A Workflow-Based Failure Recovery in Web Services Composition

Omid Bushehrian¹, Salman Zare², Navid Keihani Rad¹

¹Department of Computer Engineering and IT, Shiraz University of Technology, Shiraz, Iran; ²Department of Computer Engineering and IT, Tehran University, Kish, Iran.

Email: {bushehrian, navid.keihanirad}@sutech.ac.ir, salman.zare@ut.ac.ir

Received November 20th, 2011; revised December 25th, 2011; accepted January 9th, 2012

ABSTRACT

In previous researches in the field of supporting reliability and fault tolerance in web service composition, only low level programming constructs such as exception handling (for example in WSBPEL) were considered. However we believe that the reliability and fault tolerance for composite services must be handled at a higher level of abstraction, i.e. at the workflow level. Therefore a language and technology independent method for fault-tolerant composition of web services is needed. To do this, a fault tolerant workflow is built in which the execution order of the services is determined such that upon a service failure a recovery process with the lowest cost is started. The cost of a service failure includes the cost of failed service and the total costs of roll-backing the previously executed services which are dependent on the failed service. In this article a FSP language is applied to formally specify the workflow.

Keywords: Fault Tolerance; Service Oriented Architecture; Service Composition; Finite State Process

1. Introduction

Nowadays SOA architecture is used as a platform for accessing to data and services in distributed form. Service Composition in this architecture is a way to obtain more complicated services by combining the functionality of individual services [1,2]. The main problem here is the fault tolerance and recovery of failures while executing a composite service [3,4]. In the situation of using a composite service, different faults may occur that mainly causes a service to fail [5]. However a fault-tolerant service composition is the one that ends up the whole transaction in a safe state upon a service failure where the related services are also roll-backed appropriately. Consider a service composition in arranging an itinerary. If the “Flight Reservation” service is failed all other committed services such as “Hotel Reservation” should be roll-backed. Therefore in a fault-tolerant service composition each set of related services may form a transaction for which the atomicity property is a must and by failing one of them, others have to be rolled-back. However for some services the roll-back operation may not be available or only available partially. Therefore failing a service may cause the whole composition ended up in an inconsistent execution due to the violation of atomicity property. To model this limitation, each service within the composition is associated with a “roll-back cost” which its value is an indicator of the amount of impact on the whole composition resulted from requesting the “roll-back” operation for that service. If a service supports the roll-back operation, its associated roll-back cost is zero otherwise some value is considered for this cost. For example if the “Flight Reservation” is not a roll-back supporting service, its rollback cost will be equal to the whole ticket price. In addition to the roll-back costs, for each service within the composition it should be determined on which services this service is dependent. The dependency here is the roll-back dependency and is defined as follows: Service $S_j$ is dependent on service $S_i$ when failure of $S_i$ must start roll-backing of $S_j$ (if already executed).

The main question of our research is how to automatically build a workflow, considering the roll-back dependencies among services, that sequences the execution of the services in a way that upon a service failure, the mean roll-back cost of the service composition becomes minimal. In this context, we call such a workflow a “fault-tolerant” workflow. In brief, we are trying to provide some degree of atomicity property in the execution of services within a service composition. Our solution to this problem should answer the following questions: 1) To what extent the workflow generation can be automated, 2) To what extent the solution can be language independent and 3) Is the solution founded on a rigorous theory (and hence it is easily verifiable)? In the previous works in this field
these important questions are not addressed. There are many previous works that apply fault tolerance techniques at different levels of abstractions to create a fault-tolerant web service composition [6-8]. For example in some languages such as WS-BPEL [1] low level programming constructs like exception handling is provided to support fault tolerance for service composition. However, we believe that the support for reliability and fault tolerance should be considered at the higher level of abstraction: the workflow level. In addition to this shortcoming, the previous studies poorly addressed the robust recovery management once a failure happens.

In this paper these shortcomings are addressed by proposing a method to automatically build a workflow considering the services transactional properties that not only is specified at the high degree of abstraction (and hence language independent) but also supports robust failure recovery.

2. Related Work

In the field of web services reliability and fault tolerance, the previous researches are divided into four main groups [6]:

1) Improvement of the web service reliability in the architectural definition level;
2) Assessment of the system fault tolerance with error injection;
3) Analysis of properties of the second generation web services;
4) Definition of the reliability assessing models of the web service-based systems.

Articles related to the first category generally have used the old reliability techniques for web services which increase the fault tolerance by using redundant services in the architecture; moreover none of them presented a formal method. In the second category error injections for the fault tolerance support have been used [9,10]. Studies in the third category basically include reliability evaluation of the second generation web services, i.e. WS-Reliable Messaging [11], WS Security and WS-Atomic Transaction [11]. In the fourth category, web service based systems generally are created by composing the simpler services according to a work-flow. The reliability of the workflow is evaluated considering the individual services reliability in the workflow. In some researches, the “Markov chain” has been used to model the system behaviour. The probability that the Markov chain comes to a final state from a start state with some limited state depends on the other movements, which means it depends on the reliability of the other states [12]. In [13] to automatically compute the overall QoS of a workflow, a mathematical model and an algorithm (SWR algorithm) are proposed. To support the composition of Web services, they also have presented an ontology-based solution in which a discovery mechanism is applied to find Web services with desired interfaces and operational metrics, and to assist designers in resolving heterogeneity issues among Web services. In [14] to achieve a higher reliability in a composite web service system, it is proposed to decrease the failure rate and increase the repair rate. In this paper a method for calculating the MTTF (Mean Time to Failure) of composite web based on the workflow composition pattern is presented. In [14] a formal verification approach of the workflow-based composite web services is presented. And it has been translated to the BPEL4WS primitives. In [8] they have used EXTRA (Exception handling + TRAnsaction), a hybrid fault-tolerant mechanism which combines exception handling and transaction techniques to improve the reliability of composite services. The first one (exception handling) tries to repair fault and let composite services to continue. The second one (transaction) ensures composite services to terminate in a consistent state when faults are not repairable. They have also presented FACTS framework, which present an integrated environment for specification, verification, and execution of fault tolerant composite services. However, in their work the termination cost of non-cancellable service in a service composition is not taken into account.

3. The Methodology

Our proposed methodology for creating a fault-tolerant workflow for web service composition is illustrated in Figure 1. The first step is to create a Rollback graph considering the service dependencies. Service $S_j$ is dependent on service $S_i$ when failure of $S_i$ must start rollbacking of $S_j$ (if already executed). The set of all dependent services on $S_i$ is depicted by Rollback $[i]$. In the Rollback graph each vertex $S_i$ represents a service and each edge $\langle S_i, S_j \rangle$ represents the existence of a rollback relationship between $S_i$ and $S_j$. The second step is to remove...
the useless edges from the Rollback graph to avoid def-
ing unnecessary dependencies in the subsequent steps. If there is a path from A to C via other services in the Ro-
llback graph and there is also a direct edge (A, C), this
direct edge should be deleted (Figure 2). The third step is to remove the cycles within the rollback graph, first the
cycles are located, and then for each cycle, the order of the services for which the average rollback cost is the
lowest, is determined considering the failure probability and the rollback cost corresponding to each service. At the
end of the third step a Rollback DAG (RBDAG) is ob-
tained. The final step is to use the prerequisite dependen-
cies in the RBDAG to create the fault-tolerant workflow
which is specified in FSP language [15,16].

4. Cycle Elimination
Each cycle within the Rollback graph represents a set of
services with the atomicity property, i.e. once one of them
fails; the previously executed ones should be roll-backed.
To minimize this rollback cost the order of services in
the cycle with the minimum cost is selected and then the
cycle is eliminated to form a DAG. For each permutation
r of the services, the probability of the failure at position
k is denoted by P(r, k) and is computed as follows:

\[ P(r, k) = \prod_{i=1}^{n} (1 - P_i)^{1 - P_k} \]  

(1)

where \( P_i \) is the failure probability of service \( S_i \). The average rollback cost for permutation \( r \) then is calculated as follows:

\[ \text{Avg Cost}(r) = \sum_{i=1}^{n} P(r, k) * \left( \sum_{m=1}^{w} C_m \right) \]  

(2)

where \( C_m \) denotes the rollback cost corresponding to service \( S_m \).

Now it is possible to determine the permutation \( r_{opt} \) for which the value of Avg Cost \( (r_{opt}) \) is minimum. After the best order is specified we can remove that cycle from Rollback graph.

5. Workflow Creation RULES
In order to translate services to an FSP model, the fol-
lowing rules are used:

- **R1**: Corresponding to each service in RBDAG create a
  FSP process.

- **R2**: If service \( w_j \) is prerequisite for service \( w_i \) in the
  RBDAG, create another process called \( lock \). The lock pro-
cess prevents process \( i \) from starting until \( j \) is finished.
  For each service \( w_k \) in RBDAG, If the immediate
  predecessor of \( k \), denoted by \( \text{Pred}(w_k) \), is a member of
  Rollback \( [w_k] \), it must be roll-backed once \( k \) is failed:

- **R3**: To start roll-backing of \( \text{Pred}(w_i) \) only after \( w_k \)
  enters its failed state, a \( lock \) process named \( \text{lock}_k \) is created
  and added to the FSP model. This process prevents \( \text{Pred}(w_i) \)
  from entering to its roll-back state unless \( w_k \) enters its
  failed state.

If there are more than one successor for a given service
\( w_i \) (Figure 3(a)) the corresponding lock process
(step 2) should be created such that all the successors
start theirs executions just after \( w_i \) is finished success-
fully. If there is more than one predecessor for \( w_j \) (Figure
3(b)), the lock process should be created to allow the
execution of \( w_j \) only when all its predecessors are fin-
ished successfully. In some cases, the rollback of \( w_i \) is
dependent on the failure of two or more services together.
This concept is shown in RBDAG like Figure 3(c).

For each service \( (w_i) \) an FSP process with four actions
is defined as shown in Figure 4.

First the service is in state 0 and moves to state 1 by
\textit{start}_k action, and then if it fails, it will move to the final
state 3 and if it succeeds, it goes to state 2. If after the
successful termination of \( w_k \) another service \( w_j \) which
belongs to the same cycle as \( w_k \) fails, then \( w_k \) should be
roll-backed and goes from state 2 to state 3.

However if \( w_k \) is not a member of any cycle in the
Rollback graph or it is, but after removing the cycle it
was placed at the last position in the optimal order, the
rollback never happens after successful termination of \( w_k \)
because according to rule R2, it starts only after all its
dependent services in the cycle are terminated successfully.

![Figure 2. Removing extra edges.](image)

![Figure 3. Prerequisites states.](image)

![Figure 4. Process actions.](image)
This kind of service has actions as shown in Figure 5.

The lock process mentioned in rule R2 is a FSP process with two states as shown in Figure 6.

The Rlock process mentioned in rule R3 is a FSP process with three states as shown in Figure 7. The Rlock goes to state 1 once wk is failed or roll-backed. At state 1, the rollback action is ready to fire assuming that wi is the service which is dependent on wk (i.e. wk has to be roll-backed upon wk unsuccessful termination).

6. Translation Algorithm

Figure 8 shows the Translation Algorithm in which G is a Rollback Directed Acyclic Graph (RBDAG). This algorithm creates a process for each service. If service wi is prerequisite for service wj a lock process is created and added to the FSP model. For each service wk in RBDAG, a subset of Rollback [k] which is predecessors of wk in RBDAG, must be rollback. To do this, a Rlock process is also added to the set of FSP processes.

6.1. AddService() Function

The procedure AddService(), adds a service to the FSP model according to the rule R1, which described in Section 5. It gets a service like wk and checks whether wk is the last service in the order of services in its cycle. If so, then it creates a process without a rollback action:

\[
\text{wk} = (\text{start} \rightarrow (\text{fail} \rightarrow \text{STOP}|\text{succeed} \rightarrow \text{STOP})). 
\]

when wk is not a member of any cycle the created process is also as above.

If wk is not the last service in the order of services in its cycle, it should be roll-backed once its successor services in RBDAG (belonging to a same cycle) fail:

\[
\text{wk} = (\text{start} \rightarrow (\text{fail} \rightarrow \text{STOP}|\text{succeed} \rightarrow \text{rollback} \rightarrow \text{STOP})). 
\]

6.2. AddLock() Function

The function AddLock(), adds a Lock process to the FSP model according to the rule R2, which described in Section 5. The getAllpre (wk) function, gets all the prerequisite services of wk in RBDAG. If the result is only one process wi, the following code is added to the FSP model:

\[
\text{Lock}_k = (\text{unlock} \rightarrow \text{permit} \rightarrow \text{STOP})/\{\text{succeed}/\text{unlock}, \text{start}/\text{permit}\}. 
\]

If there is more than one prerequisite service, the start of the wk depends on the successful termination of all its prerequisite services: (w1 · · · wn):

\[
\text{Lock}_k = (\text{unlock}_1 \rightarrow \cdots \rightarrow \text{unlock}_n \rightarrow \text{permit} \rightarrow \text{STOP})/\{\text{succeed}/\text{unlock}_1, \text{succeed}/\text{unlock}_2, \ldots, \text{succeed}/\text{unlock}_n, \text{start}/\text{permit}\}. 
\]

6.3. AddRLock() Function

The AddRLock() function, adds the Rlockk processes to the FSP model according to the rule R3, described in Section 5. This lock controls the execution of the rollback action of wk. If wk is the last service in the order of services of a cycle or it is not a member of any cycle, no Rlock process is needed because there is no rollback action in the definition of wk process. Otherwise the rollback action of wk should be executed once the service on which wk depends, fails or rollbacks:

\[
\text{RLock}_k = (r\_unlock_k \rightarrow \text{permit}_\text{rollback} \rightarrow \text{STOP})/\{\text{fail}/r\_unlock_k, \text{rollback}/r\_unlock_k, \text{rollback}/\text{permit}_\text{rollback}\}. 
\]

There are situations where the rollback of wk only is required when a set of services (w1 · · · wn) fail together (Figure 3(c)). At these situations, the following Rlock process is added to the FSP model:

\[
\text{RLock}_k = (r\_unlock_k \rightarrow r\_unlock_2 \rightarrow \cdots \rightarrow r\_unlock_n \rightarrow \text{permit}_\text{rollback} \rightarrow \text{STOP})/\{\text{fail}/r\_unlock_1, \text{fail}/r\_unlock_2, \ldots, \text{fail}/r\_unlock_n, \text{rollback}/\text{permit}_\text{rollback}\}. 
\]
A Workflow-Based Failure Recovery in Web Services Composition

7. Case Study: Travel Agency

A travel agency uses the following services to arrange an itinerary:

- \( w_1 \): Granting Visa Service
- \( w_2 \): Flight Reservation Service
- \( w_3 \): Hotel Reservation Service

In order to create a fault-tolerant workflow corresponding to the composition of these services, the methodology described in Section 3 is used. For each service the rollback set is defined as follows:

\[
\text{Rollback}[w_1] = \{w_2, w_3\} \\
\text{Rollback}[w_2] = \{w_1, w_3\} \\
\text{Rollback}[w_3] = \{w_1, w_2\}
\]

Using the above Rollback sets the Rollback graph is created (Figure 9):

The next step is to remove the useless edges from the Rollback graph. After removing these edges the graph shown in Figure 10 is resulted.

In the next step, first the cycles in the above graph are located, and then the average rollback cost corresponding to each order of services in a cycle is calculated. The following failure probabilities and rollback costs for services are assumed:

\[
P_1 = 40\%, \ P_2 = 50\%, \ P_3 = 10\% \\
C_1 = 100, \ C_2 = 80, \ C_3 = 30
\]

There are three services in this example therefore 3! different orders of services are possible. Corresponding to each order a rollback cost is calculated using formula (2) as listed in Table 1.

According to the above table the best order for composition of \( w_1, w_2 \) and \( w_3 \) is: \( w_2, w_1, w_3 \). The resulted RBDAG is shown in Figure 11.

By using the prerequisite dependencies in the resulted RBDAG, the workflow specified in FSP language is created as follows:

7.1. Creating Processes

\[
\begin{align*}
\text{w}_1 &= (\text{start}_1 \rightarrow (\text{fail}_1 \rightarrow \text{STOP} | \text{succeed}_1 \rightarrow \text{rollback}_1 \rightarrow \text{STOP})). \\
\text{w}_3 &= (\text{start}_3 \rightarrow (\text{fail}_3 \rightarrow \text{STOP} | \text{succeed}_3 \rightarrow \text{STOP})). \\
\text{w}_2 &= (\text{start}_2 \rightarrow (\text{fail}_2 \rightarrow \text{STOP} | \text{succeed}_2 \rightarrow \text{rollback}_2 \rightarrow \text{STOP})).
\end{align*}
\]

7.2. Creating Locks

\[
\text{Lock}_1 = (\text{unlock}_2 \rightarrow \text{permit}_1 \rightarrow \text{STOP}) /\{\text{succeed}_2/\text{unlock}_2, \text{start}_1/\text{permit}_1\}. \\
\text{Lock}_3 = (\text{unlock}_3 \rightarrow \text{permit}_3 \rightarrow \text{STOP}) /\{\text{succeed}_3/\text{unlock}_3, \text{start}_3/\text{permit}_3\}.
\]

7.3. Create RLocks

\[
\text{RLock}_1 = (\text{r_unlock}_1 \rightarrow \text{permit}_\text{rollback}_1 \rightarrow \text{STOP}) /\{\text{fail}_2/\text{r_unlock}_1, \text{rollback}_2/\text{r_unlock}_1, \text{rollback}_1/\text{permit}_\text{rollback}_1\}. \\
\text{RLock}_3 = (\text{r_unlock}_3 \rightarrow \text{permit}_\text{rollback}_3 \rightarrow \text{STOP}) /\{\text{fail}_1/\text{r_unlock}_3, \text{rollback}_1/\text{r_unlock}_3, \text{rollback}_3/\text{permit}_\text{rollback}_3\}.
\]

Figure 12 shows the final workflow in LTSA, which is achieved from composition of processes in our case study.

8. Discussion

In Table 2, three different methods in the field of supporting reliability and fault tolerance in web service composition are compared with our work using six factors.
Table 2. Comparing different methods for fault-tolerant web service composition.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Language (Technology) Dependency</td>
<td>Independent</td>
<td>Independent</td>
<td>Independent</td>
<td>Workflow-based:</td>
</tr>
<tr>
<td>Automated</td>
<td>No</td>
<td>Automatically Generates</td>
<td>Yes</td>
<td>Independent</td>
</tr>
<tr>
<td>Recovery Support</td>
<td>No, Uses Redundant service without any recovery support</td>
<td>Partial: Terminates in a consistent state</td>
<td>No</td>
<td>Yes: Terminates in a consistent state with minimum cost</td>
</tr>
<tr>
<td>Fault Tolerance Technique</td>
<td>Redundancy</td>
<td>Exception Handling</td>
<td>Pattern based design</td>
<td>Recovery Process with the Lowest Rollback Cost</td>
</tr>
<tr>
<td>Theoretical Foundation</td>
<td>FSP Language</td>
<td>No</td>
<td>No</td>
<td>FSP Language</td>
</tr>
<tr>
<td>Transactional Support</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The first factor is the language or technology dependency. All four methods including ours are language independent. The second factor is the support for an automated method to generate the fault-tolerance mechanisms from the high level specifications. Our proposed method supports automatic generation of reliable workflow from rollback dependencies. The third factor is the recovery support, which means if a service fails how the composition continues in order to terminate in a consistent state. The first method uses redundant services without any recovery support. The second method tries to terminate the composition in a consistent state. In contrast, in our work a rollback method for failed services with minimum cost is supported. The forth factor is the fault tolerance technique applied by each method. The fifth factor is the existence of a theoretical foundation to support the fault tolerance or recovery. The first method uses the FSP language to be able to evaluate the correctness of its model. We also have used the FSP language as a formal language as a workflow specification language for assessing the correctness of the resulted workflow. The FSP language chosen in this paper, allowed us to present an algorithm for automatically creating a fault tolerant workflow from the service dependencies, due to its rigorous theoretical foundation. The last factor is the transactional support which indicates whether a method considers the concept of atomic transactions in a composition or not. We have presented a method to automatically build a fault-tolerant workflow considering the service transactional properties (Each cycle in the Rollback graph is composed of a set of services with the atomicity requirement).

9. Verification Approach

The presented algorithm in the previous section for translating RBDAG to FSP verifies that it is possible to fully automate the workflow generation task (our first research question stated in the introduction part). Since FSP is a technology independent workflow specification method and also it is based fully on a rigorous theoretical background, the second and third research questions (mentioned in the introduction section) are also answered.

To prove the correctness of our translation algorithm formally, it should be verified that the generated work-
flow always terminates in a consistent state, i.e. that upon the failure of service $S_i$, all its rollback-dependent services like $S_j$ in the RBDAG are terminated in their roll-back state (state 3 in Figure 4). To show this, from step III of the translation algorithm, the following process is created for $S_i$:

\[
\text{R} \text{Lock} = (r \_unlock \rightarrow \text{permit} \_\text{rollback} \rightarrow \text{STOP})
\]

\[
\text{/}(\text{fail} / r \_unlock, \text{rollback} / r \_unlock, \text{rollback} / \text{permit} \_\text{rollback}).
\]

The synchronization part specifies that upon a failure (fail; action) or roll-back (rollback; action) of $S_i$, the permit_rollback action is fired which is synchronized with the rollback action of $S_j$ and it means that eventually after $S_i$ entered to state 2 (succeeded state), it cannot stay in this state due to that fact that rollback action is ready to fire.

10. Conclusions and Future Works

The fault tolerance in the web service composition usually is supported by the exception handling constructs at the language level. However, using the workflow technology for composing web services with fault tolerance consideration provides a language independent solution. The FSP language chosen in this paper, allowed us to present an algorithm for automatically creating a fault tolerant workflow from the service dependencies, due to its rigorous theoretical foundation. The resulted workflow executes orrollbacks the services in the composition in an order such that the minimum rollback cost is incurred upon the service failures in the composition.

REFERENCES


