Towards Pattern-Based Reliability Certification of Services

Ingrid Buckley[1], Eduardo B. Fernandez[1], Marco Anisetti[2], Claudio A. Ardagna[2], Masoud Sadjadi[3], and Ernesto Damiani[2]

[1] Department of Electrical Engineering and Computer Science
Florida Atlantic University
777 Glades Road, Boca Raton, Florida, USA
ibuckley@fau.edu,ed@cse.fau.edu
[2] Dipartimento di Tecnologie dell'Informazione
Università degli Studi di Milano
Via Bramante 65, 26013 Crema (CR), Italy
firstname.lastname@unimi.it
[3] School of Computing and Information Sciences
Florida International University
University Park, 11200 SW 8th St., Miami, Florida, USA
sadjadi@cs.fiu.edu

Abstract. On Service-Oriented Architectures (SOAs), the mechanism for run-time discovery and selection of services may conflict with the need to make sure that business process instances satisfy their reliability requirements. In this paper we describe a certification scheme based on machine-readable reliability certificates that will enable run-time negotiation. Service reliability is afforded by means of reliability patterns. Our certificates describe the reliability mechanism implemented by a service and the reliability pattern used to implement such a mechanism. Digital signature is used to associate the reliability claim contained in each certificate with the party (service supplier or accredited third-party) taking responsibility for it.

1 Introduction

When Service-Oriented Architectures (SOA) came of age, no specific reliability technology for Web services was available; reliability mechanisms designed for Web servers, such as server redundancy, were used in its stead. Later, standards for reliable service invocation like WS-Reliability and WS-ReliableMessaging emerged [4], but the problem of achieving reliability of SOA-based Web services remains unsolved.

When trying to address the problem of Web service reliability, one has to recognize that many of the features that make SOAs attractive, such as run-time service recruitment and composition, conflict with traditional reliability models and solutions. Reliability relates to a system's ability to function correctly in the presence of faults, errors, and failures. This ability is traditionally achieved at

R. Meersman, T. Dillon, and P. Herrero (Eds.): OTM 2011, Part II, LNCS 7045, pp. 560–576, 2011.

system design time; but on a SOA – especially one hosted on a virtual execution environment like a cloud – the "system" that will execute a given business process instance is put together run-time via service discovery and selection. How can we ensure that the overall reliability requirements will be met? Our proposal to answer this question is a scheme for the *reliability certification* of services using *reliability patterns*. In our approach, a Web Service reliability certificate is a machine-readable representation of the reasons why a given service claims to be reliable, including the reliability mechanism the service relies on and the reliability pattern used to implement it.

As we shall see in the following, we use a POSA format for reliability patterns that provide a concise yet expressive way to specify how a given reliability mechanism is implemented by a service. Like other certification schemes, we envision that the party taking responsibility for a reliability claim will digitally sign the corresponding reliability certificate. Depending on the business scenario, the signing party can be the service supplier itself, or a third party accredited in the framework of a regulated certification process, as the ones adopted for life- or mission-critical applications [10]. Such certificates can then be used at service discovery time to make sure that the reliability requirements of a business process are met by all the services recruited to execute it.

A major difference between our reliability certificates and other types of Service Level Agreement-style metadata is runtime monitoring of reliability using reliability patterns [1,10,21]. Indeed, the SOA framework makes it possible to instrument Web services for monitoring the reliability mechanism implemented by the pattern mentioned in the certificate. At service discovery time, monitoring rules can be evaluated to check whether the certificate is still valid or not for a specific process execution, without requiring re-certification.

In the remainder of the paper we describe the building blocks of our proposal and how they can be put together to obtain a full scheme for the reliability certification of SOA Web services. Namely, in Section 2, we present the concept of pattern for improving system reliability. In Section 3, we describe how reliability patterns can be used to validate the reliability of services. In Section 4, we present our certification scheme, illustrating our machine-readable certificate format and a first solution to the integration of SOA and reliability certification. Finally, in Section 5, we present our related work and, in Section 6, we give our conclusions.

2 Using Patterns for Improving Reliability

The widespread diffusion of Web services and SOA is raising the interest for SOA-based implementation of life- and mission-critical applications for which reliability is a crucial requirement.

Often, in a traditional component-oriented scenario, reliability is added to the systems after the implementation phase, but experience has shown that this is not the most effective way to implement reliable services [13,20]. In this paper, we aim to define an approach for building reliable services that involves incorporating reliability in every phase of the system design and throughout

the entire software development life cycle [5,27]. There are five major reliability techniques that are used to handle failures and their consequences in a system [23] as follows.

- *Detection.* Detecting the occurrence of an error.
- *Diagnostic.* Locating the unit or component where the error has occurred.
- *Masking.* Masking errors so as to prevent malfunctioning of the system if a fault occurs.
- *Containment.* Confining or delimiting the effects of the error.
- *Recovery.* Reconfiguring the system to remove the faulty unit and erasing the effects of the error.

These five techniques can be implemented using the following reliability mechanisms:

1. *Redundancy.* The duplication of critical components in a system with the intention of increasing the reliability of the system. This mechanism is often applied to chemical, power, nuclear, and aerospace applications.
2. *Diversity.* Requires having several different implementations of software or hardware specifications, running in parallel to cope with errors or failures that could arise directly from a specific implementation or design.
3. *Graceful degradation.* This mechanism is essential in systems where, in the event of a failure, a system crash is highly unacceptable. Instead, some functionality should remain in the event of a failure. If the operating quality of the system decreases, the decrease should be proportional to the severity of the failure.
4. *Checking and monitoring.* Constant checking of the state of a system to ensure that specifications are being met is critical in detecting a fault. This mechanism, while very simple, plays a key role in obtaining a fault tolerant system.
5. *Containment.* Faults are contained within some specific execution domain, which prevents error propagation across system boundaries.

Challenges in software reliability not only stem from the size, complexity, difficulty, and novelty of software applications in various domains, but also relate to the knowledge, training, and experience of the software engineers involved. Our approach is based on the notion of reliability patterns [23,24]. A pattern is an encapsulated solution to recurrent software or system problems in a given context, and it can be described using UML diagrams [28]. Reliability patterns support widespread application of best practices and best solutions, and offer an effective guideline for software developers that may not have expert knowledge and experience in reliable system development. Specifically, a reliability pattern [23,24] consists of several parts which provide a detailed description of the patterns' objective, and serves as a tangible reference for an effective reliability solution. We utilize the POSA template (see Table 1) to describe reliability patterns, mainly because it provides a complete encapsulation of a solution to a given problem. Additionally the POSA template is a widely accepted format that sufficiently

Table 1. POSA template

Intent
Example
Context
Problem and forces
Solution
Implementation
Example resolved
Known uses
Consequences
Related patterns

describes a reliability pattern. Reliability patterns described using the POSA template provide class and sequence diagrams that can be used to help generate monitoring rules consistent with the reliability aspect being sought.

3 Certifying Services Built Using Reliability Patterns

Once a system is built using some methodology that uses reliability patterns, we need a way to show it has reached a given level of reliability. In a SOA environment we can go even further; we can certify that the services satisfy some standards of reliability making digitally signed information available at runtime, that is, in SOA-based business process enactment. In our approach, this information includes the reliability pattern used in the service implementation. Therefore, we start by looking at some patterns to ascertain their effect on the reliability of a service.

3.1 A Reliability Pattern Solution

A reliability pattern includes a section which describes its solution (see Figure 1). This section includes a class diagram illustrating the structure of the solution and showing the class functions, and their relationships. Figure 1(a) depicts an example of a class diagram for the well-known *Acknowledgment* pattern whose intent is "to detect errors in a system by acknowledging the reception of an input within a specified time interval". In the *Acknowledgment* pattern, the *Sender* in conjunction with the *Timer* constitute the *Monitoring System*, and the *Receiver* in conjunction with the *Acknowledger* entity constitute the *Monitored System*. In particular, the *Sender* is responsible for contacting the *Monitored System*. Whenever the *Sender* has sent the input to the *Receiver*, the *Timer*, that is responsible for counting down the timeout period every time an input is provided to the *Monitored System*, is activated. Upon receiving an input by the *Sender*, the *Receiver* notifies the *Acknowledger*. The *Acknowledger* is then responsible for sending an acknowledgment to the *Timer* for the received input. If the timeout period of the *Timer* expires for N consecutive times without

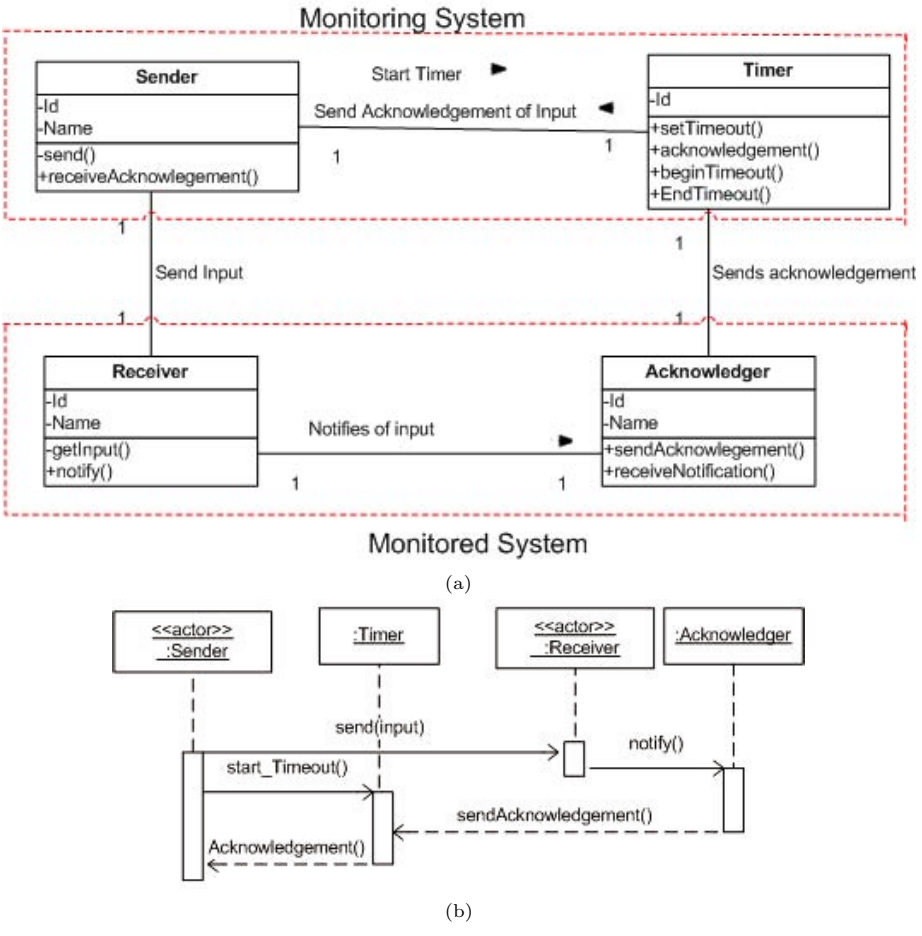

Fig. 1. Class diagram (a) and sequence diagram (b) for the Acknowledgment pattern

receiving an acknowledgment from the *Monitored System*, the *Timer* detects an error on the *Monitored System* and notifies the *Sender*.

The sequence diagram in Figure 1(b) provides the dynamics of one of the use cases of the pattern's solution. The two diagrams can be evaluated by transforming them to a formal representation and conducting some model checking to test that they perform some reliability function, that is, they avoid some failure.

3.2 A Priori Validation of Reliability Patterns

A priori validation of reliability patterns provides an estimation of the level of reliability that can be achieved before the actual system is implemented. A priori validation can be performed using the consequences and the failure/fault coverage of a reliability pattern.

The *consequences* of a reliability pattern describe the advantages and disadvantages of using the pattern. This information can be used a priori to compare patterns. A widely used criterion is the amount of computational resources required by the pattern.

The *failure/fault coverage* of a pattern, instead, is described as the number of failures that can be identified, masked, contained, or recovered with its use. This information can also be used for a priori selection of a pattern. For instance, the Dual Modular Redundancy (DMR) pattern can detect one fault but does not mask any faults; the Triple Modular Redundancy (TMR) pattern can detect two faults and mask one; the N-Modular Redundancy (NMR) pattern can detect (N-1) faults and mask (N-2) faults. Thus, the DMR pattern provides a lower level of reliability than the TMR pattern. Similarly, NMR provides a higher level of reliability than TMR. Figure 2 illustrates the structure of DMR, TMR, and NMR.

The evaluation of pattern consequences and coverage can permit to compare functionally equivalent services a priori (i.e., before they are invoked) on the basis of the level of reliability provided by the corresponding patterns. In this paper, however, we will focus on a posteriori reliability certificate checking.

3.3 A Posteriori Validation of Service Reliability

A posteriori validation of service reliability is an evaluation of the level of reliability provided by a given service implementation. We can assess the level of reliability in an implemented system that was built with the use of reliability patterns by evaluating different reliability metrics. There are many metrics in the literature [3,16,17,22,27,28] that can be used to calculate the reliability using data collected by monitoring an implemented system. We have selected some reliability metrics from the literature and proposed some of our own. Metrics (see Tables 2 and 3) are classified based on time- and cardinality-related failures. Such metrics correspond to the five reliability mechanisms discussed earlier in Section 2. Additionally, the metrics in Table 2 and Table 3 are easily measurable in a SOA environment, using available logging services and toolkits [7,18].

The metrics and related monitoring features can be used to enhance a priori comparison of services by taking into account the service implementation.

4 Machine Readable Certificates for Reliable Services

Current software certification schemes for security and dependability (e.g., Common Criteria [12]) provide human-readable, system-wide assurance certificates to be used at deployment and installation time. This type of certificates does not match the requirements introduced by a SOA in terms of runtime selection and composition of services. A solution providing machine-readable certificates is therefore required. Similarly to the definition of security property in [1], here we define a concrete specialization of the reliability property as the one that *provides enough information to support monitoring procedures aiming to establish*

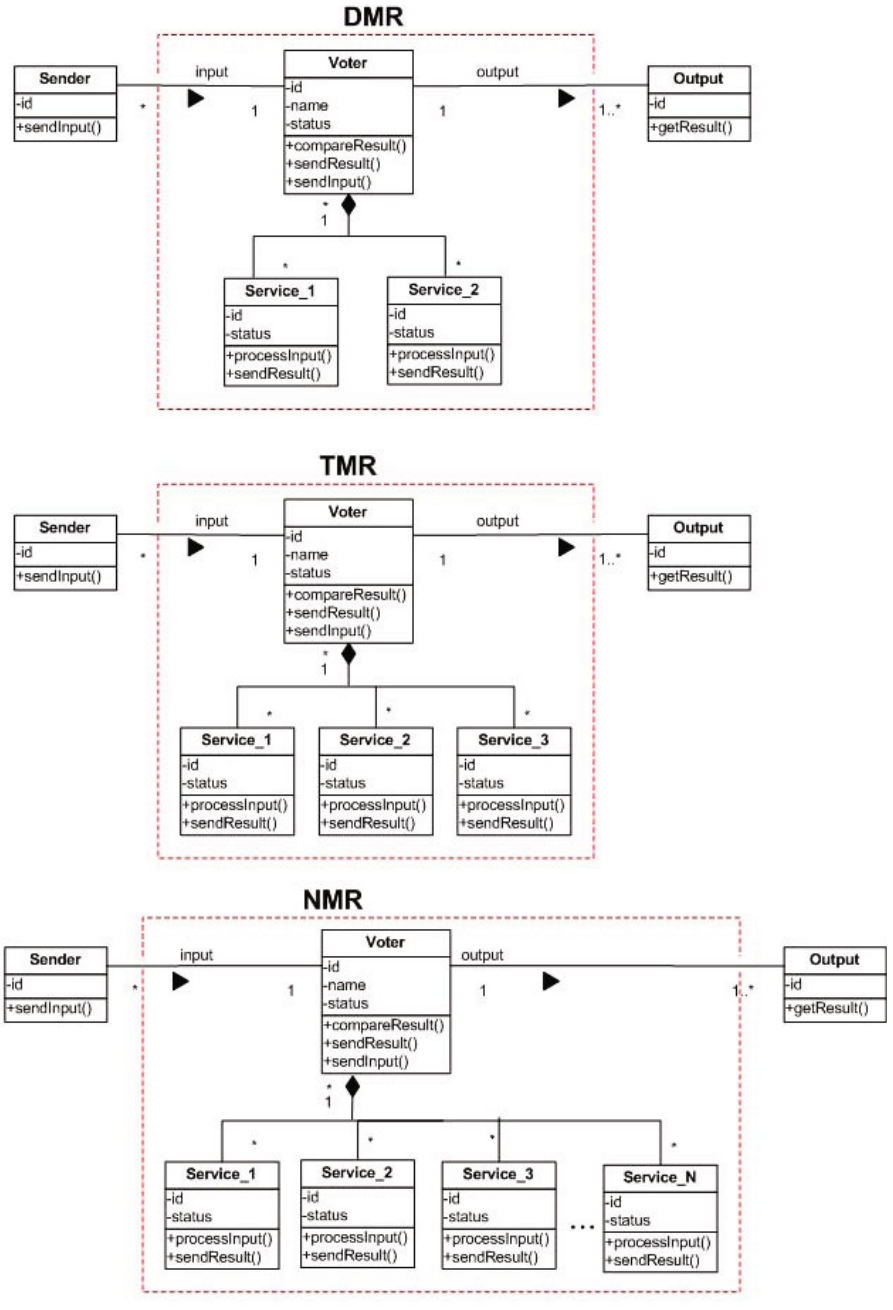

Fig. 2. Class diagram for DMR, TMR, and NMR patterns

Table 2. Time-related reliability metrics

Reliability Mechanism	Time-Related Metric	Description
Redundancy - Invokes one or more copy of the same mechanism	Time-to-Failure (TTF)	The time the service runs before failing
	Time-of-Failure (TF)	The time at which a failure occurs
	Mean Time to Fail (MTTF)	The average time it takes for the system to fail
	Failure Occurrence Rate (FOR)	The rate at which failures occur when the system is active
	Mean Time Between Failures (MTBF)	The average time before a failure occurs in the system
Diversity - Invokes one or more copy of a particular mechanism that performs the same function	Time-to-Failure (TTF)	The amount of time the service runs before failing
	Time-of-Failure (TF)	The time at which the failure occurred
Monitoring - Checks the system continuously to identify failures and sends alerts	Response-Time (RT)	The amount of time it takes to send an alert
	Time-to-Identify-Failure (TIF)	The time it takes to identify that a failure has occurred in the system
	Time-to-Failure (TTF)	The time the service runs before failing
Diagnosis - Identifies the source of failure	Investigation-Time (IT)	The time it takes to identify the unit that has failed
	Mean-time-to-Investigate (MTTA)	The average time it takes to investigate a failure
Masking - Hides the effects of a failure	Time-to-Replace-Failed-Component (TRFC)	The time it takes to replace a failed component
Containment - confines a failure to stop its propagation	Time-of-Failure-Arrest (TFA)	The time at which the failure was confined
	Time-to-Arrest-Failure (TAF)	Time it takes to confine the failure so that it does not propagate throughout the system
Recovery - Erases failure and restores normally	System-Recovery-Time (SRT)	The time needed for the system to recovery from a failure and return to a failure-free operational state
	Time-to-Recover (TTR)	The time needed to repair and restore service after a failure

if the property holds or not. In other words, a concrete definition of reliability specifies the mechanism in place to assert it (e.g., a redundancy mechanism) and the faults/failures the property is meant to support (e.g., loss of service failure). This information is represented as a set of *class attributes* specifying the mechanisms and/or the faults/failures. For instance, we can define a specialization of the *reliability* property whose class attributes are: *mechanism*=redundancy, *level*=4, *swapping time*=10ms, and *failure*=loss of service failure.

Besides the mechanisms and faults used to assert reliability, our machine-readable certificate includes all the other information in the reliability pattern used in service implementation and any available evidence supporting reliability. Our certificates are designed to be checked a posteriori, that is, on a working implementation of a service (Section 3). As we shall see, our evidence consists of a set of metrics that must be continuously monitored and updated using *monitoring rules*.

More in details, our machine-readable certificates for reliability include the following information:

Table 3. Cardinality-related reliability metrics

Reliability Mechanism	Cardinality-Related Metric	Description
Redundancy - Invokes one or more copy of the same mechanism	Number-of-Simultaneous-Failure (NSF)	The number of failures occurring at the same time
	Number-of-Invocation(NI)	Total number of calls made to a service
	Number-of-Failure (NF)	Total number of failures that occurred in the system
	Number-of-Expected-Failures (NEF)	The expected number of failures over a time interval
Diversity - Invokes one or more copy of a particular mechanism that performs the same function	Number-of-Simultaneous-Failure (NSF)	The number of failures occurring at the same time
	Number-of-Invocation(NI)	Total number of calls made to a service
	Number-of-Failure (NF)	Total number of failures that occurred in the system
	Number-of-Expected-Failures (NEF)	The expected number of failures over a specified time interval
Monitoring - Checks the system continuously to identify failures and sends alerts	Number-of-Failure-Alerts (NFA)	The total number of failure alerts sent
	Number-of-Successful-Acknowledgments (NSA)	The total number of successful acknowledgments sent
Diagnosis - Identifies the source of failure	Number-of-successful-Investigations (NSI)	The total number of times when the source of a failure is identified
	Number-of-Unsuccessful-Investigations (NUI)	The total number of times when the source of a failure is not identified
Masking - Hides the effects of a failure	Number-of-Failed-Components-Replaced (NFCR)	The total number of times a failed component is replaced
Containment - confines a failure to stop its propagation	Number-of-Confinement-Attempts (NUA)	The total number of times a confinement attempt is made
	Number-of-Resource-Needed-to-Contain-Failure (RNCF)	The percentage of system resources that was needed to prevent the failure from propagating throughout the system
	Number-of-Successful-Failure-Arrest (NSFA)	The total number of times a failure was detained
	Number-of-Unsuccessful-Failure-Arrest (NUFA)	The number of times a failure was not detained
Recovery - Erases failure and restores normally	Number-of-Successful-Recovery (NSR)	The total number of successful recoveries
	Number-of-Unsuccessful-Recovery (NUR)	The total number of failed or aborted recovery attempts
	Mean-time-to-Recover (MTTR)	The average time it takes to recover from a failure in the system

- *Reliability property*: a description of the concrete reliability property including class attributes with reference to mechanisms/faults used to assess it.
- *Reliability pattern*: a concise description of the reliability solution. We adopt the POSA template to describe reliability patterns.
- *Evidence*: a set of elements that specify the metrics and monitoring rules used for supporting the reliability in the certificate as follows.
  - *Set of metrics*: the metrics used to verify that a given property holds. For each metric, we define the expected value that is requested for the metric.
  - *Monitoring rules*: the rules used for monitoring the metrics in the evidence. Each rule contains a human-readable description and a reference to a standard toolkit for reliability monitoring on SOAs that permits to do the measurements of the corresponding metrics.[1] A violation of the monitoring rules produces a runtime revocation of the certificate.

Figure 3 shows our XML schema of the reliability certificate, which includes all information explained above. We note that the certificate contains the link to the certified service (*ServiceBinding* element), the reliability property (*Property* element), and the pattern used for the implementation (*Pattern* element). Then, it contains the evidence composed by a set of monitoring rules (*MonitoringRules* element) and a set of metrics (*Metrics* element). The *MonitoringRules* element includes a set of rule groups (*RuleGroup* element) that in turn contain a set of rules each one with an *ID* attribute. The *Metrics* element contains a set of metrics each one specifying the id of the rule to which the metric refers (*RuleID* attribute), the runtime and expected values for the metric, and an operator that specifies how to compare the two values. The runtime validity of the certificate is obtained by monitoring each rule in the evidence and comparing the metric runtime values with the expected values. This comparison can be simple (e.g., equality or inequality) or complex (e.g., including tolerance, bounding values).

When we come to the evaluation of the validity of the certificate for the service, we have to consider that all rules assume a boolean value at each time instant. A rule assumes value *true* if and only if *all* metric elements that refer to it are satisfied (i.e., the runtime value is compatible with the expected value). Rules in the *RuleGroup* element are then ANDed, while different *RuleGroup* elements are ORed, finally producing a boolean value for the *MonitoringRules* evidence. If it is *true*, the certificate is valid, otherwise it is revoked.

We now provide two examples of XML-based certificate for the service *Voter* (see Figure 2) implementing TMR and DMR patterns, respectively.

Example 1. Figure 4 shows an example of an XML-based certificate that proves the property *Reliability* for a *Voter* service available at the endpoint *http://www.acme.com/wsdl/voter.wsdl*. The Voter implementation follows the TMR pattern and includes a single *RuleGroup* element with two rules. The TMR requires software redundancy including at least three parallel instances of the

[1] Several commercial products are available, including the Microsoft Reliability Monitor [18].

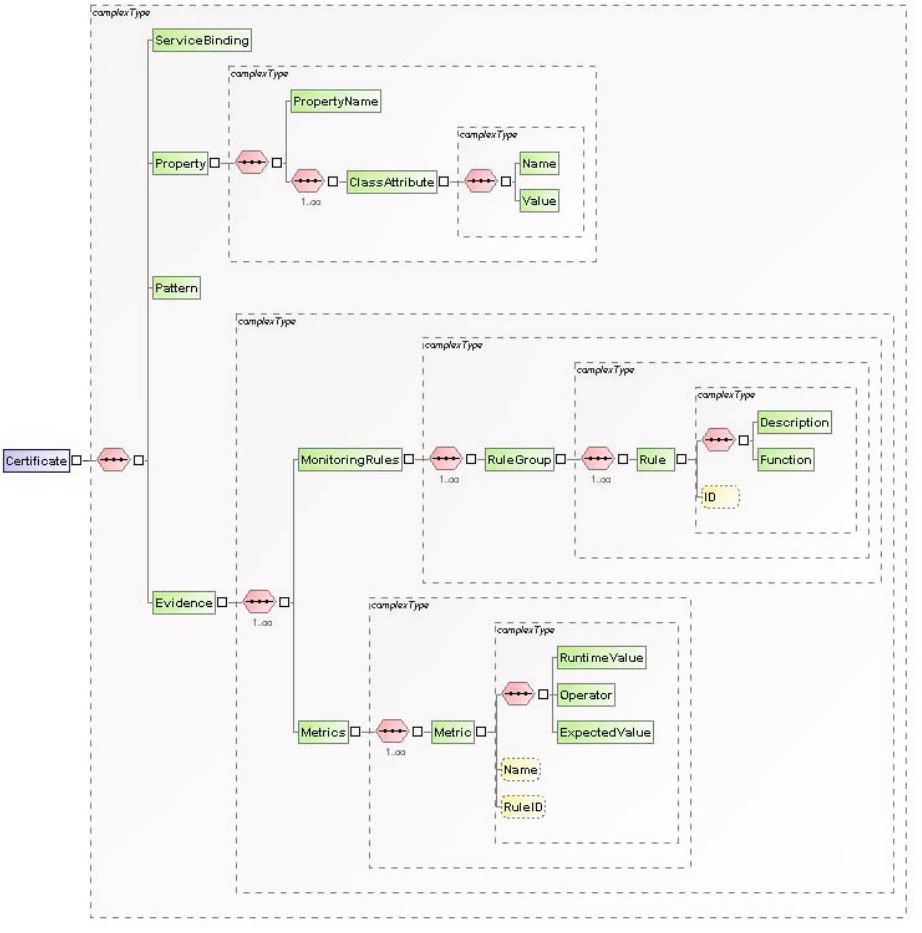

Fig. 3. XML schema of the reliability certificate

service. The first rule (rule 1) requires to continuously count all available service instances, using a toolkit function called *CountingRedundancy*(). The second rule (rule 2) requires that in case of an instance failure, the recovery time is measured. This measure is done via a function called *EvaluateRecoveryTime*(). The number of available instances and the recovery time are used in the *Number-of-Simultaneous-Failure* (*NSF*) and *Time-to-Recover* (*TTR*) metrics, respectively. In particular, the expected value for the number of available instances (or in other words the number of simultaneous failures the system can manage) is three, while the recovery time is 1 minute. The operators used in the comparison are both $\geq$. Since the runtime values of the metrics in the certificate are equal to/greater than the expected values, the certificate is valid.

Example 2. Figure 5 shows an example of XML-based certificate that does not prove the property *Reliability* for the *Voter* service available at the endpoint

```
<Certificate xsi:noNamespaceSchemaLocation="..." xmlns:xsi="...">
    <ServiceBinding>http://www.acme.com/wsdl/voter.wsdl</ServiceBinding>
    <Property>
        <PropertyName>http://www.acme.com/reliability/Reliability</PropertyName>
        <ClassAttribute>
            <Name>mechanism</Name>
            <Value>redundancy</Value>
        </ClassAttribute>
        <ClassAttribute>
            <Name>level</Name>
            <Value>3</Value>
        </ClassAttribute>
        <ClassAttribute>
            <Name>failure</Name>
            <Value>loss of service failure</Value>
        </ClassAttribute>
    </Property>
    <Pattern>TMR</Pattern>
    <Evidence>
        <MonitoringRules>
            <RuleGroup>
                <Rule ID="1">
                  <Description>Count all available service instances</Description>
                  <Function>CountingRedundancy()</Function>
                </Rule>
                <Rule ID="2">
                  <Description>
                    In case of an instance failure, measure the recovery time
                  </Description>
                  <Function>EvaluatingRecoveryTime()</Function>
                </Rule>
            </RuleGroup>
        </MonitoringRules>
        <Metrics>
            <Metric Name="NSF" RuleID="1" >
                <RuntimeValue>4</RuntimeValue>
                <Operator>greaterThan/equalTo</Operator>
                <ExpectedValue>3</ExpectedValue>
            </Metric>
            <Metric Name="TTR" RuleID="2">
                <RuntimeValue>1m</RuntimeValue>
                <Operator>greaterThan/equalTo</Operator>
                <ExpectedValue>1m</ExpectedValue>
            </Metric>
        </Metrics>
    </Evidence>
</Certificate>
```

Fig. 4. An example of valid certificate

http://www.acme.com/wsdl/voter.wsdl. This Voter implementation follows the DMR pattern. The monitoring rules and metrics are the same as the ones in Example 2, except for the expected value of *Number-of-Simultaneous-Failure* (*NSF*) metric that is equal to two. Since the runtime value of the redundancy metric is less than the expected value for the metric, the monitoring rule *rule 1* is not satisfied and the certificate is revoked.

A final aspect to consider is the integration of the reliability certification process and metadata within the SOA infrastructure. We need to provide a solution that allows clients to select the service that best fits their reliability requirements at runtime, on the basis of the information specified in the reliability certificate. This selection is performed by matching client requirements with service certificates.

4.1 An Architecture for Reliability Certificates Checking

Let us consider an enhanced SOA infrastructure composed by the following main parties. *i) Client* (*c*), the entity that needs to select or integrate a remote service based on its reliability requirements. *ii) Service provider* (*sp*), the entity implementing remote services accessed by *c*. *iii) Certification Authority* (*CA*),

```
<Certificate xsi:noNamespaceSchemaLocation="..." xmlns:xsi="...">
    <ServiceBinding>http://www.acme.com/wsdl/voter.wsdl</ServiceBinding>
    <Property>
        <PropertyName>http://www.acme.com/reliability/Reliability</PropertyName>
        <ClassAttribute>
            <Name>mechanism</Name>
            <Value>redundancy</Value>
        </ClassAttribute>
        <ClassAttribute>
            <Name>level</Name>
            <Value>4</Value>
        </ClassAttribute>
        <ClassAttribute>
            <Name>failure</Name>
            <Value>loss of service failure</Value>
        </ClassAttribute>
    </Property>
    <Pattern>TMR</Pattern>
    <Evidence>
        <MonitoringRules>
            <RuleGroup>
                <Rule ID="1">
                  <Description>Count all available service instances</Description>
                  <Function>CountingRedundancy()</Function>
                </Rule>
                <Rule ID="2">
                  <Description>
                    In case of an instance failure, measure the recovery time
                  </Description>
                  <Function>EvaluatingRecoveryTime()</Function>
                </Rule>
            </RuleGroup>
        </MonitoringRules>
        <Metrics>
            <Metric Name="NSF" RuleID="1" >
                <RuntimeValue>1</RuntimeValue>
                <Operator>greaterThan/equalTo</Operator>
                <ExpectedValue>2</ExpectedValue>
            </Metric>
            <Metric Name="TTR" RuleID="2">
                <RuntimeValue>1m</RuntimeValue>
                <Operator>greaterThan/equalTo</Operator>
                <ExpectedValue>1m</ExpectedValue>
            </Metric>
        </Metrics>
    </Evidence>
</Certificate>
```

Fig. 5. An example of revoked certificate

an entity trusted by one or more users to assign certificates. *iv) Evaluation Body* (*EB*), an independent, trusted component carrying out monitoring activities. *EB* is trusted by both *c* and *sp* to correctly check the certificate validity on the basis of the monitoring rules and metrics. *v) Service Discovery* (UDDI), a registry of services (e.g., [26]) enhanced with the support for reliability certificates and requirements.

Our service invocation process enhanced with reliability certification is composed by two main stages (Figure 6). In the first stage (Steps 1-2), *CA* grants a reliability certificate to a service provider *sp* based on a service implementation *s* and a reliability pattern. In the second stage (Steps 3-9), upon receiving the certificate for the service *s*, *sp* publishes the certificate together with the service interface in a service registry. Then the client *c* searches the registry and compares the reliability certificates of the available services. Once the client has chosen a certificate, it will ask to the trusted component *EB* to confirm its validity. *EB* checks that the corresponding monitoring rules hold and returns a result to *c*. If the result is positive *c* proceeds to call the service.

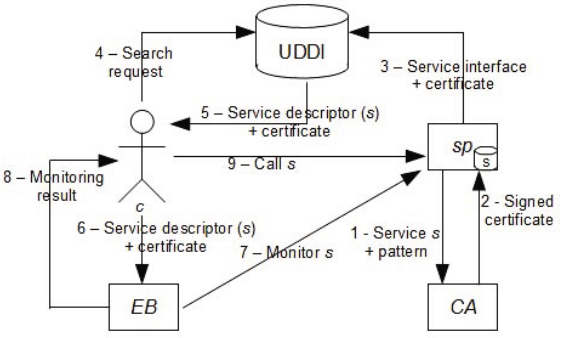

Fig. 6. A SOA enhanced with reliability certification

5 Related Work

Many approaches to software reliability have been proposed in the past and are presented in the following.

Becker et al. [2] proposed the Palladio Component Model (PCM) as a meta-model for the description of component-based software architectures with a special focus on the prediction of QoS attributes, especially performance and reliability. Spanoudakis et al. [25] proposed a tool called SERENITY which provides dynamic configuration and assembly of both security and dependability at runtime. Here, patterns are incorporated to realize the properties of security through location-based access control and dependability through monitoring and diagnostics. Monitoring and diagnostics are achieved with the use of a runtime framework called Everest. Everest employs event calculus to detect threats and failures in the system. The proposed solution also provides a monitoring framework for runtime checks of conditions related to the correct operation of security and dependability patterns. These conditions are specified as monitoring rules in Event Calculus.

Bernardi et al. [3] proposed an approach called MARTE-DAM for dependability modeling and analysis using UML. MARTE-DAM extends UML and includes features for the modeling, evaluation, and analysis of real-time systems. The authors first created a domain model that considers the main dependability concepts and organized them into top- and low-level packages. These packages include attribute descriptions and their relationships with each other. After the packages are defined they are stereotyped or mapped to a concrete profile. The packages usually contain dependability stereotypes and attributes called tags. They used a library of defined non-functional dependability types to measure different dependability metrics. They conducted a case study using a Message Redundancy Service (MRS) which is described using sequence diagrams. They then annotated the MRS with dependability properties using MARTE-DAM. They conducted an analysis and assessment of MARTE-DAM by transforming the MRS case into a Deterministic and Stochastic Petri Nets (DSPN). The DSPN

model was then simulated to measure its dependability and performance. Similarly the work by Koopman [15] proposed the Ballista approach to quantitatively assess fault tolerance, quality assurance and computer security. Ballista can be used for robustness testing in operating systems. The aim is to use Ballista as a quality check for software assurance, by measuring generic robustness for fault tolerance and transforming security aspects into analogs of Ballista data types. Ballista uses both software testing and fault injection to test robustness.

Lastly, Mustafiz et al. [20] proposed a model-based approach for developers to analyze the dependability of systems whose requirements are described as use cases, and to help identify reliable and safe ways of designing the interactions in a system. They introduced probabilities at each step of the use case; this probability represents the success of that step. They utilized a probabilistic extension of state charts which are deterministic finite state automata to formally model the interaction requirements defined in the use cases. AToM^3, a tool used for formal modeling, was used to model the state charts. An analytical evaluation of the formal model is then carried out based on the success and failure probabilities of events in the environment.

Several other approaches adopted a Markovian approach to evaluate system reliability (e.g., [14,19]). Most of the above approaches produce model-based reliability evidence. However, experience with security certification suggests that the integration in our reliability certification scheme is not unfeasible.

Other works focused on evaluating dependability and reliability of services in a SOA environment. Cardellini et al. [6] proposed a model-based approach to the realization of self-adaptable SOA systems with the goal of addressing dependability requirements. The authors introduced a SLA (System Level Agreement) Monitor for monitoring aspects of the services agreed in SLA. According to their definition of SLA, the proposed solution includes a large set of parameters for different kinds of functional and non-functional service attributes, as well as for different ways of measuring them. In our vision, this type of SLA monitoring can be responsible for doing the measurements of the monitoring rules specified in the service certificate. Cortellessa and Grassi [9] analyzed the problem of reliability in SOA environments with focus on the composition and on mathematical aspects of the reliability modeling. They define a generic set of information to support SOA reliability analysis that can be monitored also in the case of composed services. Grassi and Patella [11] presented a reliability evaluation algorithm for SOA services. The algorithm uses the flow graph associated with a service to compute its reliability. Finally, Challagulla et al. [8] proposed a solution based on AI reasoning techniques for assessing the reliability of SOA-based systems based on dynamic collection of failure data for services and random testing. Again, failure data could be integrated as evidence in our reliability certificates.

6 Conclusions

Reliability is a key concern in most systems today. The SOA paradigm, which supports runtime selection and composition of services, makes it difficult to

guarantee a priori the reliability of a process instance. In this paper, we presented a technique based on machine-readable reliability certificates using reliability pattern. In our work, we used the certificates to conduct a posteriori evaluation of reliable services. We are currently extending our approach to support a wider number of reliability patterns. Also, we are working on the integration of other types of dependability evidence.

Acknowledgements. This work was partly funded by the European Commission under the project ASSERT4SOA (contract n. FP7-257351). This material is based upon work supported by the National Science Foundation under Grant No. OISE-0730065.

References

1. Anisetti, M., Ardagna, C.A., Damiani, E.: Fine-grained modeling of web services for test-based security certification. In: Proc. of the 8th International Conference on Service Computing (SCC 2011), Washington, DC, USA (July 2011)
2. Becker, S., Koziolek, K., Reussner, R.: The palladio component model for model-driven performance prediction. Journal of Systems and Software (JSS) 82(1), 3–22 (2009)
3. Bernardi, S., Merseguer, J., Petriu, D.: A dependability profile within marte. Journal of Software and Systems Modeling 10(3), 313–336 (2009)
4. Buckley, I., Fernandez, E., Rossi, G., Sadjadi, M.: Web services reliability patterns. In: Proc. of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009), Boston, MA, USA (July 2009), short paper
5. Buckley, I., Fernandez, E.: Three patterns for fault tolerance. In: Proc. of the International Workshop OOPSLA MiniPLoP, Orlando, FL, USA (October 2009)
6. Cardellini, V., Casalicchio, E., Grassi, V., Presti, F.L., Mirandola, R.: Towards self-adaptation for dependable service-oriented systems. In: Proc. of the Workshop on Architecting Dependable Systems (WADS 2008), Anchorage, AK, USA (June 2009)
7. Challagulla, V., Bastani, F., Paul, R., Tsai, W., Yinong Chen, Y.: A machine learning-based reliability assessment model for critical software systems. In: Proc. of the 31st Annual International Computer Software and Applications Conference (COMPSAC), Beijing, China (July 2007)
8. Challagulla, V., Bastani, F., Yen, I.L.: High-confidence compositional reliability assessment of soa-based systems using machine learning techniques. In: Tsai, J., Yu, P. (eds.) Machine Learning in Cyber Trust: Reliability, Security, Privacy, pp. 279–322. Springer, Heidelberg (2009)
9. Cortellessa, V., Grassi, V.: Test and analysis of web services. In: Baresi, L. (ed.) Reliability Modeling and Analysis of Service-Oriented Architectures, vol. 154, pp. 339–362. Springer, Heidelberg (2007)
10. Damiani, E., Ardagna, C.A., El Ioini, N.: Open source systems security certification. Springer, New York (2009)
11. Grassi, V., Patella, S.: Reliability prediction for service-oriented computing environments. IEEE Internet Computing 10(3), 43–49 (2006)
12. Herrmann, D.: Using the Common Criteria for IT security evaluation. Auerbach Publications (2002)

13. Holzmann, G.J., Joshi, R.: Reliable Software Systems Design: Defect Prevention, Detection, and Containment. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 237–244. Springer, Heidelberg (2008)
14. Iyer, S., Nakayama, M., Gerbessiotis, A.: A markovian dependability model with cascading failures. IEEE Transactions on Computers 58(9), 1238–1249 (2009)
15. Koopman, P.: Toward a scalable method for quantifying aspects of fault tolerance, software assurance, and computer security. In: Proc. of the Computer Security, Dependability, and Assurance: From Needs to Solutions (CSDA 1998), York, U.K (July 1998)
16. Kopp, C.: System reliability and metrics of reliability, `http://www.ausairpower.net/Reliability-PHA.pdf` (accessed in date July 2011)
17. Lyu, M.: Handbook of Software Reliability Engineering. McGraw-Hill (1995)
18. Microsoft: Using Reliability Monitor, `http://technet.microsoft.com/en-us/library/cc722107(WS.10).aspx` (accessed in date July 2011)
19. Muppala, J., Malhotra, M., Trivedi, K.: Markov dependability models of complex systems: Analysis techniques. In: Ozekici, S. (ed.) Reliability and Maintenance of Complex Systems. NATO ASI Series F: Computer and Systems Sciences, vol. 154, pp. 442–486. Springer, Berlin (1996)
20. Mustafiz, S., Sun, X., Kienzle, J., Vangheluwe, H.: Model-driven assessment of system dependability. Journal of Software and Systems Modeling 7(4), 487–502 (2008)
21. O'Brien, L., Merson, P., Bass, L.: Quality attributes for service-oriented architectures. In: Proc. of the IEEE International Workshop on Systems Development in SOA Environments (SDSOA 2007), Minneapolis, MN, USA (June 2007)
22. Pan, J.: Software reliability.18-849b dependable embedded systems. Tech. rep., Carnegie Mellon University, `http://www.ece.cmu.edu/~koopman/des_s99/sw_reliability/` (accessed in date July 2011)
23. Saridakis, T.: A system of patterns for fault tolerance. In: Proc. of the EuroPLoP Conference, Kloster Irsee, Germany (2002)
24. Saridakis, T.: Design patterns for fault containment. In: Proc. of the EuroPLoP Conference, Kloster Irsee, Germany (2003)
25. Spanoudakis, G., Kloukinas, C., Mahbub, K.: The serenity runtime monitoring framework. In: Spanoudakis, G., Kokolakis, S. (eds.) Security and Dependability for Ambient Intelligence, pp. 213–238. Springer, Heidelberg (2009)
26. Tsai, W., Paul, R., Cao, Z., Yu, L., Saimi, A., Xiao, B.: Verification of Web services using an enhanced UDDI server. In: Proc. of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), Guadalajara, Mexico (January 2003)
27. Walter, M., Schneeweiss, W.: The modeling world of reliabiliy of reliability/safety engineering. LiLoLe Verlag (2005)
28. Walter, M., Trinitis, C.: Automatic generation of state-based dependability models: from availability to safety. In: Proc. of the 20th International Conference Architecture of Computing Systems (ARCS 2007), Zurich, Switzerland (March 2007)