Dynamic Architectural Constraints Monitoring and Reconfiguration in Service Architectures

Jose John, MingXue Wang, Claus Pahl

Lero and School of Computing, Dublin City University
Dublin, Ireland
jose.john2@mail.dcu.ie, mwang@computing.dcu.ie, cpahl@computing.dcu.ie

Abstract. Service-oriented architecture is an architectural approach that can be applied for building autonomous service systems dynamically to satisfy on-demand business requests. During the execution of service compositions, architectural constraint violations relating to functional and non-fucntional system properties need to be handled intelligently and autonomously, possibly requiring architectural reconfigurations. We propose integrated architectural constraint violation handling to deal with architectural quality problems through dynamic reconfiguration. We concentrate on service replacement selection as a remedial strategy for a possible quality violation requiring architectural remedies.

1 Introduction

Service-oriented architecture (SOA) allows us to build interoperable distributed systems. Service processes are build using orchestration languages like WS-BPEL. Composing processes dynamically is a solution for on-demand requests. Dynamic reconfiguration is often the consequence of faults (e.g., caused by the violation of architectural constraints). The severity of some faults might not allow a service to be used further. BPEL provides fault handling mechanisms, but no remedial mechanisms. A solution is to dynamically select a remedial strategy. Architectural constraint violations indicating quality problems are important faults that can occur during execution [12].

Our solution is an operationalisation of dynamic service architecture through an architectural quality monitoring instrumentation of processes using the WS-BPEL fault handling mechanism. Fault and violation handling based on dynamically available architectural knowledge in the form of quality-oriented service annotations acts here as a framework for dynamic architectural decision making:

- A dynamic remedial strategy selection mechanism. In [11], remedial strategies are proposed for business constraint violations and runtime faults, which are mapped to architectural remedial strategies for reconfiguration. This paper focuses on the service replacement remedial strategy.
- Service replacement selection based on a service quality annotation scheme. The annotation scheme captures different architectural properties for each replacement service. When a quality constraint violation occurs, the annotation scheme will be searched for a suitable replacement.

We focus on the operationalisation of dynamic selection techniques. Based on an empirical study, we have identified a number of properties that can be used for the annotation scheme of recomposable services. We introduce a similarity metric based on an aggregated distance, which is used for selecting a suitable replacement. We also use a history-based success ranking heuristics as a weighting mechanism to further discriminate between replacement candidate services.

Section 2 introduces service fault handling. In Section 3, we outline our architecture. We define the annotation scheme in Section 4, the selection mechanism in Section 5, and the monitoring and violation handling implementation in Section 6. Finally, we discuss our implementation and conclusions are given.

2 Service-oriented Architecture and Service Composition

Constraint Violation and Fault Monitoring. If we compose services dynamically based on on-demand user requests, we can customise services based on user profiles or remedy requirements validation [9]. BPEL process instances interact with the constituent web services through invoking various activities. Normally, the process ends its execution with a reply activity. During the execution of a process, faults can occur. One category of faults are technical runtime exceptions which are thrown by the BPEL engine itself. There are also business or requirements constraint violations. The faults can be the consequence of violations of architectural quality constraints or can impact on these. Quality constraints need to be monitored and faults need to be handled appropriately so that the composed process do not fail. Fault monitoring detects faults and records data for analysis. We use BPEL fault handlers for architectural constraints monitoring and fault handling. BPEL has fault handlers for handling specific faults (<*catch*>) and for handling all kinds of faults (<*catchAll*>). We will use a constraint monitoring and fault handling framework to monitor architectural quality (expressed as architectural constraints) and handle violations by recomposing the service process.

Fault Analysis. Used for finding the best remedial strategy for a fault instance, it takes fault data as input and outputs a strategy. Pre-defined remedial knowledge is used for fault analysis. Defining remedial knowledge involves three steps: defining a fault taxonomy, defining remedial strategies, and matching each fault category with remedial strategies. The types of faults that can occur define a fault taxonomy [1],[4]. In order to deal with business constraint validations, a fault taxonomy is derived from the context model which is used for constraint validation services. Remedial strategies like process goal-preserving retry, replace, ignore or recompose [1],[11] are selected and applied dynamically:

- Ignore: this strategy completely ignores the fault occurred. This is suitable for faults that do not have any effect on the overall architectural goal.
- Retry: this strategy tries to execute the faulted service again. Maximum retries and the retry interval can be defined.
- Replace: this strategy replaces the faulty service with a suitable one with same the business functionality.

– Recompose: this strategy discards the entire faulty process and establishes a new process with the same architectural goal.

Non-goal preserving strategies identified are log (the fault data is recorded), alert (concerned parties will be alerted) and suspend (suspends the faulty process based on a threshold value of past failure ratio). The fault taxonomy is mapped to the strategies. We can have two kinds of constraint violation faults, pre-condition constraint violation faults and post-condition constraint violation faults, which are validated before or after service execution, respectively.

3 Fault Handling Architecture

We focus on the replacement strategy in particular as it is the core activity in architectural reconfiguration – recomposition creates specific problems in terms of planning techniques [10] that go beyond the focus of this paper. Replacement requires additional supporting infrastructure for discovering alternatives. We can implement it in two ways: pre-assign a replacement service so that the strategy can be instantly applied or discover alternative services dynamically. This discovery can be based on functional and/or non-functional architectural annotations. We select an alternative service from a service repository, which may have multiple services which match the functionality of the faulty service. A decision which one to select is made by a selection mechanism.

We use BPEL fault handling to implement annotation and remedial activities. This avoids the overhead of BPEL engine-dependent modifications and additional monitoring components in order to reduce monitoring and fault handling overhead. We add validation services for business constraint validations. Constraint violations are thrown from these constraint services as service faults. This allows us to catch the architectural constraint violation faults in the BPEL fault handlers. Fig. 1 shows the architecture. Main layers identified are process execution layer, fault tolerance layer and database layer. The fault-tolerance layer contains monitoring, analysis (selection mechanism) and a service wrapper component. The database layer stores all available services to be considered for a possible replacement in a service repository. The annotation is stored in an annotation scheme database. The wrapper handles the invocation of the replacement service. The execution of the process happens at the process execution layer.

4 Service Quality Annotation Scheme

The annotation scheme is a central component that enables dynamic alternative service selection. Annotations of replacement services are kept in a dynamically accessible and updatable repository. The annotation scheme works based on operational (QoS) properties of services. The values of these architectural quality properties play a crucial role in the selection of a replacement service. We choose here three architecturally important properties which can be measured - all suitable for easy operationalisation:

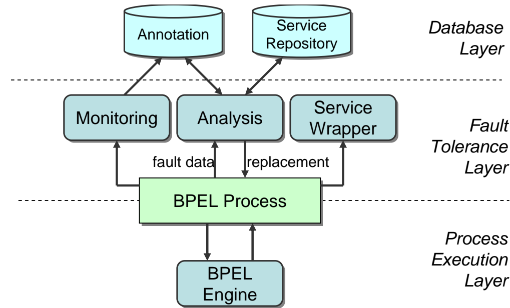

Fig. 1. Fault Handling Architecture.

- Response Time (Latency): This property measures the difference between the time a service request takes between the request and response. It can be calculated as follows: *Response Time = Response Completion Time - User Request Time*. Response Completion Time is the time when all data for a response arrives at the user. User Request Time is the time when the user sends a request. This is a measure of the performance of a web service.
- Availability: It is the time period in which the service is ready for use or the service is maintained. If the time when a system is not available is 'Down Time' and when its is available is 'Up Time', then availability is the average uptime. It can be measured using *Availability = 1 - (Down Time / Up Time)*.
- Accessibility: Accessibility represents the degree that a system is normatively operated to counteract request messages without delay. In some cases, a service system could be accessible for external users to try accessing its resources even if its services are not available. We can determine whether a web service system is accessible by just ensuring that the system can return an acknowledgment for a request message. Thus, accessibility can be calculated as the ratio of number of acknowledgments received to the number of request messages: *Accessibility = Number of Acknowledgments Received / Number of Request Messages*.

Other properties such as throughput and reliability, but also integrity, compliance and security are also considered to be important, but have not been addressed in our framework yet. We focus on selection based on operational criteria of architectural relevance, which capture classical quality-of-service properties in the first category (such as response time or accessibility). Maintaining architectural quality through quality monitoring and remedy is our objective.

5 Analysis and Selection Mechanism

The selection mechanism that we use here is based on the concept of an aggregated distance (AD). An aggregated distance is the sum of distances of all the annotated properties for a service. For each annotated property there is some

threshold value for the running process. A distance is the difference between this threshold value and the actual property value of the service. Let P_{i_j} be the value of the j-th property of the i-th replacement service P_i. T_j as the threshold value is defined for the j-th property in order to normalise the values. Then, the aggregated distance, a simple additive weighting, for the i-th service is

$$AD_i = \sum_{j=1}^{n_i} \frac{P_{i_j} - T_j}{max(P_{i_j}) - min(P_{i_j})}$$

for all properties j where max and min refer to the maximal and minimal values of each property in order to normalise each property in comparison to the other properties. AD_i shall be defined 1 where $max(P_{i_j}) - min(P_{i_j}) = 0$. The service with the least aggregated distance is the best replacement candidate.

In addition to the AD, we use a heuristic function to support the selection. A history-based success ranking system shall support the decision. The heuristic is in this case an approximation of the expected reliability of a service. If the post-constraint evaluation finishes without any exception, we increase the rank of the service by one. If the execution flow reaches the fault handler, the service has generated some fault and we decrease the rank by one. While selecting the replacement service, we take the service with highest rank into account as a weighting to discriminate between similarly valued services based on AD. We adjust the distance measure using the rank for service i, $rank(i)$, with 1 being the best and $\|rank\|$ denoting the total number of ranked services:

$$AD_i^{norm} = AD_i \times (1 + \frac{rank(i)}{\|rank\|})$$

This ranking-based weighting works as a passive recommendation system as it gives up-to-date feedback on each service. The normalised AD_i^{norm} value that is closest to the original AD_i is considered the best (lower ranked services would create a greater distance to AD_i).

6 Architectural Constraint Monitoring and Handling

Two aspects need to be distinguished: monitoring in order to keep the service annotations up to date and architectural constraint monitoring and handling.

Annotation Monitoring and Updating. The annotation scheme is kept up to date. The advantage of dynamic monitoring is that the selection mechanism can make decisions based on the latest information to increase the accuracy of selections. We monitor the response time of a service, measured as the time between the end of pre-condition validation to the start of post-condition validation. This time is updated for that service in the annotation scheme as the new response time. We are working on a constraint monitoring instrumentation that can be applied to provide measurement for the suggested quality properties.

Instrumentation Template for Constraint Handling. The implementation of the violation handling needs a BPEL process instrumentation that integrates

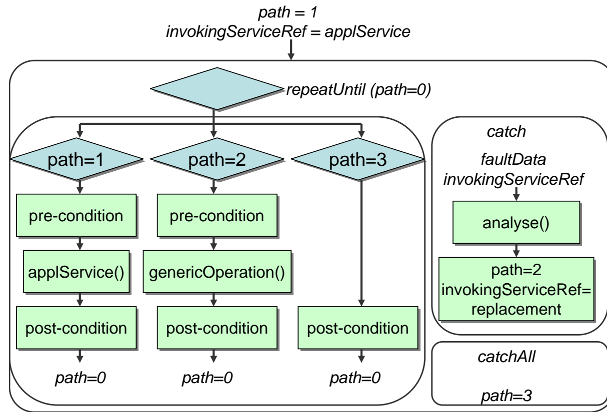

Fig. 2. Instrumentation Template.

fault handling and monitoring capabilities. To achieve this, we use constraint services. The instrumentation also applies the selected remedial strategies. We use a modified version of the instrumentation template which is used in [11]. Fig. 2 shows this modified instrumentation template. Two important variables are used in the instrumentation template, invokingServiceRef and path. invokingServiceRef holds a reference to the current activity which is invoked. invokingServiceRef is passed to both pre- and post-constraint services so that they can inspect the properties of the invoking service to see whether constraints are satisfied. Whenever there is a fault, the invokingServiceRef will be passed to the fault handlers for further analysis along with the fault data. There are two main execution paths in the template. In the default path (path=1), the original service is invoked along with the pre- and post-constraint services. If there is a fault, then the path variable is changed so that the execution follows the second path (path=2). Once an execution path is completed without faults, the path variable is assigned to 0 (path = 0) and the repeatUntil construct finishes.

The Replacement Strategy. Faults caused by pre-condition constraint violation are caught by the <*catch*> fault handler. It passes fault data as well as the invokingServiceRef variable to the analyse() service. Since the replacement strategy is applied, the analyse operation sets path=2. It also assigns a new service found by the selection to invokingServiceRef. This alternative service is called by the genericOperation() wrapper. Faults caused by the invoking service are caught by a <*catchAll*> handler. It sets path to 3. This path has a post-constraint validator which converts this fault into a constraint validation fault. This is caught by the <*catch*> and analyse() will run as in case 1.

7 Discussion

Our evaluation focus was on the effectiveness and performance of the monitoring and fault handling system. Performance is critical as the context is dynamic

architectural reconfiguration. Effectiveness and reliability are equally important in an autonomous setting.

The aggegrated distance approach essentially creates an attribute vector of normalised values (threshold), which based on a manual validation determines good candidate replacements. We have used the success ranking to have a second analysis stage to remove unsuitable cases. 35 test cases were designed for a payment process, which involves four business services (requestBill; payBill; updateRecords; infoProvider) to evaluate the remedial strategy as a whole. We developed three alternative services for each service to test the replacement remedies with alternative services. Test cases cover architectural constraint violations and runtime faults in the context of all proposed remedial strategies.

The overhead created for monitoring and updating annotation attributes did not exceed 9% of the overall execution time and the overhead for the violation was in average between 3 and 4% due to the embedding of the instrumentation into the BPEL fault handling. Only the access to the database for replacement selection was a significant element in the 9% figure. Each process was composed of a different number (2 to 10) of application services. We instrumented each process and created pairs of processes to compare their performance. The performance evaluation results show that the instrumented processes does not introduce any significant overhead (in average less than 1%). The instrumented processes do not delay the overall execution unless a fault needs to be handled.

Effectiveness, performance and reliability shall also be looked at in the context of related work. In [10], a solution using various planning techniques for dynamic service composition is provided. However, they lack comprehensive fault-tolerance mechanisms. Constraint integration and monitoring platforms has been looked at. In [2], a constraint language is proposed for the Dynamo monitoring platform. We use a simpler and more efficient standard BPEL fault handling without requiring additional execution monitoring subsystems.

Different remedial strategy selections have been proposed. An interesting approach [5] is to invoke all alternative services in parallel and select the one which gives back the first response. It allows to select the best service quickly, but causes computational and network overheads and has the risk of multiple transactions, which is avoided in our annotation repository-based solution.

Our selection approach is based on aggregated distances and heuristics as a basic recommendation mechanism. Recommendation system are based on the learning done by the system from user or system feedbacks [7]. While aggregated distances seem to perform well as a similarity measure in terms of determining effective replacements, agglomerative clustering algorithms (e.g. association coefficient based similarity measures) can also be used.

8 Conclusions

We have introduced an integrated constraint monitoring and violation handling mechanism for dynamic service compositions. Flexible service process orchestrations at runtime form the problem setting [8],[3]. We used replacement as

the basis of our remedial architecture strategy. We provided an instrumentation template to support the integrated fault monitoring and handling for architectural quality constraints. A quality-oriented selection mechanism has been implemented to select from the available replacement services. Architectural reconfiguration is a problem of technical fault-tolerance, but also the consideration of architectural compliance with respect to business rules.

An extension is a more intelligent selection strategy based on machine learning. We will also address access performance improvements for the selection mechanism. Storage of the annotation scheme is another point of improvement.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grants 03/CE2/I303_1 (Lero) and 07/RPF/CMSF429 (CASCAR).

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 *World Wide Web Conf*, 2006.
2. L. Baresi, S. Guinea, and L. Pasquale. Towards a unified framework for the monitoring and recovery of bpel processes. In *Workshop on Testing, analysis, and verification of web services and applications*, 2008.
3. R. Barrett, L. M. Patcas, J. Murphy, and C. Pahl. Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *International Conference on Web Engineering ICWE'06. Palo Alto, US*, pages 129–136. ACM Press, 2006.
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. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *32nd EUROMICRO Conf on Software Eng and Adv Applications*, 2006.
6. A. Liu, Q. Li, L. Huang, and M. Xiao. A declarative approach to enhancing the reliability of bpel processes. In *IEEE Intl. Conf. on Web Services*, 2007.
7. U. Manikrao and T. Prabhakar. Dynamic selection of Web services with recommendation system. In *Next Generation Web Services Practices*, 2005.
8. C. Pahl. A Formal Composition and Interaction Model for a Web Component Platform. In *Proc. ICALP Workshop on Formal Methods and Component Interaction FMCI'02*. Electronic Notes on Computer Science ENTCS Vol. 66 No. 4, 2002.
9. C. Pahl. Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. In *European Conference on Model-Driven Architecture Foundations and Applications ECMDA2005*, pages 88–102. Springer LNCS 3748, 2005.
10. M. Pistore, F. Barbon, and P. Bertoli. Planning and monitoring web service composition. *Workshop on Planning and Scheduling for Web and Grid Services*, 2004.
11. M. Wang, K. Y. Bandara, and C. Pahl. Integrated Constraint Violation Handling for Dynamic Service Composition. In *IEEE International Conference on Services Computing SCC'2009*, 2009.
12. L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.