Web Services Workflow Reliability Estimation Through Reliability Patterns

Luigi Coppolino, Luigi Romano
Dipartimento per le Tecnologie
Università degli Studi di Napoli Parthenope
via Medina 40, 80133 Napoli, Italy
luigi.coppolino@uniparthenope.it
luigi.romano@uniparthenope.it

Nicola Mazzocca, Sergio Salvi
Dipartimento di Informatica e Sistemistica
Università degli Studi di Napoli Federico II
via Claudio 21, 80125 Napoli, Italy
nicola.mazzocca@unina.it,
sersalvi@yahoo.it

Abstract

Web Services are one of the most promising technologies for the development of component based systems. Nevertheless, there are still several open issues which limit their potential benefits and impair their adoption. Among these, lack of effective techniques for systems dependability forecasting is a key problem. The approaches proposed in the literature are mainly borrowed from traditional software reliability theory, and they typically do not take into account specific characteristics of the Web Services technology. In this paper, we propose a novel approach to reliability evaluation of a web services based system. The approach exploits the concept of reliability patterns to derive an aggregate reliability function for a wide class of workflow processes. A new technique is also used to determine the reliability of individual services.

Keywords: Web Services, Workflow Systems, Reliability, Reliability Patterns.

1 Web Services dependability

Most of the papers focusing on Web Services (WS) dependability fall in one of four categories, namely: i) the definition of architectures allowing dependability improvement, ii) the assessment of WS fault tolerance by fault injection, iii) analysis of second generation WS specifications, and iv) the definition of models for dependability evaluation of Web Services based systems.

Papers belonging to i), basically apply traditional software reliability techniques (e.g. N-Version Programming) to web services. They tolerate host failures by exploiting the redundancy available in the architecture. Some works, such as [1] and [2], extend to WS techniques developed for CORBA. Other works, like [3] and [4], propose new architectures based on the N-Version Programming approach. None of them provides formal evidence showing that the adoption of such techniques actually improves the reliability of the system (it is in fact known that the adoption of replication may either improve or reduce the reliability of the overall architecture, depending on the reliability of individual components [10]).

As to ii), Fault injection is a widely adopted technique for assessing the fault coverage of explicit and implicit fault tolerance mechanisms, and to study system failure modes and fault propagation models [15]. It has been adopted in several stages of the system development process, and at the various logical and physical levels of a system. Fault injection has also been applied to web services. A major fraction of the research in this field is due to Looker and Xu [16][17][18], and mainly focuses on the assessment of the fault tolerance properties of the SOAP protocol.

Studies in category iii) basically consist of dependability evaluation of second generation WS specifications (e.g. WS-ReliableMessaging, WS-Security, WS-AtomicTransaction).

As to iv), Web Services based systems are typically composed by orchestrating a number of simpler services (generally Web Services themselves) in a common workflow. In such a case it is widely accepted that the reliability function of the workflow must be derived based on the reliability functions of individual tasks in the workflow. Some authors [5],
[6], [7], use commonly adopted frameworks for reliability evaluation, such as Markov chains. The basic idea here is to model the behaviour of the system as a set of states, each representing one or more tasks executing concurrently. One of them is identified as the initial state and represents the invocation of a service by means of a client. Another state, representing the correct termination of the service is labelled as the final state. The reliability is then evaluated as the probability that the Markov chain evolves from the starting state to the final one in a finite number of transitions. Such a probability is then related to the probability of moving from a state to the next one, probability which in turn depends on the reliability of the activities performed in the previous state. An alternative way to approach the problem has been proposed in [8] and in [19]. In [8] authors propose both a way to evaluate the reliability of a single service and a way to evaluate the reliability of a complex workflow composed by a number of services. The workflow is supposed to be described by means of ACDATE/Scenario, a language for the description of activity workflows. The idea is based on the assumption that individual services fail independently of each other. Under this assumption, the authors obtain the reliability expression of the basic patterns defined in ACDATE. The most interesting part of this approach is the attempt of moving the level of abstraction up to the business process level, and – ultimately – closer to the designer’s point of view. However, the approach taken to the evaluation of the reliability of individual nodes is quite naïve, since: i) the definition used differs from the classical one, based on the concept of random variable, and ii) a fault is said to occur if one out of N-versions of a service deviates from the other ones (which is neither general, nor universally acceptable). Nevertheless, this paper represents one of the rare attempts to draw up a model for the reliability of an elementary service.

A more mature work is [19], where the author proposes a set of workflow patterns with related reliability expressions, and presents METEOR, a workflow engine which allows combining them so to build a more sophisticated workflow, and deriving its aggregate reliability expression. Since only the set of patterns defined in [9] can be combined using METEOR, this is not applicable to a generic workflow.

Finally, it is also worth of note that - to the best of our knowledge - no study exists, which tries to evaluate the reliability of a single service while explicitly taking into account the presence of infrastructure frameworks which provide non-functional services (e.g. reliable messaging, security, etc.).

In this work, we propose an approach for reliability evaluation of workflow systems, which is more general than the one presented in [19]. Our starting point is the results presented in [9], where a set of 20 basic workflow patterns is identified, which is suitable to describe virtually any control flow. Starting from such patterns, we derive a set of new patterns – which we will refer to as reliability patterns - meaningful in the context of dependability. For each reliability pattern, we derive a rule which gives the reliability formula of the pattern. Since our reliability patterns are extracted from the workflow patterns identified in [9], our formulas can be applied to a wider class of workflows (as compared to the one studied in [19]).

Another contribution of the paper is a novel approach to the evaluation of the reliability of a single service, which explicitly takes into account the presence of infrastructure frameworks providing non-functional requirements.

2 Workflow patterns

The starting point for our work is the results presented in [9], where a set of 20 basic patterns is identified, which can be combined so to generate virtually any control flow.

The twenty workflow patterns fall into six main categories, namely:

- Basic Control Flow Patterns
- Advanced Branching and Synchronization Patterns
- Structural Patterns
- Patterns involving Multiple Instances
- State-based Patterns
- Cancellation Patterns

While analyzing the patterns provided in [9], some observations are due:

i) Since we are only interested in the reliability of the workflow from an architectural point of view, not all the patterns are relevant for our purposes. As an example, the pattern Cancel Case relates to the workflow management system and is therefore not relevant for the composition process.

ii) From a reliability point of view some patterns are equivalent. As an example the Multiple Instances pattern provides the same reliability of a Parallel Split
or of a Multiple Choice depending on the necessity of completing or not all the activated instances.

iii) Combinations of patterns are often needed - in order to address reliability - instead of individual patterns. This is the case of the Parallel Split, for which deriving reliability requires knowing if the following task is a Synchronization or a Multi-Merge

iv) Finally, not all pattern combinations are meaningful from a controlflow point of view (just think of a XOR-Split followed by an AND-Join, where one and only one from a set of tasks is activated, while the termination of all of them is needed).

In the next section, we first describe an algorithm which derives the aggregate reliability function through a workflow graph reduction, then we discuss the derived reliability patterns and their reliability formulas, finally we present an example showing how the algorithm works.

3 Reliability Workflow Patterns

3.1 Reduction algorithm

When dealing with workflow, we are assuming that web services are composed in an orchestration. We assume a workflow described as

\[ W = (t, a, fr, fp, fc) \]

Where
- \( t \) is a set of tasks (each represented by a circle);
- \( a \) is a set of transitions (each represented by an arrow)
- \( fr \) is a function which associates to every task \( t_i \) in \( t \) its reliability function
- \( fp \) is a function which associates to every transition \( a_{ij} \) (connecting the task \( i \) to the task \( j \)) a probability \( p_{ij} \), representing the probability that once the task \( t_i \) terminates the task \( t_j \) is activated. In other words \( p_{ij} \) represents the probability of activation of the transition \( a_{ij} \).
- \( fc \) is a function which for every task \( t_i \) in \( t \) associates a value \( c_i \) in \([0,1]\) representing the probability that a failure of the task \( t_i \) does not lead to a failure of the workflow. Hence \( c_i \) represents a coverage factor, and can be expressed as:

\[ c_i = \sum_{g \in G} \varphi(g) P(g) \]

Where:
- \( g \) is a failure mode for the task \( i \)
- \( G \) is the fault dictionary for the task \( i \)
- \( \varphi(g) = 1 \) if the failure \( g \) can be tolerated, 0 otherwise
- \( P(g) \) is the occurrence probability of the failure \( g \)

This implies that the reliability for the single task is increased by a factor representing the probability that the component will fail without leading to a workflow failure, that is:

\[ R_i = R_i' + (1 - R_i') c_i \]  

(1)

Where \( R_i' \) represents the reliability of the task \( t_i \) and \( R_i \) represents the reliability of the task \( t_i \) as perceived by the workflow engine. The latter equals the former when \( c \) is zero, i.e. the workflow cannot tolerate a service fault. If \( c \) equals 1 the formula returns 1 meaning that the component is optional from a reliability point of view. In the next two sub-divisions we will always use the term reliability with reference to the meaning it assumes in (1).

A start task and an end task must be identified into the set of the tasks. The start task does not have any incoming transition and represents the invocation of the orchestrated service by an external client. The end task does not have any outgoing transition and represents the end of the orchestration.

Once the graph representing the web services orchestration is defined, the reduction algorithm is performed by going backward through the graph (from the end task to the start one) and each time an individual reliability pattern is found its component tasks are collapsed in a single task whose reliability is defined by the pattern reliability formula. The process is than iterated until the whole workflow is collapsed in a single task whose reliability depends on the reliability of the individual tasks, the probabilities \( p_{ij} \) and the coverage factors \( c_i \).

3.2 Reliability Patterns

Sequence

\[ A \rightarrow B \]
This reliability pattern directly matches the homonymous pattern presented in [9], where it is defined as follows:

"An activity in a workflow process is enabled after the completion of another activity in the same process."

The pattern represents a series configuration where the success probability is obtained as the probability that both the component tasks succeed, hence:

\[ R = R_A \times R_B \]

**Synchronizing Parallel**

![Diagram of Synchronizing Parallel]

This reliability pattern is obtained by aggregating the following combinations of patterns:

- **AND-Split/Interleaved Parallel Routing** + **AND-Join/Synchronizing Join**
- **OR-Split** + **Synchronizing Join**

It represents a point in the workflow where after the execution of a task A a number of tasks are executed in parallel, each with a given activation probability \( p_i \) for \( i = 1, \ldots, n \). The orchestration engine waits for all the activated tasks to complete and then executes task B.

It is worth of noting that:

i) When all the \( p_i = 1 \) the task A becomes an AND-Split

ii) If exists \( i \) such that \( p_i \neq 1 \) the task A becomes an OR-Split

iii) From the controlflow point of view the task B starts only if all the activated tasks \( t_i \) have terminated, anyway from a reliability point of view it doesn’t mean that all the activated tasks have to succeed. Hence, we associate an integer number \( k \leq n \) to the pattern, which represents the minimum number of tasks that must accomplish their execution successfully in order to activate task B. By doing so the pattern can also model a \( k\text{-out-of-}n \) fault tolerant configuration.

To obtain the probability that the whole pattern succeeds it is necessary to consider all the possible combinations in which at least \( k \) tasks, over the activated ones, succeed. This is then multiplied by the reliability of tasks A and B, obtaining:

\[
R = R_A R_B \sum_{i=0}^{2} \sum_{j=0}^{n} u(\sum_{j=0}^{n} d(i_j - 1) - k) \prod_{j=1}^{n} (d(i_j - 1) R_j p_j + d(i_j - 2)(1 - p_j))
\]

where:

\[ u(n) = 1 \text{ if } n \geq 0, 0 \text{ otherwise} \]

\[ d(n) = 1 \text{ if } n = 0, 0 \text{ otherwise} \]

The formula sums all the probabilistic states for which at least \( k \) tasks over the activated ones execute successfully. The summation over \( i_1, \ldots, i_n \) corresponds to the generation of a sequence of 3 values for which 0 represents that the task has been activated but has failed, 1 represents the success of a task and 2 represents the fact that a task hasn’t been activated. In each term of the summation, based on the value assumed by \( i_j \), the \( d( ) \) function allows to select only one of the three terms in the product.

A probabilistic state must be considered only if the number of succeeding tasks is greater or equal than \( k \), i.e.

\[
\sum_{j=1}^{n} d(i_j - 1) > k \iff \sum_{j=1}^{n} d(i_j - 1) - k > 0 \quad (2)
\]

The states non satisfying the (2) are set to zero by using the \( u( ) \) function.

**Multi-merge Parallel**

![Diagram of Multi-merge Parallel]

This reliability pattern is obtained by aggregating the following combinations of patterns:

- **AND-Split/Interleaved Parallel Routing** + **OR-Split** + **Multi-Merge Join**

In [9] the multi merge join is defined as follows:

"A point in a workflow process where two or more branches reconverge without synchronization. If more than one branch gets activated, possibly
concurrently, the activity following the merge is started for every activation of every incoming branch.”

Since in this case task B is re-activated after every activated task \( t_i \) accomplishes, the reliability of B is included in the product, thus:

\[
R = R_A \sum_{i=0}^{\infty} \sum_{j=0}^{\infty} \prod_{k=1}^{n} \left( \delta(i_j - 1) R_j p_j R_B \right) + \delta(i_j)(1-R_j)p_j + \delta(i_j - 2)(1-p_j)
\]

This implies the necessity to scan backward the graph so that when this pattern is computed all the tasks following the pattern in the workflow have already been collapsed in B.

**XOR Parallel**

This reliability pattern matches the following aggregated patterns:

- Exclusive Choice/Deferred Choice + Simple Merge/Synchronizing Merge/Multiple Merge.

Let’s recall the definition of the Exclusive Choice:

“A point in the workflow process where, based on a decision or workflow control data, one of several branches is chosen.”

The pattern is successful if the activated task is successful, where every task has a probability \( p_A \) to be activated. Therefore the reliability of the pattern can be obtained as the summation of the reliability of individual tasks, each weighted by its probability \( p_A \), multiplied by the reliability of both the task A and B

\[
R(t) = R_A R_B \sum_{i=1}^{n} p_A i R_i \quad \text{where} \quad \sum_{i} p_A i = 1
\]

**Loop**

This reliability pattern strictly matches the Arbitrary Cycles Pattern, which is defined as follows:

“A point in a workflow process where one or more activities can be done repeatedly.”

By considering the set of tasks in the loop as a single task whose reliability is R and identifying by \( R_i \) the reliability of the loop, we have:

\[
R_i(t) = \frac{R(t)(1-p)}{1-pR(t)}
\]

**Discriminator Parallel**

This reliability pattern matches the aggregated patterns:

- Parallel Split + Discriminator.

In [9] the Discriminator pattern is defined as follows:

“The discriminator is a point in a workflow process that waits for one of the incoming branches to complete before activating the subsequent activity. From that moment on it waits for all remaining branches to complete and “ignores” them. Once all incoming branches have been triggered, it resets itself so that it can be triggered again.”

Hence the probability that the pattern executes successfully is the probability that the task \( t_i \) is the first one to complete and it executes successfully, for \( i = 1, \ldots, n \), thus:

\[
R = R_A R_B \sum_{i=1}^{n} p_A i R_i \quad \text{where} \quad \sum_{i} p_A i = 1
\]
3.3 Reliability pattern usage

This section provides a simple example showing how the reduction algorithm applies the rules retrieved in the last section in order to obtain the workflow reliability function.

Let’s consider the workflow graph represented in figure 1. The reduction algorithm executes the following steps:

Step 1: scanning the graph starting from the end, the first reliability pattern that is recognised is the one made up of the tasks d, e₁, e₂, f, which match a XOR-Parallel pattern. By applying the pattern reduction rule the four tasks collapse in a single task, T₁, whose reliability is given by the specific formula. Continuing going backward through the graph the next pattern to be identified is a Synchronizing Parallel made up of the tasks a, b₁, b₂, b₃, c which in turn collapse in a single task, say T₂. When evaluation its related formula, since \( p_{ab1} = p_{ab2} = p_{ab3} = 1 \) and \( K=3 \), in the summation we are left with just the term where \( i \) equals 1.

At this point the first iteration ends since it is not possible to identify any other reliability pattern.

Step 2: during the second iteration the algorithm, starting again from the right side of the graph, identifies a Loop pattern made up of the task T₁ obtained at the previous step. The algorithm substitutes the Loop with a single task, T₃, whose reliability is given by the apposite rule.

Step 3: during the third iteration the Sequence g-T₃ is resolved by obtaining a new task T₄. Again a Sequence, T₂-T₄, is identified and reduced. At this point a single task remains whose reliability is equal to the one of the whole orchestration.

4 Assumptions and limits of the model

The main hypothesis underling the analysis proposed in the previous section, and of course the proposed approach, is the independence of events \( A_i = \{ 'time to first failure of activity i' < t \} \) and \( A_j = \{ 'time to first failure of activity j' < t \} \), for each \( i \neq j \).

Let us consider for example the case of an orchestration of two services offered from the same provider and thus hosted on the same server. In such a case during the downtime of the server both the services will fail disagreeing with the requirements of independence. This also happen when two ore more services in a workflow depends from a common service.
A further simplification in this approach lies in the absence from the model of the communication channel reliability. Actually the communication channel may itself introduce faults, as an example by dropping packets, or modifying them or just delaying their delivery beyond timeout expiration. Anyway such a kind of behaviour can be embedded into the model of the single service. We will discuss about that in next sections.

Finally it is worth of noting that the obtained model provides the reliability of the services orchestration without considering the reliability of the service that performs the orchestration. As an example let’s consider a service which by means of an orchestration engine (e.g. BPEL) coordinate the invocation of other services by following a predefined workflow. In this case the reliability of the orchestration service, of the server hosting such a service and of the orchestration engine, should be modelled and in case of hypothesis of independence should be multiplied to the workflow reliability.

5 Reliability of a WS Component

To obtain the reliability of a Web Services based system by adopting the approach previously exposed, or any other approach available in the literature, an estimation of the reliability, $R'(t)$ (see (1)), of every single service is needed. Nevertheless there is not much work concerning the reliability evaluation of an individual Web Service. Actually, at the moment it is possible to make use of the literature related to generic software reliability modelling [11] [12] [13]. These models use to treat a software component as a monolithic element, whose reliability follows a certain statistical behaviour, which in turn depends on a number of parameters. To obtain the values of such parameters in a real case a collection of field failure data is required. However, it is not realistic to think of a Web Service as a single monolithic component since, more often, the single service is used together with a number of additional frameworks implementing non functional specifications (e.g. WS-Security, WS-ReliableMessaging, WS-Transaction, etc. (figure 2)).

The adoption of such frameworks can have a severe impact on the QoS of a Web Service, sometimes reverting previsions made without considering them. As an example figure 3 shows the performance measured running a Web Service with just the basic WS stack protocols and the one measured when the same Web Service is used with a framework implementing WS-Security. Not only the presence of those frameworks causes substantial modifications in performance but the presence of two different implementations reverts the results obtained while considering just the SOAP engines. This simple example shows how the presence of any of such specifications and its specific implementation becomes important when dealing with QoS attributes.

If the service and its related frameworks are all modelled as a whole it is necessary to re-evaluate the model parameters by collecting new failure data at each time a change in the set of specifications adopted is made. On the other hand we propose to consider the every individual component itself as a multi-layer and multi component system, where each layer is a different framework, made up of different components.

Figure 2 - Web Services extended stack

Following such an approach the reliability of a service can be evaluated as proposed in [13]. Given the reliability of the generic component $i$ in the framework $j$, $\lambda_{i,j}$, and a service utilization matrix $U_{i,j+1}$, where the element $u_{j,k}$ equals 1 if, during the execution, component $j$ of the framework $i$ uses the services of the component $k$ at framework $i+1$, it is possible to derive the failure rate for the service as:

$$\lambda = \sum_{j=1}^{c_i} \pi_{i,j} \omega_{i,j}$$

where $\omega_{i,j}$ is the aggregated failure rate of the component $j$ of the first framework (which is the class that provides the service itself) and depends on the utilization matrix and the failure rates of the single components, and $\pi_{i,j}$ is the mean time to failure for
the component \( j \) of the first framework (more details about the theory of multi-interpreter system can be found in [13]).

Figure 2 reports the main layers for a Web Service. In such a stack the components specific of the web services are those lying over the SOAP engine. Anyway it is worth of noting that the application server cannot be omitted since, even if it is not mandatory, in actual implementations SOAP messages are always routed on top of HTTP and the SOAP engines are always hosted into an application server. Hence in our model are considered as layers of the services those of them lying upon the application server while the type and number of such layers will change case-by-case.

The greatest advantage of this approach is the possibility of reusing the evaluation made for the single layer (which generally involves an expensive collection of failures data).

![Figure 2: Main layers for a Web Service](image)

**Figure 2 - Main layers for a Web Service**

The greatest advantage of this approach is the possibility of reusing the evaluation made for the single layer (which generally involves an expensive collection of failures data).

![Figure 3: Performance reversion: Axis 1.2 and WSS4J against Web Logic performance](image)

**Figure 3 - Performance reversion: Axis 1.2 and WSS4J against Web Logic performance**

### 6 Conclusions and future work

In this paper we have identified a set of reliability patterns which can be applied in an iterative reduction algorithm to evaluate the aggregate reliability function for a wide class of workflows. For each pattern, a reduction rule has been proposed and the reduction process evolves by substituting to every identified pattern a single task whose reliability comes from the applied rule. Since the resulting aggregated function depends on the reliability of the orchestrated services, we have also discussed a novel approach to the evaluation of the reliability of individual components. In the proposed approach, the service itself is considered as a multi-layer and a multi-component system. This approach allows reusing the reliability evaluations made for the frameworks providing non-functional features to the service.

At the moment, we are working to integrate our reduction process into an orchestration engine so to provide a useful plug-in, which can be used to evaluate the reliability of a planned workflow, as well as to compare the reliability of alternative solutions.

### 7 References


on Software Engineering and Applications, Cambridge, MA, November 2004, pp. 144-149.


[19] Antonio J.S. Cardoso, “Quality of Service and Semantic Composition of Web Services”, PhD thesis, Department of Computer Science, University of Georgia, Athens, GA (USA), 2002