt with. Autonomic service composition needs to be capable of monitoring and recovery from both types.

Semantic Web service technologies, such as service ontologies like OWL-S, combined with AI planning have been suggested as a solution to automated service composition: [4] uses a Golog planner based on situation calculus, [5] uses SHOP2 with Hierarchical Task Networking (HTN) planning. However, current implementations are still facing some problems:

- Concurrent resource access problems arise from multi-agent collaboration rather than single-agent planning [4,6,7]. Multiple planners should plan and execute workflow processes simultaneously while ensuring mutual exclusion.
- The ability to resolve failures is often lacking. Most approaches do not provide failure resolution [6]. [4] offers a middle-ground strategy to avoid service rollback problems during planning. However, fault handling during service execution still is missing.

For dynamic, fault-tolerant service composition, an interleaved approach to composition and constraint-based execution and failure handling is an appropriate approach in a number of application domains [4,6]. In this paper, we use planning-based service composition as a sample context to demonstrate a constraint integration and violation handling technique for autonomic service composition. We demonstrate how the WS-BPEL fault handling mechanism can be used for both of business constraint violations and runtime exceptions – by offering a constraint violation monitor integratable with any standard BPEL engine in order to avoid overheads of additional supervision and monitoring processes. We present a mechanism to instrument a service composition with constraint violation handling. Our aims are to demonstrate the feasibility of the identified weaknesses and that fault-tolerance can be achieved, and to show that this can be addressed with acceptable overheads.

This paper is structured as following. Section 2 analyses of interleaving approaches for dynamic service composition. Section 3 analyses requirements for a service notion for dynamic composition. Section 4 describes the architecture and data collection approach. Section 5 defines our process instrumentation algorithm for service composition. Section 6 evaluates the approach based on a case study. We end with a discussion of related work and some conclusions.

2. Interleaving Approaches for Service Composition and Execution

Extended goals, observation and non-determinism are three difficulties which a composition technique needs to address in automatic service composition [8]. An extended goal approach allows a planer to consider a set of loosely coupled goals as single planning problem. This is a built-in ability of planners, which we will not discuss here, but observation and non-determinism are important. For service composition, a composition planner often has incomplete information initially [9]. Planners need observers or gather information. As a solution to this problem, an interleaved approach integrates service execution as part of a planning and composition process (Fig.1). The composition tool dynamically queries the environment for determinations rather than searching for all possibilities in a tree-like conditional plan.

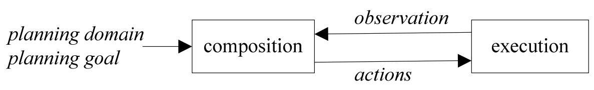

Fig.1. Interleaved composition and execution

The interleaved approach offers a high degree of adaptability for composition and execution [6]. Compositions generated can respond to changes in the run-time environment. For example, adding or deleting services or changing QoS properties of services can impact the creation and execution of the composite service. A case study in [6] shows an interleaved approach is most appropriate in dynamic scenarios since it can effectively address the functional and non-functional aspects of realizing the user request in an integrated setting. Interleaving could even improve planning problems with large search spaces.

To enable composition tools to gather information during the planning stage, we can consider primitive services as sensing actions or world-altering actions. Sensing services gather information, providing only (functional) output. World-altering services change the state of the world. In an interleaved approach, composition and execution go hand-in-hand. Therefore, resolving a failure requires rollback of the services executed when a planned execution path cannot reach a goal. A middle ground between online (composition) and offline (execution) can be introduced [4,5], i.e. a planner only executes sensing services during planning. A sequenced final plan, which contains world-altering services, is executed after planning has ended.

When sensing and world-altering is separated, i.e. the middle ground online composition and offline execution is applied, a concurrency problem appears in relation to sharing resources with multi-agent planning. For example, both planer A and planer B might find (sense) one flight available at the same time and both create a plan to book the flight. If there is only one seat left, the flight will be overbooked. Usually, a simple condition can be declared: ensure that sensed information persists during the execution and that none of the actions in the program cause it to be violated. Further problems arise if the flight is booked, but afterwards hotel booking fails caused by for instance technical failures such as network, hardware, etc. Thus, multi-agents planning and fault recovery are two basic and critical concerns to be addressed.

3. Service Categories

The distinction into sensing and world-altering is central for interleaved composition. A basic service model is insufficient for constraint integration. We expanded a primitive service notion as following.

- Primitive services: an executable primitive service *S* is either a sensing service *Sense* or a world-altering service *Alter*.
- World-altering services: an executable altering service *S* is either a non-conditional world-altering service *Alter-NC* or a conditional world-altering service *Alter-C*. There shall be at least one service *Rollback* that can rollback the effect of executing a service *S* and does not depend on any state of world for execution and *Alter* is its inverse.
- Conditional world-altering services: *Alter-C* consists of conditional checking functions *SenseFct* associated with a set of defined exceptions *E* (fault messages), followed by a non-conditional world-altering function *AlterFct-NC*.

A failure of *SenseFct* causes thread termination and *E* to be thrown.

In order to deal with concurrency problems, we define *Alter-C* for world-altering services that depends on observations or other conditions. Based on the conditions *SenseFct* as a constrained guard for *AlterFct-NC*, an exception mechanism will indicate any failure of these constraints. These constraints are essential, since the result of not meeting them is that *AlterFct-NC* will not be executed. There is also for every *Alter* service at least one rollback service.

4. Constraint Violation Handling

4.1. Architecture

Since we use a message-based, standard BPEL exception and fault handling mechanism, we are able to collect (sense) constraint violation data without modification of BPEL engines. A fault caused by service failure or any other reasons will be caught by fault handlers and viewed as a type of violation. A violation analysis service *Analyse* provides a constraint violation analysis. Fig. 2 illustrates constraint violation handling. A simple sequenced plan is converted into a WS-BPEL process with conditional paths for execution (see detail in Section 5). Data collection for constraint violation handling is triggered by WS-BPEL fault handlers. *Analyse* cooperates with a constraint engine to make decisions on validation handling.

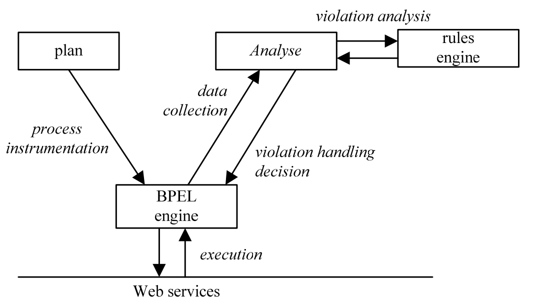

Fig.2. Constraints violation analysis and handling

An Object Constraint Language (OCL) checker (an open source component of the Eclipse EMF validation framework) is used as our constraint engine. OCL is an OMG defined constraint language used to constrain Meta-Object Facility based model and based on set theory and logic [10].

```
Goal.allInstances()
→select(e|e.name='TravelReservation')→first().criticals
 →select(ec|ec.category='bookTaxiFailure')=set{}
```

```
Goal.allInstances()
 →select(e|e.name='LoanApprover')→first().criticals
 →select(ec|ec.category='highRisk')=set{}
```

OCL constraints, as these two, can capture business goals or technical platform requirements and are checked dynamically during service process execution.

4.2. Fault handling and data collection

BPEL engines usually provide a service-based administration API to check processes that are deployed, running and completed, variable values etc. Three types of data are required for our application: *fault message*, *fault element* and *process execution instance log*. The instance log is only needed for recovery as the system provides a list of completed services for rollback or compensation.

Fault handlers define how BPEL processes respond when a Web services return error messages or other exceptions. *Analyse* is triggered by a fault handler and sends a query to the admin API for fault messages. This approach slightly depends on the BPEL engine, as admin APIs are different in different engines. Thus, *Analyse* can either be engine dependable or configurable. Two popular open-source engines are Apaches ODE (v1.2) and ActiveBPEL (v5.0.2).

	Apache ODE	ActiveBPEL
queryProcessInstance	√	√
queryFaultElement	√	√
queryFaultMessage	√	X

Since both admin APIs are only able to query instance logs by default, we developed an extension for extracting faultElement and faultMessage. In addition, ActiveBPEL only returns an 'EXECUT_FAULT' instead a detailed and defined fault message. Thus, a message-based analysis is unavailable. We can, however, modify ActiveBPEL to return detailed fault messages. We found query instant logs for fault message to be much slower than using fault handlers directly to forward fault message to *Analyse*.

As our solution (see Fig.3), we use <catch> for business exceptions thrown from service interface. A fault variable constraintViolation will be forwarded to *Analyse*. In addition, we use <catchAll> for the rest of the platform-specific runtime exceptions. We only query the instance log when recovery is necessary.

```
<bpel:faultHandlers>
  <bpel:catch faultMessageType="ns1:Exception"
     faultName="ns1:Exception"
```

```
    faultVariable="constraintViolation">
    …
    </bpel:catch>
    <bpel:catchAll>…<bpel:catchAll>
<bpel:faultHandlers>
```

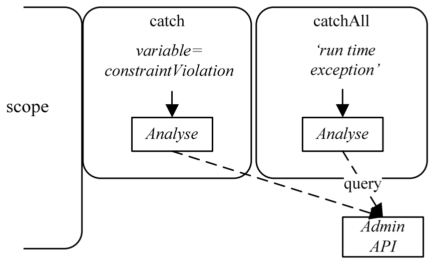

Fig. 3. Fault handling and data collection

5. Service Process Instrumentation

To integrate and enable the violation analysis, *Analyse* is trigged by fault handling events, and workflow execution steps are followed by an *Analyse* decision. For this to take place, we need to instrument the sequenced composition plan before its execution with the fault handling and constraint validation. We propose an algorithm with an execution path parameter. We create a conditional composition plan with all possible paths, which for decisions of *Analyse* include the necessary recovery as well. Except a default path, the other paths will only be executed based on a corresponding response of *Analyse*.

In additional to *Analyse*, we have a replanning service *Replan* for full recovery and recomposition. Recomposition is an execution fault recovery method by reconstructing a fresh plan to achieve the same planning goal. The faulty process will be retired and a new plan is started. We have developed the instrumentation and recomposition with the aim of keeping the delay on the client side minimal [11]. The instrumentation algorithm for a synchronized interleaved service process uses both fault variable and the admin API for data collection.

Process Instrumentation Algorithm

Input: *final plan (p)*
Output: *executable plan (bpel_p)*

```
start new bpel_scope
  start new bpel_faultHandler catchall
    new bpel_invoke operation=logging
  end bpel_faultHandler
  new bpel_receive
  AnalyseResponse=1
  start new bpel_repeatUntil condition AnalysisResponse==0
    start new bpel_scope
      start new bpel_faultHandler
                    catch variable='constraintViolation'
        new bpel_invoke operation = Analysis
      end bpel_faultHandler
      start new bpel_faultHandler catchAll
        new bpel_invoke operation = Analysis
      end bpel_faultHandler
      init path=0
      while path<size(p)
        path=path+1
        start new bpel_if  condition AnalysisResponse==path
          for each S in p(path,size(p))
            new bpel_invoke operation=S
          end for
          new bpel_assign AnalysisResponse==0
        end new bpel_if
      end while
      set path=0
      for each Alter S in p(path,size(p))
        path =path+1
      start new bpel_if
                    condition AnalysisResponse=='Rb'+path
        new bpel_invoke operation = R-Alter
        new bpel_invoke operation = Analysis
      end bpel_if
    end for
    start new bpel_if  condition AnalysisResponse=='Re'
      new bpel_invoke operation = RePlan
    endbpel_if
    end bpel_scope
    new bpel_reply
end bpel_scope
```

The algorithm builds three categories of paths.

- The first category addresses constraint violations that are acceptable, i.e. the composed process continues be executed. The numbers of path equals the size of the plan *size(p)*. Initially, *path* contains all *S* in *p*. It is the default path and will achieve the planning goal if completed without fault. *path+1* does not include the previous *S*, assuming a fault message thrown by *S*. *path+1* is created recursively until no candidate *S* is left in the *path*.
- The second category are rollback paths for *Alter* when constraint violations are not acceptable. In Section 3, we assumed each corresponding service to have one rollback service *R*. Each rollback path *r* is composed by one *R* for each *Alter* in *P* and followed by an *Analyse* to determine if more *Alter* invocations require rollback.
- The third category are recomposition paths. The path is composed of one *Analyse* for recovery by recomposition. A *bpel_scope* constrains all paths. Fault handlers are located within a *bpel_repeatUntil* container, a repeat condition that only expires if *path==0*. A detailed BPEL example is presented in Section 6.

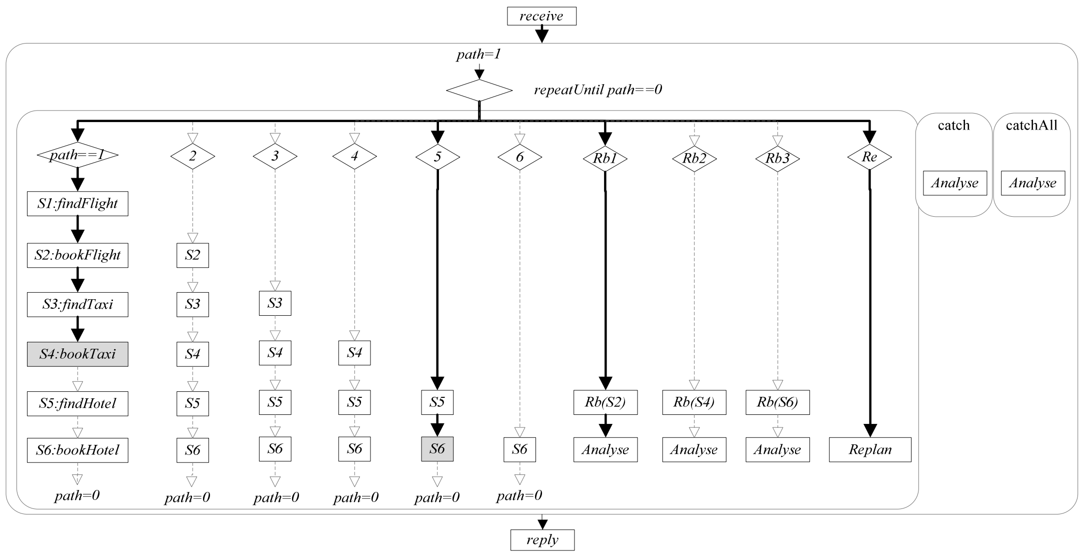

Fig. 4. Travel reservation process

6. Case Study and Evaluation

We use a Travel Reservation system as our case study to support the evaluation. The aim of the case study is demonstrate that our approach addresses problems in planning based service composition. Particularly, we look at performance and overheads of constraint violation handling.

6.1. Execution scenario

Fig. 4 illustrates a travel reservation process. The black lines represent a workflow execution scenario. The default *path=1* is executed initially. After booking a flight, a taxi is booked for pickup on arrival. However, *S4:bookTaxi* might fail. *Analyse* is invoked due to a fault being caught. *bookTaxiFailure* might be an acceptable violation. *Analyse* returns a *path=5* for continued execution. However, a *hotelOverBooking* exception caused by a concurrency problem might be thrown by *S6:bookHotel* – an unacceptable violation. *Analyse* starts to rollback the last successful *Alter* service *S2:bookFlight* and analyses again to see if more altering services require rollback. If all rollback actions are completed, the final recomposition is activated.

6.2. Evaluation

Based on our experiments, we can demonstrate that our approach solves the concurrency problem in multi-agents planning (e.g. *hotelOverBooking*) and is fault-tolerant. For the latter, we included manually operated runtime errors, including randomly turning off services in our evaluation.

Regarding performance, process instrumentation does make *bpel_p* more complex and larger than the original plan *p*. However, the size is still acceptable. Let *size(p)* be the number of services in *p* and *l* the number of *Alter* services in *p*, then

$$size(bpel\_p) = \sum_{k=1}^{n=size(p)} k + 2l + 5$$

$$paths(bpel\_p) = size(p) + l + 1$$

The formulas show the sizes of service related activities *size(bpel_p)* (BPEL assign, link, etc are not counted) and the number of paths *path(bpel_p)* in *bpel_p*. In the first formula, 5 activities *bpel_receive*, *bpel_reply*, *Analyse*, *Replan*, and *logging* are represented. In the second formula, the constant 1 represents the recomposition path. Our evaluation platform is a Pentium E2140 CPU, 1GB DDR2 memory, Window XP sp3, ActiveBPEL engine, enhanced BPEL generation (includes BPEL engine deployment file). Instrumentations only take 82 milliseconds in average.

In our experiments, querying the admin API takes in average 1672 milliseconds. Message passing by fault variables only takes in average 134 milliseconds. Querying would cause some overheads when the violation is acceptable and no recovery is needed.

For constraint violation handling, the total overhead depends on the plan execution time, the number of constraint violations, and the constraint engine performance, etc. In the case study scenario, there is only a violation handling overhead of in average

10.2% in a total plan execution time of in average 17281 milliseconds (not including replanning). Since all Web services are on hosted locally, we expect overheads in networked environments to be much lower.

7. Discussion and Related Work

The solution that we have implemented through our prototype demonstrates that BPEL fault handling mechanisms can support a high-performance constraint violation handling based on standard BPEL engines. While constraints integration solves some problems, some challenges have also arisen. To provide flexibility in business processes, various types of constraints are required. Constraints need to be at a context model level to capture business and technical aspects and also need to be integrated dynamically. Our current prototype is able to integrate context constraints into supporting services and weave these into BPEL processes for violation handling.

Some related work was already covered in Section 2. [4, 5, 8] provide a solution of using various planning techniques for dynamic service composition. However they often lack multi-agents planning and fault-tolerance. Researchers are also looking into constraint integration and monitoring platforms. In [12], a constraint language is proposed for the Dynamo monitoring platform. However, we differ from that approach, as we use a more simpler and more efficient standard BPEL fault handling without requiring additional execution monitoring subsystems. We also provide enough flexibility towards a complete autonomic service composition rather than defining recovery rules for each service.

8. Conclusion

We have presented an execution failure and error handling approach for dynamic, fault-tolerant service composition. We have developed a constraint integration and violation handling technique based on the WS-BPEL fault handling mechanism. An algorithm that generates an instrumentation of the composed service process is at the core of our solution. The evaluation result shows good performance and little overheads, avoiding additional supervision monitoring processes and components. Although presented in the context of AI planning for composition, other approaches such as brokers and mediator can also be enhanced using our solution. We have focussed on planning here to address known weaknesses and evaluate its benefits for dynamic service composition with constraints integration and violation handling.

Acknowledgements

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

References

[1] J. Rao, and X. Su, "A Survey of Automated Web Service Composition Methods," *Semantic Web Services and Web Process Composition*, Springer Berlin, 2005.

[2] S.-C. Oh, D. Lee, and S. R. T. Kumara, “A Comparative Illustration of AI Planning-based Web Services Composition,” *ACM SIGecom Exchanges*, 2005.

[3] M. El Kharbili, and T. Keil, “Bringing Agility to Business Process Management: Rules Deployment in an SOA,” *The 6th IEEE European Conference on Web Services*, Business Track. 2008.

[4] T.C. Sun, and Sheila McIlraith, "Adapting Golog for Programming the Semantic Web," *In Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, pp. 195-202, 2001.

[5] D. Wu, E. Sirin, J. Hendler *et al.*, “Automatic Web Services Composition Using SHOP2,” *Workshop on Planning for Web Services*, 2003.

[6] V. Agarwal, G. Chafle, S. Mittal *et al.*, “Understanding Approaches for Web Service Composition and Execution,” *Proceedings of the 1st Bangalore Annual Computing Conference*, 2007.

[7] A. Polleres. "AI Planning for Semantic Web Service Composition? - presentation," http://www.wsmo.org/.

[8] M. Pistore, F. Barbon, P. Bertoli *et al.*, “Planning and Monitoring Web Service Composition,” *Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

[9] S. McIlraith, and T.C. Son, “Adapting Golog for Composition of Semantic Web services” *Intl Conf Principles of Knowledge Representation and Reasoning*, 2002.

[10] J. Warmer, and A. Kleppe, *The Object Constraint Language*: Addison-Wesley Professional, 2003.

[11] C. Moore, M.W. Xue, and C. Pahl, “An Architecture for Autonomic Web Servive Process Planning,” *3rd Workshop on Emerging Web Services Technology*, 2008.

[12] L. Baresi, S. Guinea, and L. Pasquale, “Towards a Unified Framework for the Monitoring and Recovery of BPEL Processes,” *Workshop on Testing, analysis, and verification of web services and applications*, 2008.