Dependable Composite Web Services with Components Upgraded Online

Anatoliy Gorbenko[1], Vyacheslav Kharchenko[1], Peter Popov[2], Alexander Romanovsky[3]

[1] Department of Computer Systems and Networks, National Aerospace University, Kharkiv, Ukraine
A.Gorbenko@csac.khai.edu, V.Kharchenko@khai.edu
[2] Centre for Software Reliability, City University, London, UK
ptp@csr.city.ac.uk
[3] School of Computing Science, University of Newcastle, Newcastle upon Tyne, UK
alexander.romanovsky@ncl.ac.uk

Abstract. Achieving high dependability of Web Services (WSs) dynamically composed from component WSs is an open problem. One of the main difficulties here is due to the fact that the component WSs can and will be upgraded online, which will affect the dependability of the composite WS. The paper introduces the problem of component WS upgrade and proposes solutions for dependable upgrading in which natural redundancy, formed by the latest and the previous releases of a WS being kept operational, is used. The paper describes how 'confidence in correctness' can be systematically used as a measure of dependability of both the component and the composite WSs. We discuss architectures for a composite WS in which the upgrade of the component WS is managed by switching the composite WS from using the old release of the component WS to using its newer release only when the confidence is high enough, so that the composite service dependability will not deteriorate as a result of the switch. The effectiveness of the proposed solutions is assessed by simulation. We discuss the implications of the proposed architectures, including ways of 'publishing' the confidence in WSs, in the context of relevant standard technologies, such as WSDL, UDDI and SOAP.

1 Introduction

The Web Service architecture [1] is rapidly becoming the de facto standard technology for achieving interoperability between different software applications running on a variety of platforms. This architecture supports development and deployment of open systems in which component discovery and system integration can be postponed until the systems are executed. Individual components (i.e. Web Services – WSs) advertise their services via a registry (typically developed using the UDDI standard[1]) in which their descriptions, given in a standard XML-based

[1] http://www.uddi.org/

language called Web Service Definition Language (WSDL[2]), can be looked up. After a WS capable of delivering the required service has been found it can be used or even dynamically integrated into a composite WS.

The WS architecture is in effect a further step in the evolution of the well-known component-based system development with off-the-shelf (OTS) components. The main advances enabling this architecture have been made by the standardisation of the integration process (a set of interrelated standards such as SOAP[3], WSDL, UDDI[4], etc.). WSs are the OTS components for which a standard way of advertising their functionality has been widely adopted.

The problem of dealing with online system upgrades is well known and a number of solutions have been proposed (see, for example [2]). The main reasons for upgrading the systems are improving/adding functionality or correction of bugs. The difficulties in dealing with upgrades of COTS components in a dependable way are well recognised and a number of solutions have been proposed. The WS architecture poses a new set of problems mainly caused by its openness and by the fact that the component WSs are executed in different management domains and are outside of the control of the composite WS. Moreover, switching such systems off or inflicting any serious interruptions in the service they provide is not acceptable, so all upgrades have to be dealt with seamlessly and online. One of the motivations for our work is that ensuring and assessing dependability of complex WSs is complicated when any component can be replaced online by a new one with unknown dependability characteristics.

There is clearly a need to develop solutions making use of natural redundancy which exists in such systems and guaranteeing that the overall dependability of the composite system is improving rather than deteriorating. Note that the idea of using the old and the new releases of a program side by side to improve its dependability is far from new: it was first mentioned by B. Randell in his work on recovery blocks in which the earlier releases of the primary alternate are seen as a source of secondary alternates [3].

The rest of the paper is organised as follows. Section 2 gives an overview of the Web Service dependability and shows how it can be assessed using measures such as “confidence in WS correctness”. In section 3 we introduce the problem of a component WS upgrade. Section 4 discusses how keeping several releases of a component WS available can affect the composite WS. In section 5 we provide a brief description of the Bayesian inference and show how it can be applied in the context of WS for assessing the confidence in their correctness. Some simulation results are also presented to illustrate the effectiveness of the proposed architectural solutions. Finally, in section 6 we briefly outline the on-going work on building a test harness for managed WS upgrade together with several ways of ‘publishing’ the confidence in a WS, compatible with relevant standards, such as WSDL, UDDI and SOAP.

[2] http://www.w3.org/TR/wsdl

[3] http://www.w3.org/TR/soap12-part0/

[4] http://www.oasis-open.org/committees/uddi-spec/

2 Web Services Dependability

The WS architecture is now extensively used in developing various critical applications with high dependability requirements, such as banking, auctions, Internet shopping, hotel/car/flight/train reservation and booking, e-business, e-science, business account management, which in turn demands adequate mechanisms for dependability assurance and dependability assessment in the new context of WSs (see [4], [5]). In [1] the idea of 'Service Management' is advocated as a way of providing the users of a WS with information about its dependability. Such a service is achieved via a set of capabilities, such as monitoring, controlling, and reporting on the use of the deployed WS.

Dependability of a computing system is the ability to deliver service that can be justifiably trusted [6]. Dependability of the Web Services is a system property that integrates several attributes, the most important of which are availability (including responsiveness), reliability (correctness), and security. For many applications it would be desirable if the service requester (consumer) could quantify these attributes by either assessing them independently or relying for the assessment on a third party, e.g. a trusted independent dependability broker or even the WS provider.

We recognise that security is a very important dependability attribute, especially in the context of WSs. However, since the techniques for security assessment are still at an embryonic stage, security is not addressed in this paper. Whether the ideas presented here, e.g. confidence in security, are applicable, is to be seen when security assessment techniques mature.

2.1 Web Services Failures

A system failure is an event that occurs when the delivered service deviates from system specification.

A number of approaches has been used to analyse failures, their modes, effects and causes in the context of Web Services [7], system software [8] and a computer system as a whole [6], [9]. In this paper we focus on the following failure modes.

Transient failure – a failure triggered by transient conditions which can be tolerated by using generic recovery techniques such as rollback and retry even if the same code is used.

Non-transient failure – a deterministic failure. To tolerate such failure the diverse redundancy should be used. Such redundancy naturally exists during WS upgrading when the old (one or more) and new releases of the same WS are available.

Evident failure – a failure that needs *no special means* to be detected. It may be, for example, an exception, denial of service or absence of response during a predefined period of time, which will be detected by a general-purpose mechanism such as timeout.

Non-evident failure – a failure that can be detected only by using the existing redundancy at the application level (e.g. in the form of diversity). It is clear, that the non-evident failures can have more dramatic consequence than the evident failures.

This understanding of possible failure modes will be taken into account while building dependable Web Services and will affect the choice of the error detection

mechanisms and fault-tolerance techniques employing several WS releases available online.

2.2 Confidence in the Web Services

WSs, as any other complex software may contain faults which may manifest themselves in operation. In many cases the consumers of the WSs may benefit from knowing how confident they can be in the availability, responsiveness and correctness of the information processing provided by the WSs. This issue may seem new in the context of WSs but is not new for some well-established domains with high dependability needs such as safety critical applications for which it is not unusual to state dependability requirements in probabilistic terms, e.g. as *probability of failure of software on demand* [10].

This fits nicely in the context of WSs, which can be seen as successive invocations of the operations published by a WS. It may be very difficult (or impossible) to guarantee that software behind a WS interface is flawless, but the confidence of the consumers will, no doubt, be affected by knowing for how long the service has been in operation and by how many failures have been observed. Informally, we will be much more confident in the results we get from a piece of software after we have seen it in operation for a long period of time without a failure than if we have not seen it in operation at all. How long software has been used is no guarantee that we will have high confidence in its dependability. Clearly, if we have seen it fail many times in the past we will take with doubt the next result that we get from this piece of software.

Building confidence measures to assess the correctness, the availability and the responsiveness can be formalised. Bayesian inference [11] is a mathematically sound way of expressing the confidence combining the knowledge about how good or poor the service is prior to deployment with the empirical evidence which becomes available after deployment. A priori knowledge can be gained by the WS provider using standard techniques for reliability assessment, e.g. the quality of the development process or other techniques such as those described in [12].

The confidence in the dependability of the composite Web Service will be affected by the confidence in the dependability of the component WSs it depends upon and by the confidence in the dependability of the composition (the design of the composition and its implementation, i.e. the 'glue' code held in the composite WS itself). The confidence naturally links two important aspects – *the value* of the dependability attribute, e.g. probability of failure on demand, with *the risk* that the particular WS delivers this attribute (e.g. its probability of failure is better than the specific value). For instance, we may want to compare two WSs, A and B, for which the confidence is expressed as follows:

- For WS A we have confidence 99% that its probability of failure on demand (*pfd*) is lower than 10^{-3}, 70% that the *pfd* is less than 10^{-4}, etc.
- For WS B we have confidence 95% that its probability of failure on demand (*pfd*) is lower than 10^{-3}, 90% that the *pfd* is less than 10^{-4}, etc.

Now which of the two WSs will be chosen depends on the dependability requirements, i.e. the particular dependability context: A will be used if the targeted *pfd* is 10^{-3}, because the confidence that this target is satisfied with WS A is higher

(99% vs. 95% with WS B). However, if a more stringent target is set, e.g. 10^{-4}, then WS B should be preferred to WS A, because the confidence that it meets the target is higher (90% vs. 70% with WS A). In the context of this – on-line upgrade management of a component WS – confidence is particularly relevant. The key idea behind an upgrade managed on-line is that the composite WS does not switch to the newest release of the component WS as soon as this new release becomes available since its dependability may suffer as a result. The new release may provide better functionality but it also brings in the increased risk that new faults may have arisen in the new release, which did not exist in the old release. A prudent policy of switching would be for the composite WS to wait until it gains *sufficiently high confidence* that the new release will not lead to deterioration of dependability.

In section 5 we show how the Bayesian inference can be applied in the context of WSs for calculating the confidence of a component WS.

3 The Web Service Upgrade Problem

A well-known problem for any component-based software development with OTS components is the upgrade of the OTS components. When a new release of an OTS component is made available the system integrator has two options:

1. Change their 'integrated' solution[5] so that it can use the new release of the OTS component. This may cause problems for the integrated solution and significant effort to rectify.
2. Stick to the old version of the OTS component and take the risk to face the consequences if the vendor of the OTS component ceases to support the old releases of the OTS component.

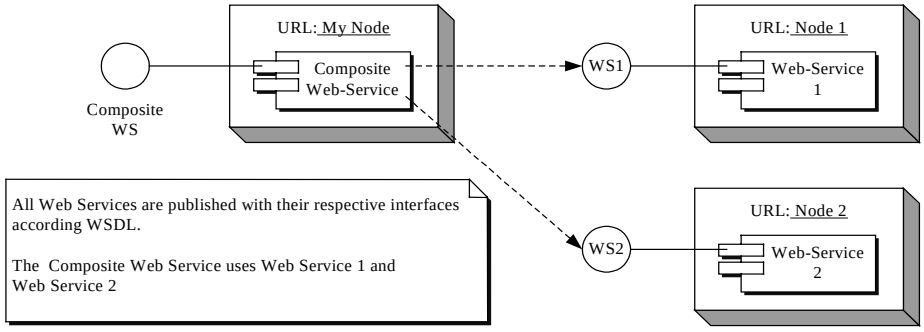

Fig. 1. A UML Deployment diagram of a composite WS, Composite Web-Service, which depends on two other component WSs provided by third parties, Web-Service 1 and Web-Service 2, respectively

The situation with a composite WS is very similar. Indeed, WS 1 and WS 2 in Fig. 1 are two component WSs used by a composite WS; conceptually this is equivalent to

[5] A term used by ECUA: http://www.esi.es/en/Projects/ecua/ecua.htm

integrating any other OTS software component in an integrated solution. There may, however, be a difference from the maintenance point of view between a composite WS and an integrated solution in which OTS components are used. In the latter case, as indicated above, the integrator has a choice whether to update the integrated solution with every new release of the OTS components or not. Such a choice may not exist in the former case of composite WSs. The deployment of a composite WS assumes that the component WSs (Web-Service 1 and Web-Service 2 in our example in Fig. 1) used by the composite WS have been deployed by their respective providers. If the providers decide to bring down their WSs the composite WS may become unavailable, too. What seems more interesting is that when the provider of a component WS, on which the composite WS depends upon, decides to update their WS the provider of the composite WS may not be even notified about the update. The composite service may be affected without its provider being able to do anything to prevent this from happening. Thus, the provider of the composite WS is automatically locked-in by the very decision to depend on another WS.

Are there ways out of the lock-in? If not, can the provider of the composite WS do something at least to make the consumers of the composite WS aware of the potential problems as a result of the update(s) which are beyond their control? Below we discuss two plausible alternatives.

3.1 Third-party Component WS Upgrade with Several Operational Releases

This scenario is depicted in Fig. 2. The choice of whether to switch to a new release of a WS used by the composite WS is with the provider of the composite WS. They may use whatever methods are available to them to assess the dependability of the new release before deciding whether or not to move to the upgraded version(s) of the used component WS.

The designer of the composite service may even make provisions at design stage of the composite WS which facilitate the assessment of the new releases of the services the composite service depends upon when these become available. An example of such a design would be making it possible to run ‘back-to-back’ the old and the new releases of the component WS used in the composite WS.

During the transitional period (i.e. after the new release, WS 1.1 in Fig. 2, becomes available) the old version of the component WS will continue to be the version used by the composite WS, but by comparing the responses coming from the old and the new release, WS 1.0 and WS 1.1 respectively, the provider of the composite WS will gain empirical evidence about how good the new release, WS 1.1, is. Once the composite service gains sufficient confidence in WS 1.1 it may switch to using it and cease using WS 1.0.

Essentially, the composite service will have to run its own ‘testing campaign’ against the new release of the WS and may use the old release as an ‘oracle’ in judging if WS 1.1 returns correct responses.

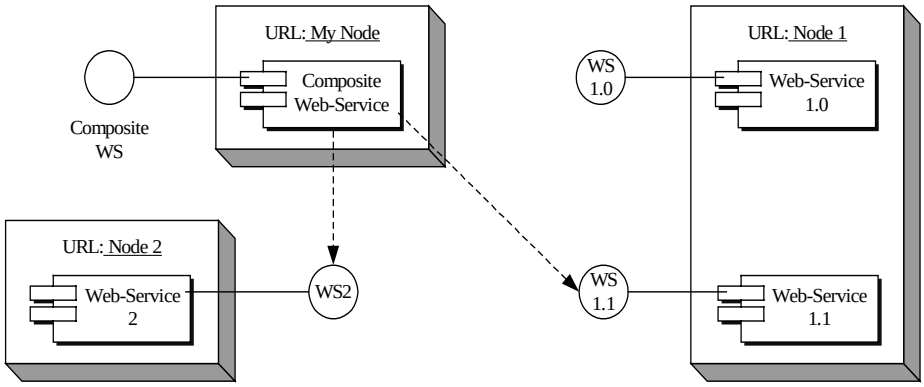

Fig. 2. A new release, Web-Service1.1, of a component WS is released, but the old version, Web-Service1.0, is also kept operational. The new release has no effect on the composite service, Composite Web-Service, as long as it continues to use the old release, Web-Service1.0, of the component WS. Eventually, the composite service is 'upgraded' to use the newer version, Web-Service 1.1

3.2 Third-party Component WS Upgrade with a Single Operational Release

Under this scenario Fig. 1 remains applicable: the most recent release of Web Service 1 will be deployed behind the interface WS 1. The options left to the provider of the composite WS are very limited. If the new release is at least distinguishable from the previous release, e.g. the release carries the release number, the provider of the composite WS will be able to detect the upgrade of the component WS and try to 'adjust' the confidence in the quality of the composite WS which may be caused by the upgrade and publish it to its consumers. A conservative view when calculating the impact of the upgrade on the dependability of the composite WS would be treating the upgraded component WS as if it were no better than the old release, i.e. the confidence in its dependability is no higher than the confidence in the old release as suggested in [12].

3.3 Own Component WS Upgrade with Several Operational Releases

In some cases a composite WS may use the component WS maintained by the same vendor. In this case the upgraded component WS will be deployed in a way which reflects the vendor's view on whether the upgraded component WS may have detrimental impact on the dependability of the own composite WSs which depends on the upgraded component WS.

We expect that even in this case, when the vendor has access to the internal details of the upgraded component WS, that prudence may dictate deployment of the new release of the component WS side by side with the old release in a special environment which has features for transparent upgrade including: interactive features for monitoring the dependability of old and new versions (including typical

adjudicator functionality for comparing their results), support for several modes of operations (using the old release only, running the old and the new releases in parallel and adjudication of their responses, switching to the new release only and phasing out the old release from the composite WS) and a standard interface (i.e. using the WSDL description of the component WS). The component WS provider should be able to monitor the way the new release of the WS is operating and choose the best way of ensuring the dependability of the service. The main difference between this form of the upgrade and the upgrade of the third-party component WS is that here the extra information that might be available about the component WS may affect the way the dependability is measured. For instance, an extensive validation and verification (e.g. regression testing and testing the bugs of the previous release on the new release or the introduction of sophisticated mechanisms of fault-tolerance in the new release of the component WS) prior to deployment may justify placing high confidence in the dependability of the new release than has been achieved in the old release. This, in theory, may justify the immediate switch of the composite WSs developed by the same vendor to using the latest release of the component WS or at least configuring the environment responsible to manage the upgrade in a way, which will require a very limited amount of operational evidence before the composite WS switches s to using the upgraded component WS.

4 Solutions for Dependable WS Upgrading

In this section we describe several architectures which allow for a managed upgrade of a WS. The architecture can be deployed as part of a composite WS in which the WS in question is used as component WS or deployed by a dependability-conscious consumer of the WS aware of the inevitable upgrade of the WS. The architecture can also be deployed by the vendor of the WS if they want to provide high dependability guarantees to the consumers of the WS. In either case the impact of the upgrade on the consumers of the WS will be minimised.

4.1 General Architecture

The general architecture for a managed WS upgrade consists of:

- a specialised middleware which runs several releases of the WS. The middleware intercepts the consumer requests coming through the WS interface, relays them to all the releases and collects the responses from the releases. It is also responsible for ‘publishing’ the confidence associated with the WS (or its releases);
- a subsystem which monitors the behaviour of the releases and assess their dependability including confidence;
- a management subsystem which adjudicates the responses from the replicas and returns an adjudicated response to the consumer of the WS. This subsystem is also responsible for reconfiguration (switching the releases on or off), recovery of the failed releases and for logging the information which may be needed for further analysis.

The architecture can be used to implement the forms of upgrade discussed above: third-party WSs (Fig. 3, 4) and own component WSs (Fig. 5).

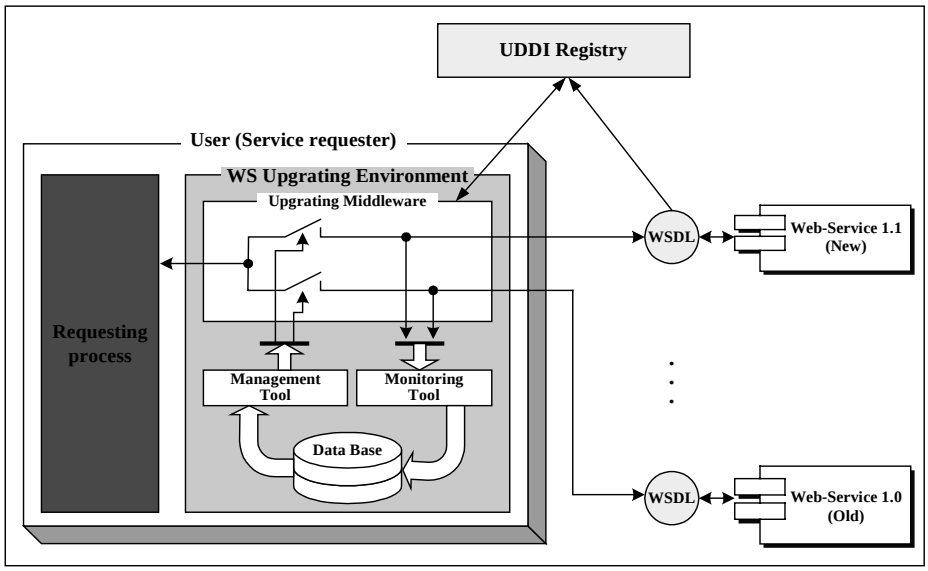

Fig. 3. Architecture for managed upgrade of third-party Web Service deployed by the consumer of the WS

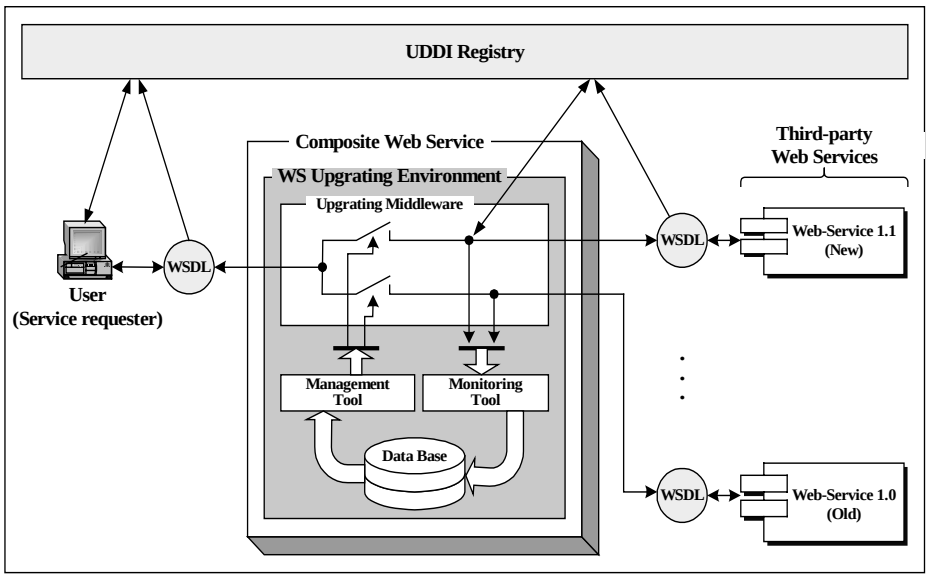

Fig. 4. Architecture for managed upgrade of third-party WS deployed as a composite WS

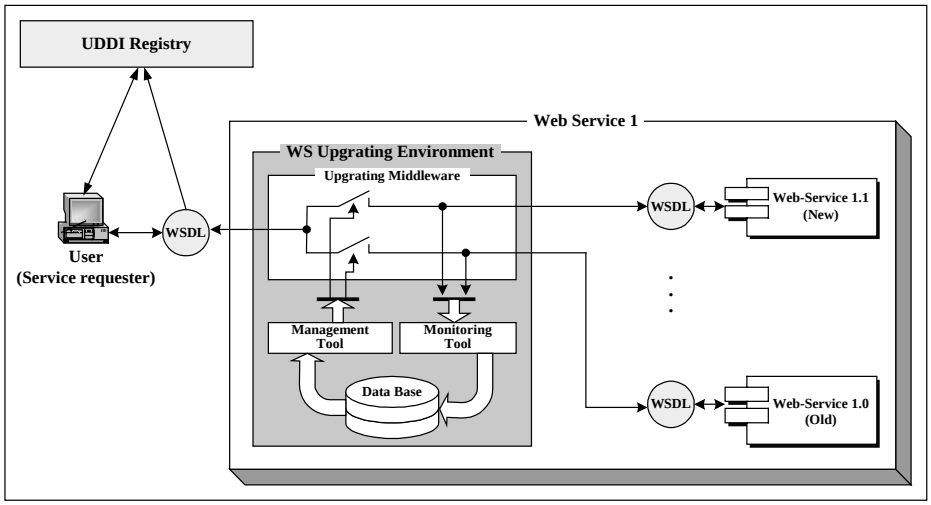

Fig. 5. Architecture for managed upgrade of a WS deployed by the vendor of the WS

The architecture for managed upgrade of third-party WS can be deployed either as part of the consumer of the WS (Fig. 3) or as a composite WS solely dedicated to the management of the upgrade (Fig. 4). The architecture shown in Fig 5 which is deployed by the WS provider and which makes the upgrade *transparent for any service subscriber* is particularly relevant in practice since it allows for optimal management of the upgrade based on full knowledge about the design and implementation of the releases available to the vendor of the WS.

4.2 Operating Modes with Several WS Releases

There are some possible operating modes of the web services with several operational releases:

1. *Parallel execution for maximum reliability.* All available releases of the WS are executed concurrently and their responses are used by the middleware to produce an adjudicated response to the consumer of the WS. Various adjudication mechanisms can be used which range from tolerating evident failures only to detecting and tolerating non-evident failures. In the latter case some form of self-checking may be need which will allow for diagnosing which of the releases has produced a (non-evidently) incorrect response before the adjudicated response can be determined.
2. *Parallel execution for maximum responsiveness.* All available releases of the WS are executed concurrently and the fastest non-evidently incorrect response is returned to the consumer of the service as an adjudicated response.
3. *Parallel execution with dynamically changed reliability/responsiveness.* It is a generalised parallel execution mode. All available releases of the WS are executed concurrently. The middleware may be configured to wait for up to a certain number of responses to be collected from the deployed releases, but no longer than

a pre-defined timeout. The actual responses collected are then adjudicated to define the response returned to the consumer of the WS. The number of responses and the timeout can be changed dynamically so that different configurations for the adjudicated response can be defined.

4. *Sequential execution for minimal server capacity*. The releases of the WS are executed sequentially (the order of execution can be chosen randomly or can be predefined). The subsequent releases are only executed if the responses received from the previous releases are evidently incorrect. A variation of this mode would be to collect more than one non-evidently incorrect responses and adjudicate them using an appropriate rule.

4.3 Monitoring and Measurement

The monitoring subsystem conducts measurement of the dependability characteristics including the confidence associated with them of the deployed releases of the WS, compares their responses.

Every time the consumer invokes the WS this subsystem monitors the availability (timeout can be used to detect if the service is down), execution time and the correctness of the responses for each releases of the WS and stores these parameters in a database. Detecting non-evident failures and diagnosing the release which has returned a non-evidently incorrect response is far from trivial. The implications of using imperfect detection/diagnosis for the confidence are scrutinised in section 5.1.

4.4 Management

The main functions of this subsystem are controlling several operational releases and choosing the current operational mode, which is based on dependability assessment conducted by the monitoring subsystem. Adjudicating the responses collected from the deployed releases and returning a response to the consumer of the WS is also a responsibility of this subsystem. The adjudication mechanisms have already been discussed together with the operating modes in section 4.2.

5 Assessment and Modelling

5.1 Bayesian Approach to Assessment of Confidence in Web-Service

In this section we illustrate how the Bayesian approach is normally applied to assessing the confidence associated with a single dependability attribute, e.g. the probability of failure on demand (*pfd*).

If the WS is treated as a black box, i.e. one can only distinguish between failures or successes (Fig. 6), the Bayesian assessment proceeds as follows.

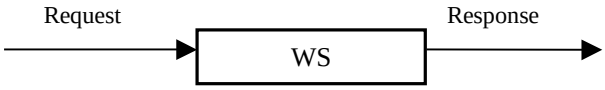

Fig. 6. Black-box model of a WS. The internal structure of the WS is unknown. Only correctness of the response (success or failure) is recorded on each request and used in the inference of the WS's *pfd*

On every request the WS may succeed, i.e. return a correct response, or fail, i.e. return an incorrect response or not return any response at all. The failure behaviour of the WS is characterised by the probability of failure (*pfd*). Let us denote it as *p*. This probability will vary between the environments in which the WS is used. The various factors, which affect the *pfd* may be unknown with certainty, thus the value of *pfd* may be uncertain. This uncertainty is captured by a (prior) probability distribution $f_p(\bullet)$, which characterises the assessor's knowledge about the system *pfd prior* to observing the WS in operation. This distribution quantifies the assessor's perception that some values of *pfd* are more likely than some other values.

Assume further that the WS is subjected to *n* requests, a sample of demands drawn from a 'realistic' operational environment (profile), and *r* failures are observed[6]. Presented with the *new evidence* the assessor may change their a priori uncertainty about the *pfd* of the WS. Now it will be represented by a posterior distribution, $f_p(\bullet \mid r,n)$, of *p* after the observations, which is defined as:

$$f_p(x \mid r,n) \propto L(n,r \mid x) f_p(x), \qquad (\mathbf{1})$$

where $L(n,r \mid x)$ is the *likelihood* of observing *r* failures in *n* demands *if* the *pfd* were exactly *x*, which in this case of independent demands is giv*en* by the *binomial* distribution, $L(n,r \mid x) = \binom{n}{r} x^r (1-x)^{n-r}$.

(1) is the general form of the Bayes's formula, applicable to any form of likelihood and any prior distribution.

Now assume that the WS is implemented as shown in Fig. 5, i.e. two releases of the WS are deployed in parallel, which see and process 'independently' a request from a consumer of the WS. On each demand (request) there are 4 possible outcomes which can be observed, given in Table 1 below. The four probabilities given in the last column of Table 1 sum up to 1. Even if these probabilities are not known with certainty, i.e. they are treated as *random variables*, their sum will be always 1. Thus, a joint probability distribution of any three (out of the four listed in Table 1) of these probabilities, e.g. $f_{p_{01},p_{10},p_{11}}(\bullet,\bullet,\bullet)$, gives an exhaustive description of the uncertainty associated with the failure behaviour of the system, which in this cases consists of WS1.0 and WS1.1. In statistical terms, the model has three degrees of freedom.

[6] The number of observed failures can be 0.

Table 1. A joint probability distribution

Event	WS 1.0	WS 1.1	Observed in *n* tests	Probability
α	Fails	Fails	r_1	p_{11}
β	Fails	Succeeds	r_2	p_{10}
γ	Succeeds	Fails	r_3	p_{01}
δ	Succeeds	Succeeds	r_4	p_{00}

The probabilities that WS 1.0 will fail, let us denote it p_A, and that WS 1.1 will fail, p_B, respectively, can be derived from the probabilities used in Table 1 as follows:
$p_A = p_{10} + p_{11}$ and $p_B = p_{01} + p_{11}$.

p_{11} represents the probability that both releases of the WS fail, hence the notation $p_{AB} \equiv p_{11}$ captures well the intuitive meaning of the event it is assigned to.

Instead of using $f_{p_{10},p_{01},p_{11}}(\bullet,\bullet,\bullet)$ we can use any other distribution, which can be derived from it through functional transformation. In this section we will use $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet)$.

It can be shown that for a given observation (r_1, r_2, and r_3 in N demands) the joint posterior distribution, $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, can be calculated as:

$$f_{p_A,p_B,p_{AB}}(x,y,z \mid N, r_1, r_2, r_3) = \frac{f_{p_A,p_B,p_{AB}}(x,y,z) L(N, r_1, r_2, r_3 \mid p_A, p_B, p_{AB})}{\iiint_{p_A,p_B,p_{AB}} f_{p_A,p_B,p_{AB}}(x,y,z) L(N, r_1, r_2, r_3 \mid p_A, p_B, p_{AB}) dxdydz}, \quad (2)$$

where $L(N, r_1, r_2, r_3 \mid p_A, p_B, p_{AB})$ is the likelihood of the observation [13].

The posterior distribution, $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$, represents the updated uncertainty about the system failure behaviour *consistent with the prior and the observations*. From this distribution one can derive the marginal uncertainties associated with the probabilities of failure of each of the releases, $f_{p_A}(\bullet \mid observation)$, $f_{p_B}(\bullet \mid observation)$ and of the probability of coincident failure of both releases, $f_{p_{AB}}(\bullet \mid observation)$. For instance the distribution of the probability of coincident failure, $f_{p_{AB}}(\bullet \mid observation)$, can be derived from $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet \mid N, r_1, r_2, r_3)$ by integrating out the 'nuisance parameters' P_A and P_B:

$$f_{P_{AB}}(x \mid r_1, r_2, r_3, n) = \int_{P_A} \int_{P_B} f_{P_{AB},P_A,P_B}(x,y,z \mid r_1, r_2, r_3, n) dP_A dP_B \quad (3)$$

Similarly the marginal posteriors, $f_{p_A}(\bullet \mid observation)$ and $f_{p_B}(\bullet \mid observation)$, can be expressed as:

$$f_{P_A}(x \mid r_1, r_2, r_3, n) = \int_{P_B} \int_{P_{AB}} f_{P_{AB}, P_A, P_B}(x, y, z \mid r_1, r_2, r_3, n) dP_B dP_{AB} \quad (4)$$

$$f_{P_B}(x \mid r_1, r_2, r_3, n) = \int_{P_A} \int_{P_{AB}} f_{P_{AB}, P_A, P_B}(x, y, z \mid r_1, r_2, r_3, n) dP_A dP_{AB} \quad (5)$$

The expressions (3-5) can be used to calculate the confidence that the pair or each of the channels meet a specific reliability target. For instance, the confidence that the probability of failure of the old release is smaller than a given target, T, will be:

$$P(P_A \mid observation \le T) = \int_0^T f_{P_A}(x \mid observation) dP_A \quad (6)$$

Using (6) we can calculate a set of *percentiles* for a set of confidence values, e.g. {90%, 95%, 99%,…}. For instance, the 99% percentile of channel A, $T_{A99\%}$, is a value of the P_A such that $\int_0^{T_{A99\%}} f_{P_A}(x \mid observation) dP_A = 99\%$.

5.1.1 Examples

We will illustrate how the Bayesian inference can be used to determine the duration of the WS managed upgrade (Fig. 5), i.e. when the old release can be replaced by the new one. We will use for this purpose several contrived but plausible scenarios.

5.1.1.1 Scenarios

Scenario 1

In this scenario we assume that the old release has been used for a very long time and, as a result, its reliability has been measured accurately: its *pfd* is believed to be 10^{-3}, and the uncertainty associated with this is *very low*. The new release has been significantly changed, compared with the old release. It is believed that the new release is an improvement, i.e. that its *pfd* is lower than the *pfd* of the old release, but since it has not seen a significant operational use there is a significant level of uncertainty about how good the new release actually is. We parameterise this scenario using the following prior distribution:

– The distribution of the *pfd* of the old release, P_A, is a Beta(α_A, β_A) distribution, $f_{p_A}(\bullet)$, defined in the range [0, 0.002] with parameters $\alpha_A = 20$, $\beta_A = 20$, i.e. the expected value of P_A is indeed 10^{-3}, consistent with the prior measurements. The parameters are chosen such that the distribution mass is concentrated in a very narrow interval, which adequately represent the low level of uncertainty about the 'true' *pfd* of the old release.

– The distribution of *pfd* of the new release, P_B, is also a Beta(α_B, β_B) distribution, $f_{p_B}(\bullet)$, defined in the same range [0, 002], with parameters $\alpha_B = 2$, $\beta_B = 3$, chosen such that the expected value of P_B is 0.8×10^{-3}, i.e. slightly better than the expected value of P_A, but the level of uncertainty about the true *pfd* of the new release is significant.
– We assumed that P_A and P_B are independently distributed, i.e. $f_{p_A,p_B}(\bullet,\bullet) = f_{p_A}(\bullet) f_{p_B}(\bullet)$.
– We assume uniform distribution of the conditional probability $P_{AB}|P_A,P_B$ in the range [0,min(P_A, P_B)], which represents the assessor being 'indifferent' about the values of the probability of coincident failure. This, in fact, is a very conservative assumption, since the expected value is 1/2 of min(P_A, P_B), i.e. the system is expected to tolerate only 50% of the failures, which seems justified given the fact that we are dealing with two releases of the same product.

Scenario 2

In this scenario we assume that the old release has been only used for a short time without a failure. The uncertainty associated with the *pfd* of the release, therefore, is significant. The new release has been produced following a very thorough development process. However since this process has never been applied in the context of WS there is a significant level of uncertainty about the *pfd* of the new release, too.

The new release is conservatively considered to be worse than the old release. This scenario is parameterised with the following prior distribution:

– The distribution of *pfd* of the old release, P_A, is a Beta(α_A, β_A) distribution, $f_{p_A}(\bullet)$, in the range [0, 0.01] with parameters $\alpha_A = 1$, $\beta_A = 10$, i.e. the expected value of P_A is $\sim10^{-3}$, but a significant level of uncertainty is built-in this prior.
– The distribution of *pfd* of the new release, P_B, is also a Beta(α_B, β_B) distribution, $f_{p_B}(\bullet)$, with parameters as in the first scenario $\alpha_B = 2$, $\beta_B = 3$. The level of uncertainty about the true *pfd* of the new release is significant.
– We assumed, again, that P_A and P_B are independently distributed, i.e. $f_{p_A,p_B}(\bullet,\bullet) = f_{p_A}(\bullet) f_{p_B}(\bullet)$.
– As in the previous scenario, we assume uniform distribution of the conditional probability $P_{AB}|P_A,P_B$ in the range [0,min(P_A, P_B)].

50,000 observations used with the two scenarios have been Monte-Carlo simulated using the following parameters:

Scenario 1: $P_A = 10^{-3}$, $P_B|A$ failed = 0.3, $P_B|A$ did not fail = 0.5×10^{-3}. The chosen parameters define a marginal probability of failure for the new release $P_B = 0.8\times10^{-3}$. Thus the marginal *pfd* of both channels are equal to the expected values of their respective distributions. The chance of coincident failure of the releases is significant: every 3 out of 10 failures of the old release will coincide with failures of the new release. This is, however, less frequent than assumed in the prior (every other failure of the less reliable channel was assumed to coincide with a failure of the more reliable channel).

Scenario 2: $P_A = 5\times10^{-3}$, i.e. the actual *pfd* is significantly worse than assumed in the prior (the mean of the prior distribution is 10^{-3}), $P_B|A$ failed = 0.1, $P_B|A$ did not fail = 0 (i.e. never failed on its own). The chosen parameters define a marginal probability of failure for the new release $P_B = 0.5\times10^{-3}$, an order of magnitude better than the old release.

5.1.1.2 Upgrade criteria (switching from managed upgrade to WS 1.1)

We will apply a few plausible alternatives of switching from the old to the new release as follows:

- *Criterion 1*: the new release, WS 1.1, reaches the dependability level offered by the old release, WS 1.0, *at the time of deploying the managed upgrade*, i.e. as defined by the prior distribution, $f_{p_A}(\bullet)$. For instance, if prior to the upgrade there $P(P_A \leq X) = 99\%$, then the managed upgrade should last until $P(P_B \leq X) = 99\%$, i.e. the same confidence, 99%, is build that WS 1.1 is better than X. This scenario does not address the possibility that during the managed upgrade the knowledge about WS 1.0 may also change: it may turn out to be worse or better than thought prior to deploying the managed upgrade.
- *Criterion 2*: the new release, WS 1.1, reaches a predefined level of dependability with a predefined level of confidence, e.g. $P(P_B \leq 10^{-3}) = 99\%$. Under this criterion the dependability of the old release, WS 1.0, prior or during the managed upgrade is irrelevant.
- *Criterion 3*: With a given confidence, e.g. 99%, the new release, WS 1.1, is better than the old release, WS 1.0. In other words, for the 99% percentiles of the releases the following inequality holds: $T_{B99\%} \leq T_{A99\%}$. Clearly, this criterion takes into account the possibility that the priors of both WS 1.0 and WS 1.1 may be 'inaccurate' and may evolve to different distributions during the managed upgrade.

5.1.1.3 Imperfection of failure detection

As described above, Bayesian inference depends on the observations and, thus, imperfection in detecting failures of WS releases will, inevitably, affect the posteriors, hence the decisions when to switch to using the new release. We simulated *omission failures* only, i.e. such that some demands on which the releases did fail were counted as being correct. This type of failure may have dangerous consequences. First, because incorrect responses may have been returned to the consumers of the WSs, and, second, because the inference may produce optimistic predictions, which, in turn, may lead to premature decisions to switch to the new release before the required confidence has been achieved. The following omission failures have been simulated:

- omission failure of the 'oracles' judging the correctness of the responses from each of the releases;
- back-to-back testing under the *pessimistic assumption* that all coincident failures will be identical and non- evident.

The first kind on failure will lead to changes of the scores on a demand of a release from '1' (failure) to '0' (success). The greater the likelihood of such a failure the more optimistic the observations become – in the extreme case when the omission

failure takes place with probability 1 – the inference will be supplied with observations 'No failure' no matter how many times the release in question has failed.

The effect of the second kind of failure on the observations will be limited to those demands on which both releases fail. In this case the scores '11' (coincident failure of both releases) will be replaced by '00' (success of both). Clearly, in real operation there may be coincident but *different failures*, which will be detected by back-to-back testing and there is a good chance that on this demand the score of at least one of the releases will be correctly counted as a failure.

We did not include in our study the 'false alarm' type of failure of the failure detection mechanisms, i.e. when an 'oracle' flags out as a failure a valid response from a release. Although in practical systems this may be a concern, its implications are not dangerous: the consumers may be required to ignore valid responses and the inference will produce *pessimistic predictions.* As a result the decision to switch to the new release may be delayed beyond the sufficient evidence that the new release has met the set dependability target.

5.1.1.4 Inference results

The results of the study are summarised below in Table 2, in which we show for the 3 criteria specified in 5.1.1.2 how long the managed upgrade (Fig. 5) should last before switching from WS 1.0 to WS 1.1. For Criterion 2 we used $P(P_B \leq 10^{-3})=99\%$ as the target for the switch.

Table 2. Duration of managed upgrade

		Criterion 1	**Criterion 2**	**Criterion 3**
Scenario 1	*Perfect 'oracles'*	35,500 demands	Not attainable (> 50,000)	40,000 demands
	Omission, $P_{omit} = 0.15$	22,000 (oscillates till 26,000)	50,000 demands	35,000 demands
	Back-to-back testing	20,000	40,000	34,000 demands
Scenario 2	*Perfect 'oracles'*	1,400 demands	10,000 demands	1,100 demands
	Omission, $P_{omit} = 0.15$	1,400 demands	7,000	1,100 demands
	Back-to-back testing	1,400 demands	6,000 demands	1,100 demands

One can see from Table 2 that the effect of the detection coverage upon the duration of the managed upgrade is significant, which is hardly surprising. We further use 90% and 99% percentiles to illustrate the relationship between the failure detection coverage and the confidence. Fig. 7, clearly indicates, however, the link that exists between the imperfection of the detection mechanism deployed and the confidence in having achieved the specified target, Criterion 1 in this case. For instance, consider the 90% percentile that the new release is as reliable as the old release was prior to the upgrade (the solid thin curve in Fig. 1). This percentile remains always lower not only than the 99% percentile with perfect oracle (which is always the case), but also than the 99% percentile with imperfect oracles which miss a failure with probability 0.15. In other words, using imperfect oracles with detection

coverage of 85% (which is often seen as achievable, e.g. [14]) and using Bayesian inference in this case means that the confidence error caused by the imperfection of the oracles is no greater than 9%. At any stage of the inference what the assessor would consider to be a 99% percentile on the *pfd* of the new release will actually be no worse than 90% percentile.

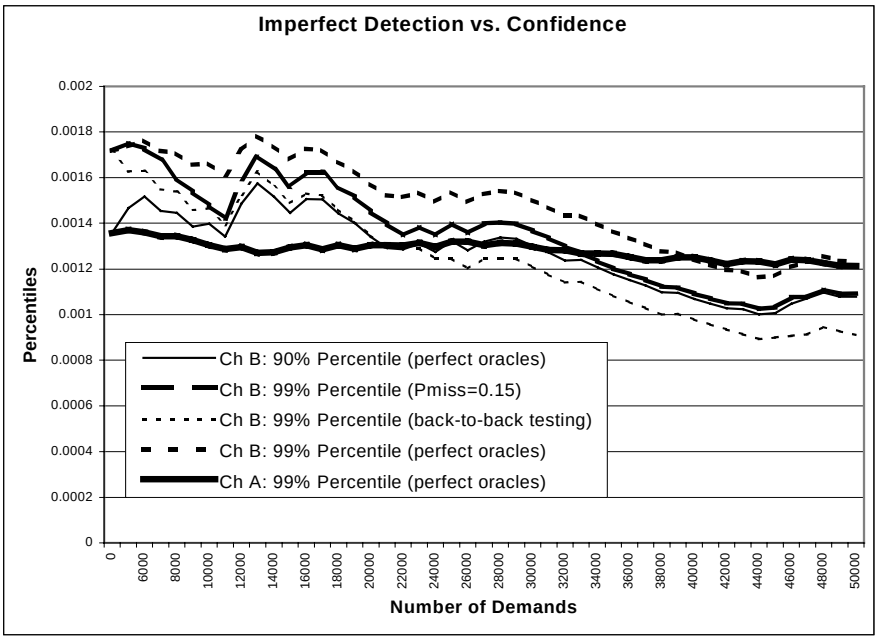

Fig. 7. Scenario 1: percentiles for perfect and imperfect failure detection

The difference between the inference with perfect detection and back-to-back testing is slightly different – the error up to ~20,000 demands does not exceed 9% and then becomes greater than 9%. Incidentally, it is 20,000 demands when the decision will be taken to switch to the new release (Table 2), i.e. the actual confidence achieved at this time will be no worse than 90%.

It is worth mentioning that the number of demands needed in order to get the required level of confidence under Scenario 1 are significant. This is a consequence of the *pfd* targets being very close to what the real reliability of the new release is (the explicitly stated target of 10^{-3} remains unattainable with perfect detection even after 50,000 demands). A significant number of failures is observed which does not allow the assessor to build quickly the required confidence.

Scenario 2 in this respect is very different – the targets to be met by the new release are significantly worse than what the 'true' *pfd* of this release is. Under this scenario the prior confidence in the old release was also low due to the minimal operation exposure of this release. As a result, meeting the set targets (with all 3 criteria) requires significantly fewer demands. The effect of imperfect detection on the decision to switch to the new release under this scenario is illustrated in Fig. 8. The 90% percentile with perfect failure detection remains lower than the 99%

percentile with imperfect detection throughout the entire range of demands of interest, including the values when a decision to switch to the new release will be taken (up to 7000 demands). Thus, again, the effect imperfect of the failure detection on the confidence is relatively modest – the error is less than 9%.

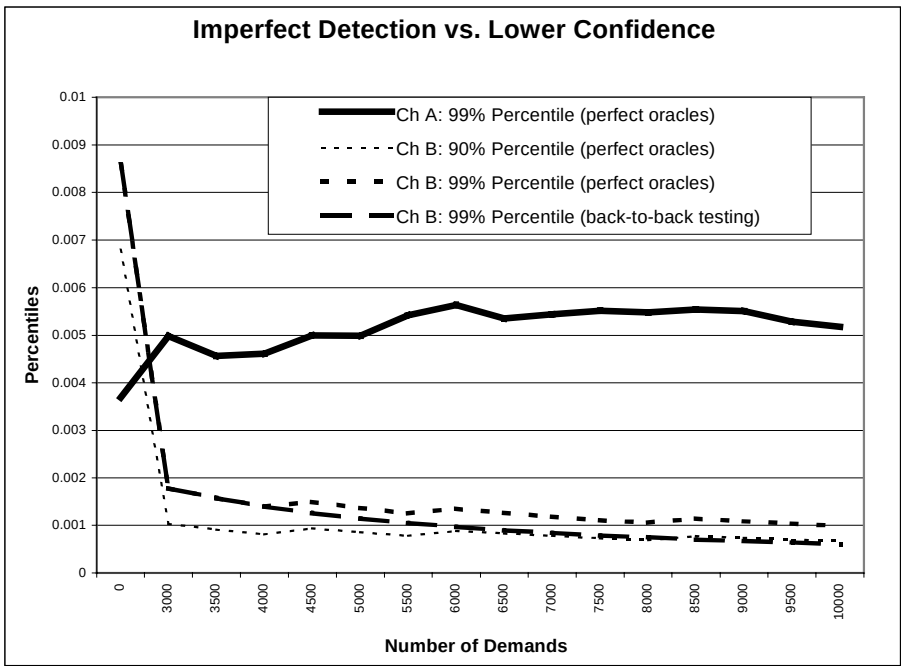

Fig. 8. Scenario 2: percentiles for perfect and imperfect failure detection

5.1.2 Realism of Bayesian inference in the context of Web-Services

*Two important aspect*s of using Bayesian assessment in the context of Web-services are worth mentioning:

1. The choice of a prior is important, as it is for Bayesian inference in any other context. In the context of a managed upgrade we use a white-box inference, which requires a trivariate distribution, $f_{p_A,p_B,p_{AB}}(\bullet,\bullet,\bullet)$ to be defined. This is far from trivial. It is reasonable to expect that the marginal distributions, $f_{p_A}(\bullet)$ and $f_{p_B}(\bullet)$, will be available in some form. The old release will have seen some (in some cases long) operational exposure, which is relatively straightforward to translate into confidence using a black-box inference. The new release will have been subjected to some level of testing, but the prior will be largely based on *expert judgement*, and thus will leave significant uncertainty about the true value of the *pfd*. The real difficulty is in defining the third variate of the distribution, which characterises how likely the two releases of the WS are to fail together. Since we are dealing with two releases of the same service it is plausible to assume

high level of correlation between their failures[7]. The 'indifference' assumption, which we used in our examples, seems a safe option. Even if the prior is inaccurate we leave it 'open to quick changes' as new empirical evidence becomes available ('data will speak for itself'). The downside of the safe option is that it will take some time for the error in the prior to be compensated, which will delay the decision to switch to the new release. This is, however, the inevitable price to be paid for the extra assurance that the WS dependability *will not deteriorate* as a result of the upgrade.

2. Coverage of the deployed failure detection mechanism. We have only touched upon this aspect due to the limited space and further studies are needed in order to get further insight into the relationship between the detection coverage and the confidence achieved. The results presented here seem to suggest that this problem can be tackled relatively simply. In our examples we saw a consistent picture that ~10% imperfection of failure detection translates into error in predicted confidence less than 10%. If this is consistent across many systems a relatively simple engineering *rule of thumb* can be defined so that despite the errors the needed confidence is *actually achieved* and one can proceed with the switch to the new release. In fact the limited coverage of failure detection is not necessarily a problem! It is clearly a problem when an explicit dependability target for the new release is stated (Criterion 2), but we doubt that this criterion is appropriate for the context of the WS upgrade. Indeed, it seems always worth deploying the more dependable release, even if it does not meet an explicitly stated higher target. Making a decision to switch to the new release based on comparison of the two releases (i.e. Criterion 1 or 3 defined above) seems much more plausible than wait until the new release meets an explicit target. Under these circumstances both releases will have been affected by the limited coverage of the deployed detection mechanisms. It seems reasonable to assume that the imperfect detection will affect *both releases similarly*, hence the decision to switch even based on inaccurate measurements will be justified.

Despite these problems, it seems clear that Bayesian inference can be used to assess the confidence in dependability of WS releases and control the managed upgrade of WSs.

5.2 Simulation Modelling of the Dependable WS Upgrading

5.2.1 Model Description

An event-driven simulation model, executed in the MATLAB 6.0 environment, was developed to analyse the effectiveness, both in terms of improved dependability and performance, of the managed WS upgrade. Below we present the simulation

[7] This plausible view is counterbalanced by the empirical fact that in many cases the new releases of software products may fail in circumstances where the old releases do not. Thus, even if the new release fixes faults in the old release(s) it is far from clear whether the new release is always an improvement and what might be a plausible expectation regarding the coincident failures.

results obtained for running concurrently two releases of a WS. The middleware for managed upgrade implements the following rules:

1. A request from a consumer is forwarded to both releases;
2. The middleware waits to collect responses from the releases, but no longer than a predefined *Timeout*. The collected responses are adjudicated and the consumer of the WS is presented with the adjudicated response. The implemented adjudication rules are as follows:

– if all collected responses are evidently incorrect then the middleware raises an exception (i.e. the adjudicated response itself is evidently incorrect);
– if all releases return the same response (correct or non-evidently incorrect) then this response is returned to the consumer of the Web Service, too;
– if all the responses collected from the releases are valid (i.e. none is evidently incorrect), then the middleware returns to the consumer of the Web service a response, selected at random from the ones collected. Clearly, even if a correct response exists among the collected ones a possibility still exists that the consumer of the Web service gets an incorrect response, when the middleware picks at random an incorrect response from those collected;
– if the TimeOut expires and a single valid response is collected this response is returned to the consumer of the Web service, which may turn out to be non-evidently incorrect.
– if no response has been collected the middleware returns a response 'Web Service unavailable'.

It takes each release some time (execution time) to respond to a request. The execution times of the releases may be affected by various factors. The execution time is modelled as a sum of two components as follows:

$$\text{Ex. Time(Release(i))} = T1 + T2(i) \quad (7)$$

where T1 – is the same for both releases and models the computational difficulty of the demand, which is common for both releases, while T2(i) may differ for the two replicas and may be due to differences between the releases. Both T1 and T2 are simulated as exponentially distributed random variables, exp(T1Mean), exp($T2Mean_1$) and exp($T2Mean_2$), respectively, with different parameters.

The overall execution time of the *system* with several operational releases of the WS is calculated as:

$$\text{Ex. time(WS)} = \min(\text{TimeOut}, \max(\text{Ex. time(Release(i))}) + dT \quad (8)$$

where dT is the time taken by the middleware to adjudicate the release responses.

The behaviour of the releases is simulated under the assumption that a degree of correlation between the types of responses exists which is modelled through a set of conditional probabilities:

$$P(\text{slower response is X} \mid \text{faster response is Y}) \quad (9)$$

Where the types of responses (X and Y) are:

– correct (CR);
– evident failure (ER);
– non-evident failure (NER).

A special case would be independence of the behaviour of the releases (i.e. the type of response they returns on demand), which is included in our results for reference, although it is clearly unrealistic.

5.2.2 Simulation Settings

Table 3. Marginal probabilities associated with the responses of the releases

Run	Independent probabilities for different outcomes					
	Release 1 (Rel_1)			Release 2 (Rel_2)		
	CR	ER	NER	CR	ER	NER
1	0.70	0.15	0.15	0.70	0.15	0.15
2	0.70	0.15	0.15	0.60	0.20	0.20
3	0.70	0.15	0.15	0.50	0.25	0.25
4	0.60	0.20	0.20	0.40	0.30	0.30

Table 4. Conditional probabilities associated with the response from the slower release (10)

Run	Condition		Probabilities: P(outcome Rel_2 | outcome Rel_1)		
			CR	ER	NER
1	Outcome of Release 1	CR	0.90	0.05	0.05
		ER	0.05	0.90	0.05
		NER	0.05	0.05	0.90
2	Outcome of Release 1	CR	0.80	0.10	0.10
		ER	0.10	0.80	0.10
		NER	0.10	0.10	0.80
3	Outcome of Release 1	CR	0.70	0.15	0.15
		ER	0.15	0.70	0.15
		NER	0.15	0.15	0.70
4	Outcome of Release 1	CR	0.40	0.30	0.30
		ER	0.30	0.40	0.30
		NER	0.30	0.30	0.40

The execution times were simulated with the following parameters:

- T1Mean=0.7 sec;
- $T2Mean_1 = T2Mean_2$=0.7 sec;
- dT=0.1 sec.

The choice of simulation parameters was dictated by us trying to cover realistic scenarios. In particular we varied widely the degree of correlation between the behaviour of the simulated channels both in terms of correct and different types of incorrect responses.

5.2.3 Simulation Results

The simulation results – mean execution time and number of responses of different types - are presented in Tables 4 and 5 obtained on 10,000 requests processed under different regimes, as defined in section 5.2.2.

Table 5. Simulation results assuming positive correlation between release failures

Run	Observations		TimeOut = 1.5 sec			TimeOut = 2.0 sec			TimeOut = 3.0 sec		
			Rel_1	Rel_2	**System**	Rel_1	Rel_2	**System**	Rel_1	Rel_2	**System**
1	MET[8]		1.0077	1.0054	**1.2194**	1.0077	1.0054	**1.2290**	1.0077	1.0054	**1.2357**
	Outcomes	CR	6709	6230	**6762**	6785	6301	**6815**	6840	6348	**6851**
		EER	1443	1668	**1449**	1460	1690	**1470**	1470	1706	**1475**
		NER	1412	1664	**1463**	1428	1676	**1472**	1437	1686	**1480**
		Total	9564	9562	**9674**	9673	9667	**9757**	9747	9740	**9806**
	NRDT[9]		436	438	**326**	327	333	**243**	253	260	**194**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
2	MET		0.9955	0.9912	**1.2052**	0.9955	0.9912	**1.2148**	0.9955	0.9912	**1.2214**
	Outcomes	CR	6733	5706	**6683**	6819	5764	**6755**	6866	5802	**6780**
		EER	1420	1944	**1502**	1436	1964	**1506**	1452	1982	**1529**
		NER	1414	1941	**1504**	1434	1962	**1514**	1447	1983	**1522**
		Total	9567	9591	**9689**	9689	9690	**9775**	9765	9767	**9831**
	NRDT		433	409	**311**	311	310	**225**	235	233	**169**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
3	MET		0.9870	0.9949	**1.2153**	0.9870	0.9949	**1.2153**	0.9870	0.9949	**1.2213**
	Outcomes	CR	6777	5231	**6661**	6777	5231	**6672**	6823	5268	**6702**
		EER	1438	2217	**1530**	1438	2217	**1521**	1449	2230	**1526**
		NER	1492	2269	**1611**	1492	2269	**1609**	1503	2283	**1618**
		Total	9707	9717	**9802**	9707	9717	**9802**	9775	9781	**9846**
	NRDT		293	283	**198**	293	283	**198**	225	219	**154**
	Total req.		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
4	MET		0.9966	0.9925	**1.2097**	0.9966	0.9925	**1.2183**	0.9966	0.9925	**1.2246**
	Outcomes	CR	6744	3519	**6395**	6808	3559	**6462**	6845	3581	**6491**
		EER	1434	3016	**1635**	1444	3042	**1629**	1457	3065	**1631**
		NER	1436	3076	**1679**	1456	3106	**1689**	1467	3134	**1705**
		Total	9614	9611	**9709**	9708	9707	**9780**	9769	9780	**9827**
	NRDT		386	389	**291**	292	293	**220**	231	220	**173**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**

[8] MET – mean execution time, in sec.

[9] NRDT – no response received within TimeOut

Table 6. Simulation results assuming independence of release failures

Run	Observations		TimeOut = 1.5 sec			TimeOut = 2.0 sec			TimeOut = 3.0 sec		
			Rel_1	Rel_2	**System**	Rel_1	Rel_2	**System**	Rel_1	Rel_2	**System**
1	MET		0.9995	0.9959	**1.2095**	0.9995	0.9959	**1.2191**	0.9995	0.9959	**1.2267**
	Outcomes	CR	6729	6647	**7759**	6794	6709	**7812**	6852	6770	**7853**
		EER	1406	1447	**755**	1424	1458	**758**	1432	1473	**768**
		NER	1453	1481	**1177**	1471	1496	**1194**	1483	1514	**1201**
		Total	9588	9575	**9691**	9689	9663	**9764**	9767	9757	**9822**
	NRDT		412	425	**309**	311	337	**236**	233	243	**178**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
2	MET		1.0086	1.0081	**1.2239**	1.0086	1.0081	**1.2327**	1.0086	1.0081	**1.2386**
	Outcomes	CR	6730	5712	**7396**	6805	5780	**7470**	6856	5824	**7509**
		EER	1428	1928	**1021**	1443	1947	**1017**	1454	1956	**1013**
		NER	1424	1949	**1286**	1446	1971	**1292**	1455	1992	**1309**
		Total	9582	9589	**9703**	9694	9698	**9779**	9765	9772	**9831**
	NRDT		418	411	**297**	306	302	**221**	235	228	**169**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
3	MET		0.9856	0.9894	**1.2013**	0.9856	0.9894	**1.2107**	0.9856	0.9894	**1.2175**
	Outcomes	CR	6700	4816	**6982**	6775	4869	**7039**	6834	4904	**7079**
		EER	1432	2400	**1203**	1446	2424	**1226**	1459	2445	**1245**
		NER	1458	2378	**1510**	1471	2404	**1515**	1483	2436	**1519**
		Total	9590	9594	**9695**	9692	9697	**9780**	9776	9785	**9843**
	NRDT		410	406	**305**	308	303	**220**	224	215	**157**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**
4	MET		0.9884	0.9926	**1.2031**	0.9884	0.9926	**1.2126**	0.9884	0.9926	**1.2193**
	Outcomes	CR	6687	3855	**6624**	6762	3887	**6680**	6813	3917	**6704**
		EER	1419	2823	**1416**	1434	2865	**1429**	1444	2885	**1444**
		NER	1484	2886	**1656**	1504	2928	**1672**	1518	2955	**1687**
		Total	9590	9564	**9696**	9700	9680	**9781**	9775	9757	**9835**
	NRDT		410	436	**304**	300	320	**219**	225	243	**165**
	Total requests		10000	10000	**10000**	10000	10000	**10000**	10000	10000	**10000**

The simulation results can be summarised as follows:

1. The system availability offered by the architecture for managed upgrade is higher than the availability of each of the versions. This is to be expected since the system is a 1-out-of-2 system. This observation is important because it reduces the pressure of having to switch to the new release quickly. From the point of view of dependability the managed upgrade is the best alternative – the 1-out-of-2 by definition is no worse than the more reliable channel. Thus we can prolong the switch to the new release as long as necessary without any negative implications for the dependability of the service.
2. The mean execution time recorded for the system is greater than for the individual releases. This is the price for the improved dependability assurance provided by the fault-tolerant architecture – it waits for the second (i.e. slower) response before adjudicating the responses. Some improvement can be achieved by returning to the consumer the fastest response as soon it is received. dT is inherent for the chosen

architecture and cannot be eliminated. The performance penalty inevitable with the managed upgrade is the real reason for us to try to minimise its duration.

3. Somewhat unexpected result from this simulation is the fact that when the releases are assumed highly correlated (the first run in Table 5 with correlation between the releases 0.9) the reliability of the system is higher than the reliability of either of the two releases. When the correlation between the releases goes down (runs 2-4 in Table 5 with correlation 0.8 – 0.4) the system reliability remains better than the less reliable release (normally the old release) but is now worse than the reliability of the better release (normally the new release). This observation, true with respect to all types of responses - correct and incorrect – may be due to the specific way the correlation between the releases has been parameterised (Table 4). A more detailed study with a wider variety of values and different combinations of the conditional probabilities will provide further details about the interplay between the properties of the individual releases and of the chosen architecture for managed upgrade.
4. For the second set of simulation runs (Table 6) under the assumption that the responses of the releases are independent, the system reliability is better than the reliability of both releases. This observation is good news – fault-tolerance works. However, the result does not seem particularly useful because the assumption of independence is implausible: after all the two releases are likely to be very similar (significant portion of the code will be reused in the newer release). Software faults present in the older release and not fixed in the newer release will lead to identical failures.

The obtained results provide indications of the potential usefulness of the architecture and of its limitation. Through extensive simulation one can identify the range of possibilities which can be encountered in practice. The particular parameters of a real life-system, e.g. which set of conditional probabilities describes best the concrete system at hand, of course, is unknowable. However, the simulation results may help in shaping the ‘prior’ for a Bayesian assessment of the chosen architecture for a managed upgrade, as described in section 5.1 above.

6 Implementation

6.1 Test harness

A test harness is under development for experimenting with the architecture for a managed upgrade of a third-party WS deployed as a composite WS (Fig. 4). It allows the requests to the WS to be forwarded to the deployed releases of the WS transparently for the consumers of the WS. When responses from the releases are collected, the test harness adjudicates them and returns a response to the respective consumer.

The test harness monitors the responses, using the calculated confidence in their dependability and adjusts the adjudication accordingly. The consumers of the WS will

be offered a set of operations for changing the configuration of the test harness according to their preferences:

- users can add new or remove some of the old releases of the WS (add or remove URI to the WSDL description of the WS releases)
- users can specify the operational modes of the composite WS (serial or concurrent execution of the deployed releases)
- users can explicitly specify the adjudication mechanism they would like applied to their own requests to the WS (e.g. majority voter or other plans)
- the user can read back the confidence associated with each of the deployed releases of the WS and calculated by the harness for different dependability attributes (e.g. confidence in correctness, confidence in availability, etc.).

The test harness is being developed in Java using IBM WebSphere SDK for Web Services[10] (WSDK). Currently under development is the visual environment for the managed upgrade of own and third-party WS, for which the Eclipse IDE[11] will be extended with a specialised plug-in, also under development.

6.2 'Publishing' the Confidence in Dependability of Web Services

In this section we discuss some practical ways of 'publishing' the confidence (or indeed any other dependability related measure) using the adopted standards for WSs. The confidence is a probability and can be accurately represented by a floating point number. To illustrate the idea of publishing the confidence let us consider a contrived example of WS with the following fragment of its WSDL description:

```
<types>
  <s:schema … >
    <s:element name=″Operation1Request″>
      <s:complexType>
        <s:sequence>
          <s:element minOccurs=″0″ maxOccurs=″1″
             name=″param1″ type=″s:int″>
          <s:element minOccurs=″0″ maxOccurs=″1″
             name=″param2″ type=″s:string″>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name=″Operation1Response″>
      <s:complexType>
        <s:sequence>
          <s:element minOccurs=″0″ maxOccurs=″1″
             name=″Op1Result″ type=″s:string″>
        </s:sequence>
      </s:complexType>
    </s:element>
  …
</types>
```

In other words, the WS interface publishes an operation ”operation1” which requires two parameters when invoked, ”param1” of type int and ”param2” of type string, and

[10] http://www-106.ibm.com/developerworks/webservices/wsdk/

[11] www.eclipse.org

returns a result ”Op1Result” of type string.[12] Now assume that the WS provider wishes to ‘publish’ the calculated confidence in the correctness of ”operation1”.

There are two ways of doing it:

– The response to a consumer invoking ”operation1” can be changed as follows:

```
<s:element name=”Operation1Response”>
  <s:complexType>
    <s:sequence>
      <s:element minOccurs=”0” maxOccurs=”1”
         name=”Op1Result” type=”s:string”>
      <s:element minOccurs=”0” maxOccurs=”1”
         name=”Op1Conf” type=”s:double”>
    </s:sequence>
  </s:complexType>
</s:element>
```

– A new operation is defined which takes as a parameter the name of an operation (for which the consumer seeks confidence) and returns the confidence in the quality of the operation:

```
<s:element name=”OperationConfRequest”>
      <s:complexType>
        <s:sequence>
          <s:element minOccurs=”0” maxOccurs=”1”
             name=”operation” type=”s:string”>
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name=”OperationConfResponse”>
      <s:complexType>
        <s:sequence>
          <s:element minOccurs=”0” maxOccurs=”1”
             name=”Op1Conf” type=”s:double”>
        </s:sequence>
      </s:complexType>
    </s:element>
```

The advantage of the first implementation is that the confidence is associated with every execution of ”operation1”. The obvious disadvantage is that the new WSDL description is not backward compatible with the old one, which is not acceptable for the existing WS but may be OK for newly deployed services.

The advantage of the second solution is that the new WSDL is backward compatible with the old WSDL. The disadvantage is that the confidence will have to be extracted in a separate invocation of a different operation published by the service (“OperationConf” in the example above), which may lead to complications.

Finally, a third option exists, which combines the advantages of both solutions given above. It consists of defining a new operation, e.g. ”operation1Conf”, in which the response is extended by a number providing the confidence in the correctness of the operation. This approach allows the ‘confidence conscious’ consumers to switch to using ”operation1Conf”, while it does not break the existing client applications which can continue to use ”operation1”, i.e. backward compatibility is achieved.

[12] For the sake of brevity the fragments of the WSDL description related to messages, parts and the service are not shown.

The confidence will have to be updated when necessary (e.g. by the service provider). The clients will be able to get this information directly from the UDDI archive. Both the clients and the provider will be able to keep this up to date. This will, for example, allow the clients to collect and publicise information about the confidence in the service, which in many situations is the most appropriate way of collecting information about confidence as only the clients know exactly if the service provided is correct. However, an architectural solution in which the WSDL description of a WS is extended with additional information reflecting confidence in this service, as was shown above, is more static.

Another two solutions are possible. The first one, which uses protocol handlers on the service and client sides to transparently add/remove additional information describing confidence to/from each XML message sent between the WS and clients, is more structured and transparent. The protocol handlers should be able to understand the additional information in the same way on both sides. This architectural solution completely separates the application functionality from dealing with the confidence-related issues and ensures compatibility in that when there is no handler on the client side it keeps functioning.

The Web Service architecture allows us to develop another solution, which consists of a dedicated trusted confidence service functioning as a mediator for all messages sent to and from the WS. This mediator can monitor all messages and express the confidence in a convenient way; an example of such an intermediary is given in section 4.1 (Fig. 4). The advantage of this solution is a complete separation of confidence from the client and service functionality. Moreover, it may be beneficial to use such mediators as trusted-third parties in online negotiations between clients and services. A disadvantage of this solution, clearly, is that the operational 'evidence' about how good the WS is will be generated by the traffic produced by the consumers connected to the intermediary. In case significant traffic bypasses the intermediary, i.e. many consumers interact directly with the WS, the confidence reported by the intermediary may be out of date.

7 Discussion and Conclusions

7.1. Related Work

Paper [15] discusses an architectural framework that allows a WS to be distributed into a number of nodes. The specific focus is on supporting uninterrupted service when a service migrates from one node to another. This approach cannot be directly used for WS upgrading when we want to make use of natural redundancy and diversity existing in the system with old and new releases and when we want to make decisions by measuring confidence in the old and new releases of an WS. Moreover, our solution guarantees uninterrupted service. The approach proposed in [15] does not explicitly work with any dependability-related characteristics of the WS (such as confidence).

The Hercules framework [16] relies on the same idea [3] of ensuring reliability of upgrading by employing old and new releases. But the main focus of this work is on formal specification of specific subdomains on which different releases of a component work correctly. Our approach uses confidence in service as the main characteristics used to reasoning about its dependability. Moreover, our technique is oriented towards the Web Service architecture with a special emphasis on service specification description and using service registries to publicize services.

7.2. Outstanding Issues

Due to space limitations we could not address several practical aspects of implementing the proposed managed upgrade. A few are discussed in this section, while others will be covered in our future work.

One of the reasons for introducing the managed upgrade is the lack of notification of consumers when an WS is upgraded, which may be useful in the context of the managed upgrade, e.g. if the managed upgrade is deployed by consumers. Here we explicitly discuss various ways for implementing such notification. It could be used to initiate the managed upgrade from the old to the new release. There are several degrees of notification and various ways of implementing it. One possibility is to use the existing registry mechanism and extend the WSDL description of a WS by adding a reference to a new release of a WS; this would allow a consumer to detect this with both releases staying operational. Another possibility is to use a WS notification service[13] as a separate mechanism to inform all the consumers of a WS about a new release. A similar approach would be to explicitly notify subscribers (consumers) using some form of "callback" function to consumers of a WS.

Another problem with the proposed approach to using the confidence in the dependability of the releases is defining a plausible 'prior' about the dependability of the new release. A related issue, which affects the accuracy of the confidence in the dependability of the releases and the effectiveness of the managed upgrade, is the perfection of the 'oracles' (adjudicators) of the responses from the releases. We touched upon these problems in section 5.1 and provided some initial assessment of the impact of imperfect detection on the predicted confidence. However, further extensive studies are needed, e.g. via simulation, to assess how severe the problem of imperfect detection is. More importantly, such studies may allow for measures to be found which, if put in place, e.g. implemented in the middleware for the managed upgrade, will reduce the problem to an acceptable level.

7.3. Conclusion

We have addressed various aspects of a dependable on-line upgrade of a WS. We concentrated on the managed upgrade in which two releases of the service can be deployed and discussed the implications of using a standard fault-tolerant architecture

[13] http://www-106.ibm.com/developerworks/webservices/library/specification/ws-notification/

in which the releases are used as ‘independent’ channels. We argued that the confidence in dependability can be calculated and used to make a decision when to switch the consumers of the WS from the old to the new release: when the confidence in the dependability of the new release becomes ‘sufficiently’ high. Through simulation we confirmed that the managed upgrade can deliver some improvement compared with the situations when either of the releases is used.

Finally, we discussed the advantages and disadvantages of various alternative ways of deploying the managed upgrade: i) by the consumers of the service, ii) by the provider or iii) by an independent broker.

Acknowledgements. This work is partially supported by the Royal Society grant (RS 16114) and by the UK Engineering and Physical Sciences Research council (EPSRC) (DOTS Project). A. Romanovsky is partially supported by IST RODIN project (IST 511599).

References

1. W3C Working Group, Web Services Architecture. 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/
2. Romanovsky, A. and I. Smith. Dependable On-line Upgrading of Distributed Systems COMPSAC'2002. 2002. Oxford. p. 975-976.
3. Randell, B., System Structure for Software Fault Tolerance. IEEE Transactions on Software Engineering, 1975. SE-1(2): p. 220-232.
4. Ferguson, D.F., T. Storey, et al., Secure, Reliable, Transacted Web Services: Architecture and Composition. 2003, Microsoft and IBM.
5. Tartanoglu, F., V. Issarny, et al., Dependability in the Web Service Architecture, in Architecting Depndable Systems. 2003, Springer-Verlag. p. 89-108.
6. Avizienis, A., J.-C. Laprie, et al., Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing, 2004. 1(1): p. 11-33.
7. AmperPoint, Managing Exceptions in Web Services Environment. 2003. http://www.eaiindustry.org/docs/member%20docs/amberpoint/AmberPoint_Managing_Exceptions.pdf
8. Chandra, S., Chen, P. M. Whither Generic Recovery from Application Faults? A Fault Study using Open-Source Software International Conference on Dependable Systems and Networks (DSN'2000). 2000, June. NY, USA. p. 97-106.
9. Deswarte, Y., K. Kanoun and J.-C. Laprie. Diversity against Accidental and Deliberate Faults Computer Security, Dependability and Assurance: From Needs to Solutions. 1998. York, England and Washington, D.C., USA: IEEE Computer Society Press.
10. Kharchenko, V., P. Popov and A. Romanovsky. On Dependability of Composite Web Services with Components Upgraded Online. In Supplemental Volume Workshop on Architecting Dependable Systems (WADS-DSN'2004). 2004. Florence, Italy. p. 287-291.
11. Box, G.E.P. and G.C. Tiao, Bayesian Inference in Statistical Analysis. 1973: Addison-Wesley Inc. 588.
12. Littlewood, B. and D. Wright, Some conservative stopping rules for the operational testing of safety-critical software. IEEE Transactions on Software Engineering, 1997. 23(11): p. 673-683.

13. Littlewood, B., P. Popov and L. Strigini, Assessing the Reliability of Diverse Fault-Tolerant Software-Based Systems. Safety Science, 2002. 40: p. 781-796.
14. Cukier, M., D. Powell and J. Arlat, Coverage Estimation Methods for Stratified Fault-Injection. IEEE Transactions on Computers, 1999. 48(7): p. 707-723.
15. Alwagait, E. and S. Ghandeharizadeh. DeW: A Dependable Web Services Framework 14th International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications (RIDE'04). 2004. Boston, Massachusetts. p. 111-118.
16. Cook, J.E. and J.A. Dage. Highly Reliable Upgrading of Components The 21st International Conference on Software Engineering (ICSE 1999). 1999. p. 203-212.