Recommendation and Application of Fault Tolerance Patterns to Services

Tunyathorn Leelawatcharamas* and Twittie Senivongse*

Abstract

Service technology such as Web services has been one of the mainstream technologies in today's software development. Distributed services may suffer from communication problems or contain faults themselves, and hence service consumers may experience service interruption. A solution is to create services which can tolerate faults so that failures can be made transparent to the consumers. Since there are many patterns of software fault tolerance available, we end up with a question of which pattern should be applied to a particular service. This paper attempts to recommend to service developers the patterns for fault tolerant services. A recommendation model is proposed based on characteristics of the service itself and of the service provision environment. Once a fault tolerance pattern is chosen, a fault tolerant version of the service can be created as a WS-BPEL service. A software tool is developed to assist in pattern recommendation and generation of the fault tolerant service version.

Keyword: fault tolerance patterns, Web services, WS-BPEL

1. Introduction

Service technology has been one of the mainstream technologies in today's software development since it enables rapid flexible development and integration of software systems. The current Web services technology builds software upon basic building blocks called Web services. They are software units that provide certain functionalities over the Web and involve a set of interface and protocol standards, e.g. Web Service Definition Language (WSDL) for describing service interfaces, SOAP as a messaging protocol, and Business Process Execution Language (WS-BPEL) for describing business processes of collaborating services [1]. Like other software, services may suffer from communication problems or contain faults themselves, and hence service consumers may experience service interruption.

Different types of faults have been classified for services [2], [3], [4], and can be viewed roughly in three categories: (1) Logic faults comprise calculation faults, data content faults, and other logic-related faults thrown specifically by the service. Web service consumers can detect logic faults by WSDL fault messages or have a way to check correctness of service responses. (2) System and network faults are those that can be identified, for example, through HTTP status code and detected by execution environment, e.g., communication timeout, server error, service unavailable. (3) SLA faults are raised when services violate SLAs, e.g., response time requirements, even though functional requirements are fulfilled. For service providers, one of the main goals of service provision is service reliability. Services should be provided in a reliable execution environment and prepared for various faults so that failures can be made as transparent as possible to service consumers. Service designers should therefore design services with a fault tolerance mindset, expecting the unexpected and preparing to prevent and handle potential failures.

There are many fault tolerance patterns or exception handling strategies that can be applied to make software and systems more reliable. Common patterns involve how to handle or recover from failures, such as communication retry or the use of redundant system nodes. In a distributed services context, we end up with a question of which fault tolerance pattern should be applied to a particular service. We argue that not all patterns are equally appropriate for any services. This is due to the characteristics of each service including service semantics and the environment of service provision. In this paper, we propose a mathematical model that can assist service designers in designing fault tolerant versions of services. The model helps recommend which fault tolerance patterns are suitable for particular services. With a supporting tool, service designers can choose a recommended pattern and have fault tolerant versions of the services generated as WS-BPEL services.

Section 2 discusses related work in Web services fault tolerance. Section 3 lists fault tolerance patterns that are considered in our work. Characteristics of the services and condition of service provision that we use as criteria for pattern recommendation are given in Section 4. Section 5 presents how service designers can be assisted by the pattern recommendation model. The paper concludes in Section 6 with future outlook.

2. Related Work

A number of researches in the area of fault tolerance services address the application of fault tolerance patterns to WS-BPEL processes even though they may have a different use of fault tolerance terminology for similar patterns or strategies. For example, Dobson's work [5] is among the first in this area which proposes how to use BPEL language constructs to implement fault tolerant service invocation using four different patterns, i.e., retry, retry on a backup, and parallel invocations to different backups with voting on all responses or taking the

* *Computer Science Program, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.*

first response. Lau et al. [6] use BPEL to specify passive and active replication of services in a business process and also support a backup of BPEL engine itself. Liu et al. [2] propose a service framework which combines exception handling and transaction techniques to improve reliability of composite services. Service designers can specify exception handling logic for a particular service invocation as an Event-Condition-Action rule, and eight strategies are supported, i.e., ignore, notify, skip, retry, retryUntil, alternate, replicate, and wait. Thaisongsuwan and Senivongse [7] define the implementation of fault tolerance patterns, as classified by Hanmer [8], on BPEL processes. Nine of the architectural, detection, and recovery patterns are addressed, i.e., Units of Mitigation, Quarantine, Error Handler, Redundancy, Recovery Block, Limit Retries, Escalation, Roll-Forward, and Voting. These researches suggest that different patterns can be applied to different service invocations as appropriate but are not specific on when to apply which. Nevertheless we adopt their BPEL implementations of the patterns for the generation of our fault tolerant services.

Zheng and Lyu present interesting approaches to fault tolerant Web services which support strategies including retry, recovery block, N-version programming (i.e., parallel service invocations with voting on all responses), and active (i.e., parallel service invocations with taking the first response). For composite services, they propose a QoS model for fault tolerant service composition which helps determine which combination of the fault tolerance strategies gives a composite service the optimal quality [9]. In the context of individual Web services, they propose a dynamic fault tolerance strategy selection for a service [3]; the optimal strategy is one that gives optimal service roundtrip time and failure rate. Both user-defined service constraints and current QoS information of the service are considered in the selection algorithm. In [10], they view fault tolerance strategies as time-redundancy and space-redundancy (i.e., passive and active replication) as well as combination of those strategies. Although their approaches and ours share the same motivation, their fault tolerance strategy selection requires an architecture that supports service QoS monitoring and provision of replica services. This could be too much to afford for strategy selection, for example, if it turns out that expensive strategies involving replica nodes are not appropriate. This paper can be complementary to their approach but it is more lightweight by merely recommending which fault tolerance strategies are likely to match service characteristics that are of concern to service designers.

3. Fault Tolerance Patterns

In our approach, the following fault tolerance patterns are supported (Figure 1). They are addressed in Section II and can be expressed using BPEL which is the target implementation of our fault tolerant services. Here the term “service” to which a pattern will be applied refers to the smallest unit of service provision, e.g., an operation of a Web service implementation.

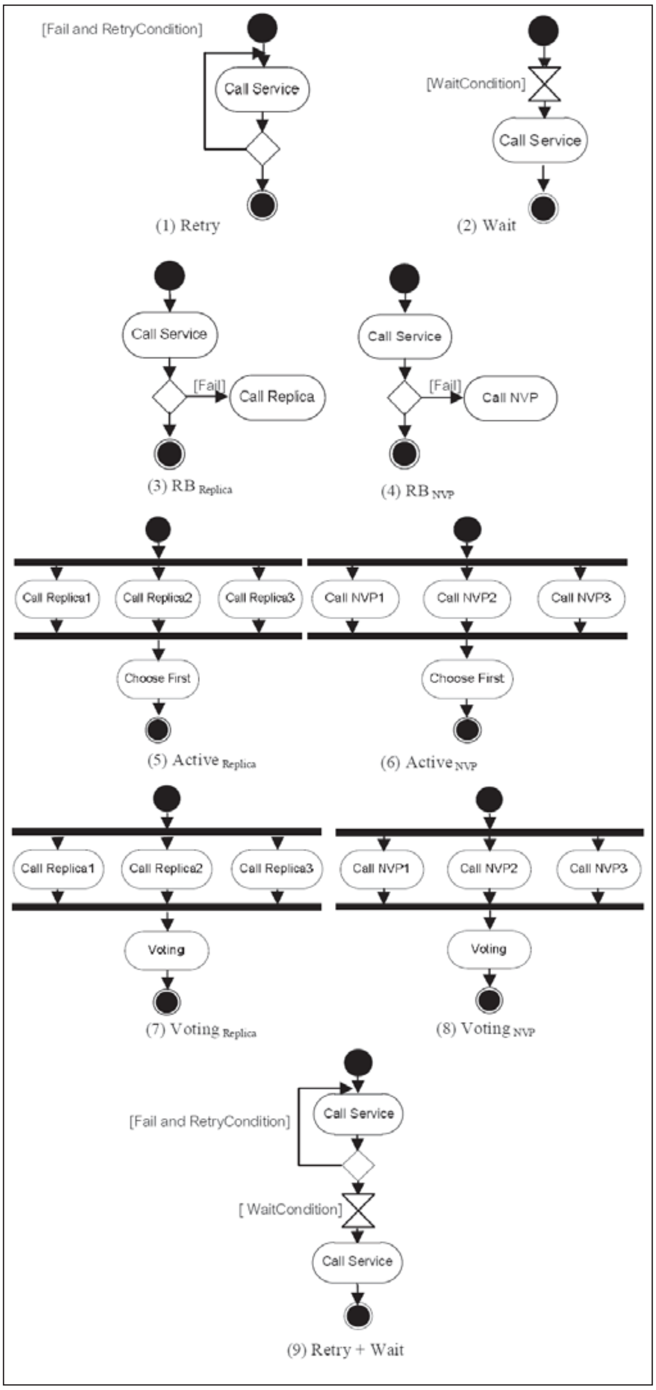


Figure 1. *Fault tolerance patterns.*

1) Retry: When service invocation is not successful, invocation to the same service is repeated until it succeeds or a condition is evaluated to true. A common condition is the allowed retry times.

2) Wait: Service invocation is delayed until a specified time. If the service is expected to be busy or unavailable at a particular time, delaying invocation until a later time could help decrease failure probability.

3) RecoveryBlockReplica: When service invocation is not successful, invocation is made sequentially to a number of functionally equivalent alternatives (i.e., recovery blocks) until the invocation succeeds or all alternatives are used. Here

the alternatives are replicas of the original service; they can be different copies of the orignal service but are provided in different execution environments.

4) RecoveryBlockNVP: This pattern is similar to 3) but adopts N-version programming (NVP). Here the original service and its alternatives are developed by different development teams or with different technologies, algorithms, or programming languages, and they may be provided in the same or different execution environment. This would be more reliable than having replicas of the original services as alternatives since it can decrease the failure probability caused by faults in the original service.

5) ActiveReplica: To increase the probability that service invocation will return in a timely manner, invocation is made to a group of functionally equivalent services in parallel. The first successful response from any service is taken as the invocation result. Here the group are replicas of each other; they can be different copies of the same service but are provided in different execution environments.

6) ActiveNVP: This pattern is similar to 5) but adopts NVP. Here the services in the group are developed by different development teams or with different technologies, algorithms, or programming languages, and they may be provided in the same or different execution environment. This would be more reliable than having the group as replicas of each other since it can decrease the failure probability caused by faults in the replicas.

7) VotingReplica: To increase the probability that service invocation will return a correct result despite service faults, invocation is made to a group of functionally equivalent services in parallel. Given that there will be several responses from the group, one of the voting algorithms can be used to determine the final result of the invocation, e.g. majority voting. Here the group are replicas of each other; they can be different copies of the same service but are provided in different execution environments.

8) VotingNVP: This pattern is similar to 7) but adopts NVP. Here the services in the group are developed by different development teams or with different technologies, algorithms, or programming languages, but they may be provided in the same or different execution environment.

9) Retry + Wait: This pattern is an example of a possible combination of different patterns. When service invocation is not successful, invocation is retried for a number of times and, if still unsuccessful, waits until a specified time before another invocation is made.

All patterns except Wait employ redundancy. Retry is a form of time redundancy taking extra communication time to tolerate faults whereas RecoveryBlock, Active, and Voting employ space redundancy using extra resources to mask faults [10]. RecoveryBlock uses the passive replication technique; invocation is made to the original (primary) service first and alternatives (backup services) will be invoked only if the original service or other alternatives fail. Active and Voting both use the active replication technique; all services in a group execute a service request simultaneously, but they determine the final result differently. Retry, Wait, and RecoveryBlock can help tolerate system and network faults. Voting can be used to mask logic faults, e.g., when majority voting is used and the majority of service responses are correct. It can even detect logic faults if a correct response is known. Active can help with SLA faults that relate to late service responses.

4. Service Characteristics

The following are the criteria regarding service characteristics and condition of service execution environment which the service designer/provider will consider for a particular service. These characteristics will influence the recommendation of fault tolerance patterns for the service.

1) Transient Failure: The service environment is generally reliable and potential failure would only be transient. For example, the service may be inaccessible at times due to network problems, but a retry or invocation after a wait should be successful.

2) Instance Specificity: The service is specific and consumers are tied to use this particular service. It can be that there are no equivalent services provided by other providers, or the service maintains specific data of the consumers. For example, a CheckBalance service of a bank is specific because a customer can only check an account balance through the service of this bank with which he/she has an account, and not through the services of other banks.

3) Replica Provision: This relates to the ability of the service designer/provider to accommodate different replicas of the service. The replicas should be provided in different execution environments, e.g., on different machines or processing different copies of data. This ability helps improve reliability since service provision does not rely on a single service.

4) NVP Provision: This relates to the ability of the service designer/provider to accommodate different versions of the service. The service versions may be developed by different development teams or with different technologies, algorithms, or programming languages, and they may be provided in the same or different execution environment. This ability helps improve reliability since service provision does not rely on any single version of the service.

5) Correctness: The service designer expects that the service and execution environment should be managed to provide correct results. This relates to the quality of service environment to provide reliable communication, including the mechanisms to check for correctness of messages even in the presence of logic faults.

6) Timeliness: The service designer expects that the service and execution environment should be managed to react quickly to requests and give timely results.

7) Simplicity: The service designer/provider may be concerned with simplicity of the service. Provision for fault tolerance can complicate service logic, add more interactions to the service, and increase latency of service access. When service provision is more complex, more faults can be introduced.

8) Economy: The service designer/provider may be concerned with the economy of making the service fault tolerant. Fault tolerance patterns consume extra time, costs, and computing resources. For example, sequential invocation

is cheaper than parallel invocation to the group of services, and providing relplicas of the service is cheaper than NVP.

5. Fault Tolerance Patterns Recommendation

The recommendation of fault tolerance patterns to a service is based on what characteristics the service possesses and which patterns suit such characteristics.

5.1 Service Characteristics-Fault Tolerance Patterns Relationship

We first define a relationship between service characteristics and fault tolerance patterns as in Table 1. Each cell of the table represents the relationship level, i.e., how well the pattern can respond to the service characteristic. The relationship level ranges from 0 to 8 since there are eight basic patterns. Level 8 means the pattern responds very well to the characteristic, level 7 responds well, and so on. Level 0 means there is no relationship between the pattern and service characteristic.

For example, for Economy, Retry and Wait are cheaper than other patterns that employ space redundancy since both of them require only one service implementation. But Wait responds best to economy (i.e., level 8) since there is only a single call to the service whereas Retry involves multiple invocations (i.e., level 7). Sequential invocation in RecoveryBlock is cheaper than parallel invocation in Active and Voting because not all service implementations will have to be invoked; a particular alternative of the service will be invoked only if the original service and other alternatives fail, whereas parallel invocation requires that different service implementations be invoked simultaneously. Recovery Block Replica (level 6) is cheaper than RecoveryBlockNVP (level 5) because providing replicas of the service should cost less than development of NVP. Similarly ActiveReplica (level 4) is cheaper than ActiveNVP (level 3) and VotingReplica (level 2) is cheaper than VotingNVP (level 1). Note that Voting is more expensive than Active due to development of a voting algorithm to determine the final result. For a combination of patterns such as Retry+Wait, the relationship level is an average of the levels of the combining patterns.

Table 1. *Relationship between Service Characteristics and Fault Tolerance Patterns*

Service Characteristics	Fault Tolerance Patterns								
	Retry	*Wait*	*RB* *Replica*	*RB* *NVP*	*Active* *Replica*	*Active* *NVP*	*Voting* *Replica*	*Voting* *NVP*	*Retry+ Wait*
Transient Failure (TF)	8	7	0	0	0	0	0	0	7.5
Instance Specificity (IS)	8	8	7	6	5	4	5	4	8
Replica Provision (RP)	0	0	8	0	8	0	8	0	0
NVP Provision (NP)	0	0	0	8	0	8	0	8	0
Correctness (CO)	2	2	3	4	5	6	7	8	2
Timeliness (TI)	4	1	5	6	7	8	2	3	2.5
Simplicity (SI)	8	8	7	6	5	4	3	2	8
Economy (EC)	7	8	6	5	4	3	2	1	7.5

For the relationship between other characteristics and the patterns, we reason in a similar manner. Retry and Wait suit the environment with Transient Failure. The patterns that rely on the execution of a single service at a time respond better to Instance Specificity than those that employ multiple service implementations. Replica Provision and NVP Provision are relevant to the patterns that employ space redundancy. For Correctness, Voting is the best since it is the only pattern that can mask/detect byzantine failure (i.e., the case that the services give incorrect results). Active is better than RecoveryBlock with regard to byzantine failure because the chance of getting the result that is incorrect should be lower than the case of RecoveryBlock due to the fact that the result of Active can come from any one of the redundant services that are invoked in parallel. Retry and Wait do not suit Correctness since they rely on the execution of a single service. For Timeliness, the comparison of the patterns on time performance given in [2], [3] (ranked in descending order) is as follows: Active, RecoveryBlock, Retry, Voting, Wait. For Simplicity, the logic of Retry and Wait which involves a single service is the simplest.

5.2 Assessment of Service Characteristics

The next step is to have the service designer assess what characteristics the service possesses; the characteristics would influence pattern recommendation.

1) Identify Dominant Characteristics: The service designer will consider service semantics and condition of service provision, and identify dominant characteristics that should influence pattern recommendation. For each characteristic that is of concern, the service designer defines a dominance level. Level 1 means the characteristic is the most dominant (i.e., ranked 1st), level 2 means less dominant (i.e., ranked 2nd), and so on. Level 0 means the service does not have the characteristic or the characteristic is of no concern.

For example, during the design of a CheckBalance service of a bank, the service designer considers Instance Specificity as the most dominant characteristic (i.e., dominance level 1) since bank customers would be tied to their bank accounts that are associated with this particular service. From experience, the designer sees that the computing environment of the bank provides a reliable service and if there is a problem, it is generally transient, and hence a simple fault handling strategy is preferred (i.e., Transient Failure and Simplicity have dominance level 2). Nevertheless, the designer is able to afford exact replicas of the service if something more serious happens (i.e., Replica Provision has dominance level 3). Suppose the designer is not concerned with other characteristics, then the others would have dominance level 0. Table 2 shows the dominance level of all characteristics of this CheckBalance service.

2) Convert Dominance Level to Dominance Weight:

a) Convert Dominance Level to Raw Score: The dominance level of each characteristic will be converted to a raw score. The most dominant characteristic gets the highest score which is equal to the dominance level of the least dominant characteristic that is considered. Less dominant characteristics get less scores accordingly. From the example of the CheckBalance service, Replica Provision has the least dominance level of 3, so the raw score of the most dominant characteristic – Instance Specificity – is 3. Then the score for Transient Failure and Simplicity would be 2, and Replica Provision gets 1. Table 3 shows the raw scores of the service characteristics.

b) Compute Dominance Weight: First, divide 1 by the summation of the raw scores. For example, for the CheckBalance service, the summation of the raw scores in Table III is 8 (2+3+1+0+0+0+2+0) and the quotient would be 1/8 (0.125). Then, multiply this quotient with the raw score of each characteristic. The result would be the dominance weights of the characteristics (where the summation of the weights is 1). The weights will be used later in the recommendation model. For the CheckBalance service, the dominance weights of all characteristics are shown in Table 4.

5.3 Fault Tolerance Patterns Recommendation Model

We propose a model for fault tolerance patterns recommendation as in (1)

Table 2. *Dominance levels of service characteristics.*

	Service Characteristics							
	Transient Failure	*Instance Specificity*	*Replica Provision*	*NVP Provision*	*Correct ness*	*Timeli ness*	*Simpli city*	*Econo my*
Level	2	1	3	0	0	0	2	0

Table 3. *Raw scores of service characteristics.*

	Service Characteristics							
	Transient Failure	*Instance Specificity*	*Replica Provision*	*NVP Provision*	*Correct ness*	*Timeli ness*	*Simpli city*	*Econo my*
Score	2	3	1	0	0	0	2	0

Table 4. *Dominance weights of service characteristics.*

	Service Characteristics							
	Transient Failure	*Instance Specificity*	*Replica Provision*	*NVP Provision*	*Correct ness*	*Timeli ness*	*Simpli city*	*Econo my*
Score	0.25	0.375	0.125	0	0	0	0.25	0

$$P = D \times R \qquad (1)$$

where P = A vector of fault tolerance pattern scores
D = A vector of dominance weights of service characteristics as computed in Section V.B
R = A relationship matrix between service characteristics and fault tolerance patterns as proposed in Section V.A

Therefore, given R as

$$R = \begin{array}{ccccccccc|c} Retry & Wait & RB_{Replica} & RB_{NVP} & Active_{Replica} & Active_{NVP} & Voting_{Replica} & Voting_{NVP} & Retry+Wait & \\ 8 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 7.5 & TF \\ 8 & 8 & 7 & 6 & 5 & 4 & 5 & 4 & 8 & IS \\ 0 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & 0 & RP \\ 0 & 0 & 0 & 8 & 0 & 8 & 0 & 8 & 0 & NP \\ 2 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 2 & CO \\ 4 & 1 & 5 & 6 & 7 & 8 & 2 & 3 & 2.5 & TI \\ 8 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 8 & SI \\ 7 & 8 & 6 & 5 & 4 & 3 & 2 & 1 & 7.5 & EC \end{array}$$

and, in the case of the CheckBalance service, D as

$$D = \begin{array}{cccccccc} TF & IS & RP & NP & CO & TI & SI & EC \\ 0.25 & 0.375 & 0.125 & 0 & 0 & 0 & 0.25 & 0 \end{array}$$

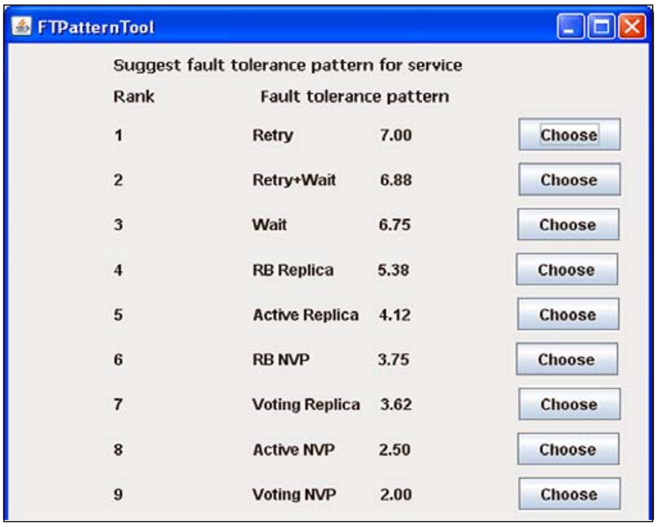
The pattern recommendation P would be

$$P = \begin{array}{ccccccccc} Retry & Wait & RB_{Replica} & RB_{NVP} & Active_{Replica} & Active_{NVP} & Voting_{Replica} & Voting_{NVP} & Retry+Wait \\ 7.00 & 6.75 & 5.38 & 3.75 & 4.12 & 2.50 & 3.62 & 2.00 & 6.88 \end{array}$$

The recommendation says how well each pattern suits the service according to the characteristic assessment. The pattern with the highest score would be best suited for the service. Since the designer of the CheckBalance service pays most attention to Instance Specificity, Transient Failure, and Simplicity, the designer inclines to rely on reliable provision of a single service. The patterns that respond well to these characteristics, i.e, Retry, Wait, and Retry+Wait, are among the first to be recommended. Here, Retry is the best-suited pattern with the highest score. Since the designer can provide replica services as well but still has simplicity in mind, RecoveryBlockReplica is the next to be recommended. Voting patterns and those which require NVP services are more complex strategies, so they get lower scores.

5.4 Generation of Fault Tolerant Service

A software tool has been developed to support fault tolerance patterns recommendation and generation of fault tolerant services as a BPEL service. The service designer will first be prompted to select service characteristics that are of interest, and then specify a dominance level for each chosen characteristic. The tool will calculate and rank the pattern scores as shown in Figure 2 for the CheckBalance service. The designer can choose one of the recommended patterns and the tool will prompt the designer to specify the WSDL of the service together with any parameters necessary for the generation of the BPEL version. For Retry, the parameter is the number of retry times. For RecoveryBlock, Active, and Voting, the parameter is a set of WSDLs of all service implementations involved. For Wait, the parameter is the wait-until time. In this example, Retry is chosen and the number of retry times is 5. Then, a fault tolerant version of the service will be generated as a BPEL service for GlassFish ESB v2.2 as shown in Figure 3. The BPEL version invokes the service in a fault tolerant way, implementing the pattern structure we adopt from [2], [7].



Suggest fault tolerance pattern for service

Rank	Fault tolerance pattern		
1	Retry	7.00	Choose
2	Retry+Wait	6.88	Choose
3	Wait	6.75	Choose
4	RB Replica	5.38	Choose
5	Active Replica	4.12	Choose
6	RB NVP	3.75	Choose
7	Voting Replica	3.62	Choose
8	Active NVP	2.50	Choose
9	Voting NVP	2.00	Choose

***Figure 2**. Pattern recommendation by supporting tool.*

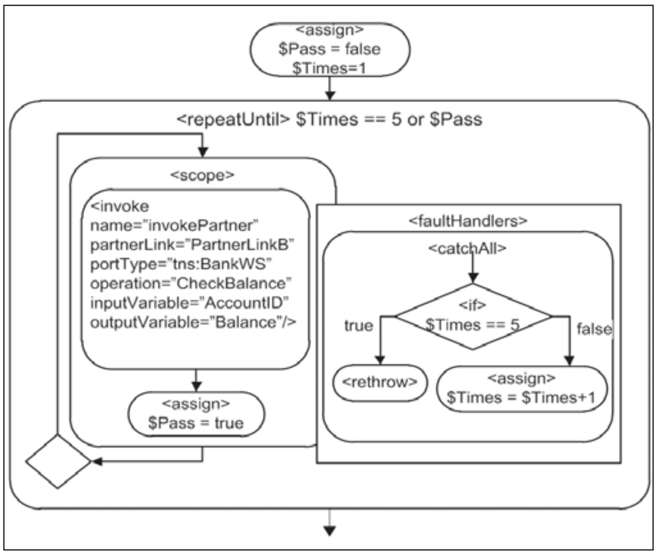


***Figure 3**. BPEL structure for Retry.*

6. Conclusion

In this paper, we propose a model to recommend fault tolerance patterns to services. The recommendation considers service characteristics and condition of service environment. A supporting tool is developed to assist in the recommendation and generation of fault tolerant service versions as BPEL services. As mentioned earlier, it is a lightweight approach which helps to identify fault tolerance patterns that are likely to match service characteristics according to subjective assessment of service designers. At present the recommendation is aimed for a single service. The approach can be extended to accommodate pattern recommendation and generation of fault tolerant composite services. More combinations of patterns can also be supported. In addition, we are in the process of trying the model with services in business organizations for further evaluation.

7. References

[1] M. P. Papazoglou, *Web Services: Principles and Technology*, Pearson Education, Prentice Hall, 2008.

[2] A. Liu, Q. Li, L. Huang, and M. Xiao, “FACTS: A framework for fault–tolerant composition of transactional Web services,” *IEEE Trans. on Services Computing*, Vol.3, No.1, pp. 46-59, 2010.

[3] Z. Zheng and M. R. Lyu, “An adaptive QOS-aware fault tolerance strategy for Web services,” *Empirical Software Engineering*, Vol.15, Iss. 4, pp. 323-345, 2010.

[4] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing,” *IEEE Trans. on Dependable and Secure Computing*, Vol.1, No.1, pp. 11-33, 2004.

[5] G. Dobson, “Using WS-BPEL to implement software fault tolerance for Web services,” *In Procs. of 32nd EUROMICRO Conf. on Software Engineering and Advanced Applications (EUROMICRO-SEAA’06)*, pp. 126-133, 2006.

[6] J. Lau, L. C. Lung, J. D. S. Fraga, and G. S. Veronese, “Designing fault tolerant Web services using BPEL,” *In Procs. of 7th IEEE/ACIS Int. Conf. on Computer and Information Science (ICIS 2008)*, pp. 618-623, 2008.

[7] T. Thaisongsuwan and T. Senivongse, “Applying software fault tolerance patterns to WS-BPEL processes,” *In Procs. of Int. Joint Conf. on Computer Science and Software Engineering (JCSSE2011)*, pp. 269-274, 2011.

[8] R. Hanmer, *Patterns for Fault Tolerant Software*, Wiley Publishing, 2007.

[9] Z. Zheng and M. R. Lyu, “A QoS-aware fault tolerant middleware for dependable service composition,” *In Procs. of IEEE Int. Conf. on Dependable Systems & Networks (DSN 2009)*, pp. 239-249, 2009.

[10] Z. Zheng and M. R. Lyu, “Optimal fault tolerance strategy selection for Web services,” *Int. J. of Web Services Research*, Vol.7, Iss. 4, pp. 21-40, 2010.