12

Reliability Modeling and Analysis of Service-Oriented Architectures

Vittorio Cortellessa[1] and Vincenzo Grassi[2]

[1] Universita' dell'Aquila, cortelle@di.univaq.it
[2] Universita' di Roma Torvergata, vgrassi@info.uniroma2.it

Abstract. Service selection and composition are central activities in service-oriented computing, and the prediction of the QoS attributes of a Service-Oriented Architecture (SOAs) plays a key role to appropriately drive these activities. Software composition driven by QoS criteria (e.g., optimization of performance, maximization of reliability) has been mostly studied in the Component-Based Software Engineering domain, whereas methodological approaches are not well established in the service-oriented area. Indeed, prediction methodologies for service-oriented systems should be supported by automated and efficient tools to remain compliant with the requirement that most of the activities connected with service discovery and composition must be performed automatically. Moreover, the adopted implementation should respect the autonomy and independence of each provider of the services we want to include in our analysis. In this chapter we focus on the modeling and analysis of the *reliability* attribute in Service-Oriented Architectures, with particular emphasis on two aspects of this problem: (i) the mathematical foundations of reliability modeling of a Service-Oriented Architecture as a function of the reliability characteristics of its basic elements and (ii) the automatization of service composition driven by reliability criteria.

12.1 Introduction

Designing and building software systems by composition is one of the distinguishing features of the component-based and service-oriented approaches. Several methodologies and techniques have been proposed to drive the assembly of software systems from pre-existing components/services [16]. In particular, in the domain of software services, the automation of service discovery, selection and composition plays a key role to fully enable a service-oriented vision.

However, current proposals for the automated assembly of service-oriented systems are mostly based on criteria related to functional features, such as the minimal distance between the descriptions of required and offered services,

despite the high relevance that QoS attributes (such as performance and availability) may have in this type of systems.

In a *service market* vision, the delivered QoS plays an important role in determining the success of a service provider [6]. In this respect, an important issue is how to assess the QoS delivered by a service, for instance its performance or dependability characteristics. Delivering QoS on the Internet is by itself a critical and significant challenge, because of its dynamic and unpredictable nature. Assessing the delivered QoS of Service-Oriented Architectures (SOAs) is even more challenging, given the emphasis on dynamically binding service requests with the available services and resources in a given context.

QoS attributes are undeniably harder to take into account with respect to functional ones also because they relate to factors that rarely enter the software development process, such as the operational profile. Approaches have been introduced for service selection and assembly that also consider QoS attributes, but the selection is based, in best cases, on intelligent agents, in most other cases on empirical estimations and/or on the developers' experience, thus lacking model-based automated support [20]. It is also true that special skills are often required to model QoS attributes of software systems, due to the mathematical aspects and modeling intricacies that must be often faced in this domain.

In the context of SOA-based applications, the QoS assessment involves both *monitoring* the actual QoS experienced by a client, and *predicting* the QoS that could be experienced in some context. In particular, QoS prediction may play a crucial role either to drive the selection of services to be assembled to fulfill a given request, or to foresee potential QoS problems caused by changes in the application environment and to support corrective actions (e.g., re-binding to a different service provider). In other words, the ability of predicting QoS would allow to answer crucial questions on SOA, such as "What the reliability of my service-oriented architecture would be if I select this given set of services?" or "How would the performance of my architecture be affected from replacing a certain service with another one?"

A real breakthrough in this area would therefore be brought from the introduction of modeling techniques that allow to predict QoS properties of an SOA on the basis of features of the services involved, thus in practice extending the mechanisms successfully applied to functional aspects.

Modeling the overall QoS of an SOA-based application on the basis of properties of the component services may require very different efforts, depending on the QoS property we are interested in. In this respect, an interesting classification of QoS properties has been presented in [10] in the field of component-based software systems. Most of the ideas presented in [10] can be applied to the SOA domain.

However, a main issue for the QoS analysis of SOA is the parameter estimation. The properties of basic services are not easily made available from service providers. Thus, monitoring/estimation techniques are needed to "populate"

QoS models with values of basic service characteristics (such as the probability of failure). In Sect. 12.3.2 we shortly discuss this aspect.

In this chapter we focus on the modeling and the analysis of the *reliability* property of SOA, with a special emphasis on two aspects of this problem: (i) the mathematical foundations of reliability modeling of a service-oriented architecture as a function of the reliability characteristics of its basic elements and (ii) the automatization of service composition activity driven by reliability criteria.

According to the classification in [10], based on the capability of assembling a system-level model starting from characteristics of basic elements (such as components or services), the reliability is defined as an *usage-dependent attribute*, i.e., an attribute which is determined by the system usage profile. This means, in the SOA domain, that the service developers and assemblers must predict as far as possible the use of the service in different systems, which may not yet exist. A second problem is the transfer of the usage profile from the assembly (or from the system) to the service. Even if the usage profile on the assembly level is specified, the usage profile for the services is not easily determined especially when the assembly is not known.

The chapter is structured as follows: in Sect. 12.2 we shortly introduce the basic concepts of reliability theory; in Sect. 12.3 we discuss specific issues of reliability in SOA and review related work; in Sect. 12.4 we introduce a model for reliability of SOA along with an algorithm for its evaluation, then in Sect. 12.5 we propose an implementation of this algorithm complying with the SOA principles of decentralization and autonomy; in Sect. 12.6 we use the example described in the Introduction of this book to present an application of our reliability model, and finally in Sect. 12.7 we provide some conclusions.

12.2 Software Reliability Basics

Reliability is a specific aspect of the broader concept of dependability [3]. Other dependability aspects are, e.g., availability and safety. Reliability specifically refers to the continuity of the service delivered by a system. In this respect, two basic definitions of reliability can be found in the literature: (i) the probability that the system performs its required functions under stated conditions for a specified period of time [19] and (ii) the probability that the system successfully completes its task when it is invoked (also known as "reliability on demand") [12].

The definition in (i) refers in particular to "never ending" systems that must operate correctly over all the duration time of a given mission (e.g., the on-board flight control system of an airplane that should not fail during the enitre duration of a flight). The definition in (ii) refers to systems offering services that, once invoked, must be successfully completed.

Both definitions can be applied to systems at whatever level of granularity (e.g., a distributed software system, a software component, a software service,

etc.), whose correctness can be unambiguously specified. A correct behavior is intended here as a "failure-free" one, where the system produces the expected output for each input following the system specifications.[3]

However, "failure-free" only refers to what can be observed at the system output level. A system may in fact experience a certain degree of incorrectness without showing any failure. This assertion is easily understandable on the basis of three fundamental reliability concepts: *fault*, *error*, and *failure*. A fault is a wrong statement introduced somewhere in the software.[4] An error is an unexpected state in which a system may enter upon executing a fault (e.g., an internal variable assumes an unexpected value). A failure occurs when an error propagates up to the system output (e.g., an output variable assumes an unexpected value).

The presence of a fault in a system does not necessarily imply that the system eventually experiences an error. In fact, a wrong statement might not be executed, even in a very long interval of time, due to the structure of the code and the sequence of inputs given. And, even if executed, the wrong statement could not originate any unexpected value of the internal variables.[5] Moreover, a system in an erroneous state does not necessarily manifest a failure, because the error may be masked from the operations that are executed between the erroneous state and the output production.

Failures can be classified with respect to different attributes. With respect to the way a failure manifests itself, they can be partitioned as follows:

- *Regular failure*—A failure that manifests itself as an unexpected value of any system output.
- *Crash failure*—A failure that immediately brings the system to stop its elaboration; systems that only account for this type of failure are also known as *fail-and-stop* systems.
- *Looping failure*—A failure that prevents the system to produce any (correct or incorrect) output; this type of failure is particularly problematic because it may take some time to assess that the system has failed under such a failure.

With respect to their severity, failures can be partitioned as follows:

- *Repairable failure*—A failure that can be somehow repaired without restarting the whole system.
- *Unrepairable failure*—A failure that requires the system to be restarted to restore its correct behavior.

[3] We assume readers are familiar with basics of reliability theory; however, we redirect those interested to details on this topic to [19].

[4] For sake of tractability, we consider in this chapter only software faults, even though the reliability of SOAs may be affected by hardware faults as well.

[5] For example, both statements $y = x * 2$ and $y = x^2$ produce the result $y = 4$ when $x = 2$.

Each reliability model undergoes several hypotheses on the types of failures that the modeled system can experience. Obviously, the less restrictive the hypotheses are the more complicated is the model formulation. Note, however, that the attributes specified above are not independent of each other, e.g., a crash failure cannot be repairable.

12.3 Specific Reliability Issues in SOA

Reliability models of modular software systems aim at formulating the reliability of the whole system as a function of the reliabilities of the basic elements. This idea is behind models for object-oriented, component-based, service-based systems and, in general, any system that can be viewed as an assembly of basic elements.

In this section we discuss specific issues for the modeling and estimation of the service reliability in SOA-based systems.

First of all, we note that the definition (ii) of reliability on demand given in Sect. 12.2 appears more suitable than definition (i) within the SOA domain, since it finely matches with the expectation and the degree of trustworthiness a user may have about a service. On the basis of this definition of reliability, in the following subsections we discuss, respectively, of

- the additional information that must be provided to support reliability analysis of SOAs;
- the estimation of this information in an SOA environment;
- the viewpoint that can be assumed in the SOA reliability analysis (i.e., client vs provider viewpoint);
- the failure model that we adopt in the modeling of SOA reliability.

In the last subsection of this section, we review existing work that is related to this chapter topic. Given the lack of specific reliability models for service-based systems, we take a wider view on software systems that are built by assembling basic elements, such as components.

12.3.1 Information to Support SOA Reliability Analysis

In an SOA environment, services are expected to publish information needed to correctly invoke them over the network. This information, expressed by a suitable language like WSDL [30], includes the name of the provided operations, and the name and type of their input and output parameters. To support predictive analysis of some QoS attribute like the service reliability, each service must also publish QoS-related information.

This raises the question about which information should be published to better support QoS predictive analysis. In this perspective, it is important to note that a basic principle of the SOA paradigm is that each composition of

services may become itself a service that can be recursively used to build other services. As a consequence, it is useful to distinguish two kinds of service:

1. *Atomic service* that does not require any other service or resource to carry out its own task; this includes, e.g., not only the services offered by basic processing and communication resources, but also "self-contained" software services strictly tied to a particular computing environment and that cannot be re-deployed;
2. *Composite service*, realized as a composition of other selected services that it requires to carry out its own tasks; the glue logic of this composition may be expressed using workflow description languages like BPEL [31].

From the reliability prediction viewpoint (but the same consideration holds for other QoS attributes), the basic difference between these two kinds of services is that the provider of an atomic service can publish complete reliability information that can be directly used by the clients to figure out the service reliability, while the provider of a composite service is only aware of reliability information concerning the part of the service implementation which is under his/her direct control (we call it as the *service internal segment*). This information must be combined with the reliabilities of the other (dynamically) selected services to get the overall service reliability.

In order to properly combine these reliabilities, attention must be paid to give the right weight to the reliability of each single service: a rarely invoked service has obviously a smaller impact on the reliability of the invoking service than a frequently used one. Hence, besides knowing which services are required by a composite service, we must also take into account how they are used (i.e., we must know the service operational profile).

Therefore, to support the reliability prediction of a service composition, we need the following information on each service[6]:

- *Internal reliability* (both atomic and composite services), i.e., a reliability measure that expresses the probability of successfully completing some task considering only the internal segment of the service; in the case of an atomic service it corresponds to the actual service reliability, whereas in the case of a composite service it must be suitably combined with the reliabilities of the invoked services;
- *Service usage profile* (composite services only), i.e., a description of the pattern of external service requests expressed in a stochastic form (also known as operational profile); e.g., if a certain service may invoke two alternative services for completing its task (depending on the user inputs) then the probability of each service invocation is an element of the service usage profile.

[6] In Sect. 12.5 we present an architecture in which providers have three alternative approaches to disclose this information.

12.3.2 Estimation of Additional Information

In an ideal scenario, the internal reliability of a service shall be associated to the service description at the time the provider publishes the service on a registry. On the contrary, the service usage profile is a very domain-dependent information, therefore it cannot be a priori estimated.

However, in a more realistic scenario, the information described in Sect. 12.3.1 can be estimated by monitoring the service activity.

In particular, with regard to the service usage profile, the structure of the composite service workflow (expressed with a service composition language like BPEL) provides information about the possible invocation patterns of external services. To estimate the probability of different patterns, we must basically monitor the relative frequencies of the different branches at each workflow branching point, and collect such data over an adequate number of different invocations of the composite service.

On the other hand, the internal reliability can be estimated as the ratio between the number of service invocations and the number of failures that occur. We point out that, in the case of a composite service, the failures that should be recorded at a composite service site are those generated by the internal segment of the service. Collecting failure statistics about the used external services could not be significant, as at different time instants we could bind to different implementations of the same abstract service and, given the autonomy principle of the SOA environment, we are not generally aware of these changes.

However, methods for the estimation of the probability of failure of software components and the usage profile (in component-based systems) have been reviewed in [12], and are extensively discussed in [11].

12.3.3 Client vs Provider Viewpoint of Reliability

We may adopt two different perspectives in the assessment of the reliability of a service in an SOA framework, depending on whether we look at services from the client or provider viewpoint. This is generally not an issue in traditional distributed systems, where the network and other environment components are under the control of a single organization. On the contrary, in an SOA environment, the reliability information published by a service provider is likely to concern only what can be observed at the service site and does not include information about the reliability of the network infrastructure used to access the service, which is generally out of the provider control. On the other hand, from the perspective of the client of a service, this information must be integrated into the overall reliability assessment procedure, as the used network infrastructure may greatly affect the reliability perceived by the user. Neglecting this information could lead to poor predictions about the overall reliability of a service. We point out,

however, that all the above considerations can be extended to other QoS attributes.

In some cases, a reference to the network used to access a remote service could be explicitly expressed in the workflow of a composite service, e.g., when the latter explicitly intends to use networking functionalities offered through some service-oriented interface (like in OSA/Parlay [29]) by a network service provider. This could facilitate the inclusion of network reliability information in the overall reliability model, assuming that a network service publishes its reliability information like any other service. If the network services needed to access remote services are not explicitly mentioned in a composite service workflow, their use should be made explicit at the reliability modeling level. However, the assignment of a meaningful reliability value to the network services could be more problematic in this case, as it could not be clear at the composite service level which kind of network is going to be used (possibly some kind of "best effort" network).

However, if the provider does not publish any data about the service reliability then the client can refer to trusted third-parties to collect this information [5].

12.3.4 Failure Models for SOA

As reported in Sect. 12.2, failures can be classified as regular, crash, and looping failures according to the way they manifest themselves, while they can be classified as repairable or non-repairable failures according to their severity.

Crash failures are the simplest ones to model from the reliability viewpoint, as they lead to the complete system failure as soon as they occur. In this respect, it is worth noting that it has been argued, based on an analysis of existing systems, that components and services for Internet-based systems should be designed to be "crash-only" [7].

On the other hand, regular failures causes the generation of incorrect output values. As discussed in Sect. 12.2, the incorrect output generated by an inner service invoked within the workflow of a composite service does not generally imply the overall failure of the composite service itself. This is due to the possibility that the inner service failure does not propagate up to the composite service outputs because some other service on the path to the output is able to mask the error. Hence, to include regular failures in the reliability analysis of SOA-based systems would require to take into account the *error maskability* factor, i.e., the capability for a service to map an incorrect input to a correct output. This factor can be ignored by assuming that any regular failure occurring in an inner service always propagates to the composite service outputs. This simplifies the analysis but could lead to overly pessimistic estimations of the overall reliability.

With regard to the failure repairability we note that, given the definition of reliability on demand that we have adopted, repairable failures are not

actually a concern. Indeed, a repairable failure does not prevent, by definition, the correct termination of a service, and hence does not affect its reliability on demand. Nevertheless, it may affect other quality attributes like the service performance (because the repair leads to a "degraded" mode of operation) or availability (because of service interruption during the repair).

Finally, we point out that many existing reliability analysis methodologies for service- or component-based systems rely on the assumption of independence among the system components. When applying these methodologies in an SOA framework we must be careful about the validity of such an assumption. Indeed, it may happen that originally independent services are assembled in such a way that they exploit some common service, so becoming no longer independent. However, considering the impact of service sharing on reliability is not an easy task.

12.3.5 Related Work

The scientific literature has produced several interesting approaches to the modeling of reliability in modular software systems based on characteristics of modular units. Most of them can be somehow adapted to the case of service-based systems, but the adaptation may bring to loose peculiarities of SOA like the ones discussed in the previous subsections. Due to the lack of specific approaches for SOA, in this subsection we briefly present the major contributions in the wider field of modular software systems.

Hence, the originality of this chapter with respect to the existing work is to build a reliability model for SOA that takes into account the specifics of service-based systems.

A thorough review of reliability modeling in the field of software architectures can be found in [12], where architectural models are partitioned as follows: (i) *path-based models*, where the reliability of an assembly of components is calculated starting from the reliability of architectural paths; (ii) *state-based models*, where the reliability is calculated starting from the reliability of system states and from the transition probabilities among states.[7]

One of the main differences between these two types of models emerges when the control flow graph of the application contains loops. State-based models analytically account for the infinite number of paths that might exist due to loops. Path-based models require instead an explicit enumeration of the considered paths; hence, to avoid an infinite enumeration, the number of paths must be somehow restricted, e.g., to the ones observed experimentally during the testing phase or by limiting the depth traversal of each path. In this respect, the methodology we propose in Sect. 12.4 adopts a state-based model.

As said in Sect. 12.3.4, the impact of service sharing on SOA reliability may be consistent, even though modeling this aspect is not an easy task. In fact, it

[7] Quite often state transitions are triggered by the control flow between system components.

falls under the more general problem of modeling error propagation in modular software systems [1]. Most of the existing models do not consider the impact of error propagation on the estimation of the system reliability due to the extremely high complexity of finding closed-form formulations to the problem.

Models for the reliability estimation of a component-based system embedding the error propagation and the error maskability factors have been recently presented in [14] and [22], which are based on quite different failure models. In [14] it is assumed that an error arising within a component does not cause an immediate failure, but it can rather propagate to other components up to the system output, unless it is masked before reaching the output. On the other hand, in [22] it is assumed that each error arising within a component immediately causes a system failure and, at the same time, it can also propagate to other components affecting their failure probability. This latter failure model, based on the contemporary assumption of immediate failure and propagation to other components, deserves in our opinion further investigation about its significance.

Quite interesting work has been done in other topics somehow related to the SOA reliability. In particular, we provide several seminal references for readers interested to the following topics: ontologies for QoS [21, 23, 27], representing QoS in UML [28], monitoring QoS in SOA [4], and Service Level Agreement in SOA [17, 18].

12.4 A Model for Predicting the Reliability of SOAs

Any reliability prediction methodology for SOA-based applications must be compliant with the specific constraints and requirements of this environment. In particular, this means that prediction methodologies must be implemented by automated and efficient tools to remain compliant with the requirement that most of the activities connected with service discovery and composition must be performed automatically. Moreover, the implementation of a methodology should meet the openness and distribution characteristics of SOAs. This implies that it should respect the autonomy of each provider of the services involved in the reliability prediction.

In order to address these automation and efficiency issues, in this section we tackle them by proposing an algorithmic reliability analysis methodology. Assuming that, in general, an offered service is built as a composition of other services, the methodology is based on the service assembly structure and exploits reliability information published by each assembled service in its description. In Sect. 12.5, we propose an architecture for the implementation of this methodology that supports different degrees of autonomy among the providers of the services involved in a composition. In this methodology, we take into account all the SOA specific issues outlined in Sect. 12.3.

The failure model we adopt is the "fail-stop with no repair" model. Hence, we only consider crash failures that occurs within a service component and

lead to the service interruption. The methodology can be applied as well to regular failures that does not cause service interruption, under the hypothesis that each error generated by a service always propagates up to the system output.

We do not consider repairs because, as discussed in Sect. 12.3, they are basically not relevant in the reliability on demand analysis. However, this model does not imply that we are assuming repairs never occur within the system. Simply, we are restricting our attention to those failures that cannot be repaired, as our focus is on the system reliability.

A key element of this methodology is the definition of a suitable model for the information associated with each service that concerns its internal reliability, and, in case of a composite service, the pattern of requests addressed to other services. We use a unique model to represent both these types of information, and assume that the QoS-related information of a service is published by means of an instance of this model.

12.4.1 A Model Based on Internal Reliability and Service Usage Profile

The model is based on a probabilistic flow graph, where each node of the graph models a "stage" of the service execution that must be completed before a transition to the next node can take place. Each stage may include the request for one or more external services. The flow graph includes two special nodes, a *Start* node that represents its entry point and an *End* node with no outgoing transitions, representing the successful completion of the service. This flow graph can be considered as a representation "distilled" from some description of the service workflow (e.g., expressed using a language like BPEL), and enriched with statistical information needed to support reliability prediction.

Transitions from node to node of the flow graph follow the Markov property, where $p(i, j)$ denotes the probability that stage j is selected after the completion of stage i. This kind of transition rule basically models (in a probabilistic way) a sequential flow of control. We introduce other kinds of control flows in our model by allowing more than one external service request to be specified within each node. In this case, before a transition to the next stage can take place, the service requests associated with node i must be completed according to a specified completion model. In the current version of our methodology, we consider two possible completion models:

1. *AND* model – all the service requests included in node i must be completed to enable a transition to the next stage.
2. *OR* model – at least one of the service requests included in node i must be completed to enable a transition to the next stage.

The *AND* model allows to represent a request for the parallel execution of a set of services, as expressed, e.g., "by the flow" control construct of BPEL

(corresponding to a *fork-join* execution pattern) [31]. The *OR* model allows to represent a race among different service requests, as expressed, e.g., "by the pick" control construct of BPEL (where one out of several activities is non-deterministically selected) [31]. It can also be used to model the presence of fault-tolerance features, where different instances of a service are tried until at least one of them succeeds. We are planning to include in our methodology other completion models (e.g., "k out of n").

Besides the pattern of requests addressed to other services, we also embed in this flow graph information about the internal reliability of a service, by associating with each node i of the graph a failure probability $intf(i)$, that represents the probability of a failure occurrence during the execution of that stage. This probability concerns only the internal segment of the service described by the flow graph. In general, $intf(i)$ may be expressed as a real valued function of suitable parameters (e.g., the probability of a failure occurrence when a processing service is invoked may depend on the number of operations to be processed). The *Start* and *End* nodes have zero failure probability, as they do not correspond to any real activity.

This flow graph can be used to model information on the reliability of atomic as well as composite services. As the example in Fig. 12.1a shows, an atomic service can be modeled by a flow graph consisting of only one node (besides the Start and End nodes), which does not contain any request for external services. Thus, the single node of this graph only reports an $intf$ value. In particular, the flow graph in Fig. 12.1a models the reliability characteristics of a computing resource that offers a processing service. Figure 12.1b depicts an example of flow graph for a composite service that includes

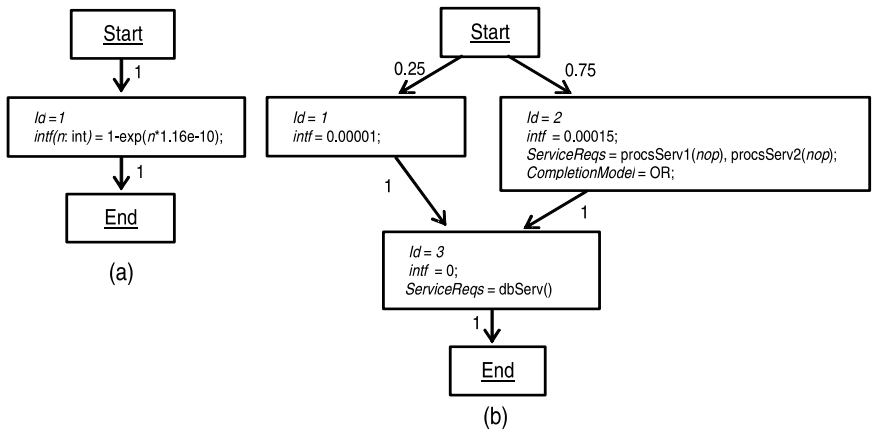

Fig. 12.1. Representation of the internal failure probability and service usage profile of a service by a probabilistic flow graph: (**a**) flow graph of an atomic service (processing service with exponential failure rate depending on the number n of operations to be processed), (**b**) flow graph of a composite service (at node 2, *nop* is the number of operations whose execution is requested to the two processing services)

requests for external services. It consists of two alternative stages 1 and 2, followed by a final stage 3. At stage 1, no external service is requested: this stage may model the execution of "locally implemented" operations, and the corresponding value of $intf(1)$ models their failure probability (i.e., 0.00001). On the other hand, at stage 2 two external processing services are requested, with an *OR* completion model. This stage may model the request for the remote execution of some code provided by the service, where the value $intf(2)$ models the intrinsic failure probability of that code (i.e., 0.00015) that must be combined with the failure probabilities of the selected processing services to get the overall reliability. The *OR* completion model of this stage models the use of a fault-tolerant approach in its design, since it is sufficient that only one of the two service requests succeeds to make the invoking service able to continue its execution. Finally, stage 3 models the request for another external service (i.e., a database service in this example).

12.4.2 An Algorithm for Model Evaluation

Given this model, we can now present our evaluation methodology, where we adopt a client-side perspective which means, as discussed in Sect. 12.3, that we include in the reliability evaluation of a service also the reliability of the network used by the client to access the service.

For this purpose, let us introduce the following notation:

- $Rel_{cli}(S)$ – the probability that a service S is able to complete its task, as seen by a client invoking S.
- $Rel_{pro}(S)$ – the probability that a service S is able to complete its task, as seen by the provider of S.
- $Rel_{net}(S)$ – the reliability of the network used to access a service S, when it is invoked by a client.
- $P_{fail}(i)$ – the probability of a failure occurrence before the completion of stage i of a given service.
- $p^*_S(Start, End)$ – probability of reaching, in any number of steps, the *End* state of the flow graph associated with S, starting from its *Start* state.[8]

Given this notation, the client-side reliability of a service S can be expressed as follows, once its provider-side reliability and the reliability of the network used to access S are known:

$$Rel_{cli}(S) = Rel_{net}(S) \cdot Rel_{pro}(S) \tag{12.1}$$

Using the flow graph model defined above, we can calculate $Rel_{pro}(S)$ in (12.1) as follows:

$$Rel_{pro}(S) = p^*_S(Start, End) \tag{12.2}$$

[8] We remind that, under the fail-and-stop assumption, reaching the *End* node means that there have been no failures in the execution path.

$p^*_S(Start, End)$ can be calculated by standard results from the Markov processes theory [24]. However, to get a meaningful result, we must elaborate transition probabilities on the flow graph before calculating $p^*_S(Start, End)$. Indeed, each transition probability $p(i, j)$ associated with an outgoing arc from a node i is an information about how frequently a stage j is executed after a stage i. For reliability prediction purposes, this information must be weighed by the probability $(1-P_{fail}(i))$ that no failure occurs before the completion of stage i. Hence, to calculate expression (12.2), we must first calculate $P_{fail}(i)$ for each stage i of the flow graph of S.

The calculation of $P_{fail}(i)$ is immediate when i does not contain any request for external services. This is the case of a single-stage flow graph associated with an atomic service, whose $Rel_{pro}(S)$ (given by (12.2)) can be immediately calculated. In all other cases we enter a recursive process, as to calculate $P_{fail}(i)$ we must first calculate the reliability of all the services S_k requested by S at stage i. In this respect, we point out that S is the "client" of the services S_k. Hence, the reliability of the generic S_k to be used in the evaluation of $P_{fail}(i)$ is $Rel_{cli}(S_k)$, which can be calculated using expressions (12.1) and (12.2).

Once the $Rel_{cli}(S_k)$ reliabilities are known, we can combine them with $intf(i)$, according to the completion model of i, to calculate the overall $P_{fail}(i)$. By adapting the results that we have presented in [13], $P_{fail}(i)$ can be expressed as follows for the *OR* completion model:

$$\begin{aligned} P_{fail}(i) = & \qquad (12.3)\\ &= 1-(1-intf(i))(1-\prod_{S_k}(1-Rel_{cli}(S_k)))\\ &= 1-(1-intf(i))(1-\prod_{S_k}(1-Rel_{net}(S_k)\cdot Rel_{pro}(S_k)))\\ &= 1-(1-intf(i))(1-\prod_{S_k}(1-Rel_{net}(S_k)\cdot p^*_{S_k}(Start, End))) \end{aligned}$$

For *AND* completion model we instead have

$$\begin{aligned} P_{fail}(i) = & \qquad (12.4)\\ &= 1-(1-intf(i))\prod_{S_k} Rel_{cli}(S_k)\\ &= 1-(1-intf(i))\prod_{S_k} Rel_{net}(S_k)\cdot Rel_{pro}(S_k)\\ &= 1-(1-intf(i))\prod_{S_k} Rel_{net}(S_k)\cdot p^*_{S_k}(Start, End) \end{aligned}$$

Expressions (12.3) and (12.4) hold under the assumption that all the S_k's requested at stage i are independent. As pointed out in Sect. 12.3, this assumption could not hold in an SOA environment. However, taking into account all

the possible inter-dependencies is really challenging. In [13] we have analyzed a restricted dependency scenario, where all the service requests at stage i are actually invocations of the same service S' offered by a single resource. This scenario occurs, e.g., when we allocate n software components to the same processing resource, thus requesting the same processing service offered by that resource. Under this scenario, we have shown in [13] that expression (12.4) still holds, with $\prod_{S_k} Rel_{net}(S_k) \cdot p^*_{S_k}(Start, End) = (Rel_{net}(S') \cdot p^*_{S'}(Start, End))^n$, where n is the number of invocations of S'. On the contrary, expression (12.3) is no longer valid, and must be substituted by the following expression:

$$P_{fail}(i) = 1 - (1 - intf(i)) Rel_{net}(S') \cdot p^*_{S'}(Start, End) \tag{12.5}$$

The intuition behind (12.5) follows from the stopping failure and no repair assumptions. If all the S_k's invocations are actually invocations of the same service, then its failure prevents the possibility of trying other alternatives. Hence, the "OR-reliability" of n invocations of S' is equal to the simple reliability of S'. Using (12.3) instead of (12.5) when this scenario occurs would lead to an underestimation of the service reliability.

The recursive Algorithm 1 summarizes all the operations described above. Given the flow graph *fg(S)* associated with a service S, the algorithm returns the provider-side reliability of S.

Algorithm 1 Model evaluation algorithm: **double** Rel-pro-Alg($fg(S)$)

for each node i in $fg(S)$ (except the *Start* and *End* nodes) **do**
  if(i does not include any request for external services)
  then $P_{fail}(i) = intf(i)$
  else
    for each S_k requested in i **do**
      net_k = reliability of the network used to invoke S_k
      get $fg(S_k)$
      $r_k = net_k \cdot$ Rel-pro-Alg($fg(S)$) //recursive step
    endfor
    case *CompletionModel:*
      OR : $P_{fail}(i) = 1 - (1 - intf(i))(1 - \prod_k (1 - r_k))$ // expression (12.3)
      AND : $P_{fail}(i) = 1 - (1 - intf(i)) \prod_k r_k$ // expression (12.4)
    endcase
  endif
  for each outgoing transition from i to j with probability $p(i, j)$ **do**
    replace $p(i, j)$ with $(1 - P_{fail}(i)) \cdot p(i, j)$
  endfor
endfor
return absorption probability in the *End* node of the discrete time Markov process described by $fg(S)$, calculated using standard Markov process solution techniques

Step 11 of this algorithm relies on the independence assumption discussed above. If this assumption does not hold and the conditions of the "single service sharing" scenario occurs, then step 11 must be modified according to expression (12.5). Step 8 is the recursive step. The recursive call returns the provider-side reliability of S_k. To turn it into the client-side reliability (as it is perceived at the S site), this reliability must be multiplied by the reliability of the network used to access S_k. The bottom of the recursion is reached when S is an atomic service. In this case, steps 5–13 are skipped, and the calculation of step 19 is greatly simplified as *fg(S)* consists of only one node, plus the *Start* and *End* nodes.

12.5 Analyzing Reliability in the SOA Framework

In this section we discuss issues concerning the implementation, in an SOA environment, of the methodology presented in Sect. 12.4.

The methodology relies on the assumption that each provider of a composite service collects and publishes information concerning the service internal structure that consists of (i) the external services it exploits, (ii) how they are glued together, and (iii) how frequently they are invoked. In Sect. 12.4 we have presented a data structure (i.e., a flow graph) which able to represent all this information. We remark here that the construction of such a data structure can be completely automated. Indeed, with regard to the flow graph structure (i.e., nodes and edges that connect them), it can be easily extracted from the executable workflow description of a composite service. For example, if the workflow is expressed in BPEL, which is an XML-based language, we can use XML navigation libraries, like JDom [32], to implement this algorithm. With regard to the flow graph parameters (i.e., branching probabilities and internal failure probability at each node), they can be estimated through monitoring activities, as discussed in Sect. 12.3.2.

However, according to the decentralization and autonomy principles of the SOA paradigm, the provider of a composite service (which could be in turn required to build a new composition) might want to adopt different transparency policies in revealing this information and in selecting the services his/her own service requires. In the following we present an architecture that implements the methodology described above, still remaining compliant with the SOA autonomy and decentralization principles.[9] For this purpose, we identify three possible policies that cover the spectrum of different autonomy degrees (from high to low):

1. No transparency—The provider reveals only information about the overall (possibly parametric) service reliability. This policy implies that it is up to the provider to select the services to be assembled when the service is invoked, and to calculate the resulting reliability. A provider adopting

[9] The content of this section is based on results presented in [15].

this policy wants to maintain full autonomy in the selection of services required by his/her service, without disclosing any information about its architecture.

2. Partial transparency—The provider reveals both the service internal reliability and its usage profile of other external services, but autonomously decides the external services to select. This policy implies that it is up to the provider to select the services to be assembled, while it is up to the invoker of the composite service to calculate its overall reliability. A provider adopting this policy wants to maintain full autonomy in the selection of services required by his/her service, but does not want to bear the burden of evaluating the resulting overall reliability (maybe because he/she selects services according to different criteria than reliability).
3. Total transparency—The provider reveals both the internal reliability and the usage profile of the service, indicating only the kind of services that should be selected, without actually selecting them. This policy implies that it is totally up to the user of the composite service to select the services to be assembled and to calculate the resulting reliability. A provider adopting this policy actually provides only a service template consisting of some glue logic that connects services to be selected, and does not want to bear the burden of both selecting those services and evaluating the resulting reliability.

Figure 12.2 illustrates the SOA compliant architecture in which we propose to implement the reliability prediction methodology of Sect. 12.4. The building blocks of this architecture are

- an implementation of the *Rel-pro-Alg()* (which, in an SOA environment, could be itself defined as a particular type of service, called *RelServ* in Fig. 12.2);
- an operation *GetFlowGraph(S)* that returns the flow graph describing the behavior of a service *S*. Note that this operation could be included within the ones offered in a Web accessible S interface; it is the responsibility of the S provider to build and parameterize the flow graph, using his/her knowledge of the S structure and the results of monitoring the S execution.

A client of a service S who wants to get a prediction about the service reliability must first call the *GetFlowGraph(S)* operation, and then invoke the *RelServ* service, passing to it the obtained flow graph as a parameter. The result returned by *RelServ* must then be combined with the reliability of the network the client uses to access S, according to expression (12.1).

In this architecture, the fulfillment of the three different policies listed above is guaranteed by the implementation of the *GetFlowGraph(S)* operation (which is under the control of the S provider) as follows:

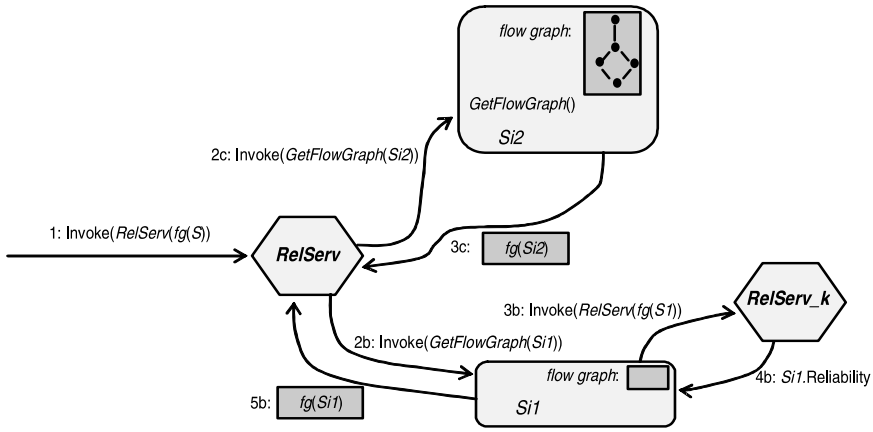

Fig. 12.2. The *RelServ* service architecture. The path (2b, 3b, 4b, 5b) describes the distributed part of this architecture (scenario *b*): the request from *RelServ* to get the flow graph of *Si1* causes the invocation of another instance of the reliability prediction service (*RelServ_k*); the flow graph returned to *RelServ* is actually the "collapsed" version of the "true" flow graph, so that *RelServ* gets no knowledge about *Si1* except for its overall reliability. The path (2c, 3c) describes the internally recursive realization of *RelServ* (scenario *c*): the reliability of *Si2* is recursively calculated by *RelServ* itself, since only the usage profile of *Si2* is returned to *RelServ*; in this way, *RelServ* gets knowledge about the internal realization of *Si2*

a) S is an atomic service – *GetFlowGraph(S)* returns the (single node) flow graph associated with S;
b) S is a composite service whose provider adopts a "no transparency" policy – *GetFlowGraph(S)* builds and returns a flow graph consisting of a single stage that expresses the overall reliability of S; this "collapsed" flow graph is built by "privately" invoking a (possibly different) instance of *RelServ* (i.e., $RelServ_k$ in Fig. 12.2). In this way, the invoker of *GetFlowGraph(S)* gets only information about the overall S reliability;
c) S is a composite service whose provider adopts a "partial or total transparency" policy – *GetFlowGraph(S)* returns the "true" flow graph describing the internal structure of S. In this way, the invoker of *GetFlowGraph(S)* gets information about the internal reliability and usage profile of S.

We point out that the flow graph $fg(S)$, that *RelServ* receives as input to calculate the reliability of a service S, must explicitly specify the services that S invokes during its execution. This information is necessary to contact these services and get the corresponding reliability information (i.e., step 7 of the Algorithm 1). These services are determined by means of a suitable selection procedure. If the provider of S adopts a "no transparency" or "partial transparency" policy, then this selection is carried out by the provider. Otherwise, it must be carried out by the one that is requesting the evaluation of the S reliability. In both cases, this procedure must in general be carried

out each time we want to evaluate the reliability of *S*. Indeed, given the dynamic nature of an SOA environment, the same *abstract service* (i.e., the type of inner service required at a certain stage of execution of a composite service) could be bound to different *concrete services* (i.e., Internet accessible implementations of an abstract service). This is due to the possibility that either previously accessible concrete services could be no longer available, or new concrete services could have emerged. A discussion on how to set up a service selection procedure is beyond the scope of this chapter, and a quite rich literature exists about this topic [2, 8, 9, 25, 26]. We only remark that the reliability analysis methodology presented here can be used to support such a selection procedure, by comparing the reliability of different available concrete services.

Given this implementation of the *GetFlowGraph(S)* operation, we get a mixed recursive/distributed implementation of the recursive algorithm *Rel-pro-Alg()* of Sect. 12.4. The distributed implementation occurs when *RelServ* executes step 6 of *Rel-pro-Alg()* under the scenario *b* described above, since this in general involves a call, on behalf of *GetFlowGraph()*, to a different *RelServ* instance. The true recursion occurs when step 6 is executed under scenario *c*; in this case *RelServ* must first solve the Markov processes associated with the services invoked by *S*, before solving the Markov process associated with *S*.

Finally, it is worth noting that, in this architecture, if a single-node flow graph is received as a result of a *GetFlowGraph(S)* call, then there is no need to be aware whether this is occurring under scenario *a* or *b*. Hence, besides allowing a service provider to select the preferred autonomy level, this architecture allows also to not revealing the autonomy policy that has been actually selected, thus preserving the "privacy" of each provider.

12.6 A Case Study

In this section we apply our modeling framework to a slightly modified and simplified version of the scenario outlined in the introductive chapter of this book. In particular, we consider the "travel planning" part of this scenario.

We assume that John, to plan his trip, invokes an Internet accessible *TripPlanner* service. This service allows John to specify his preferences and constraints about the trip, and supports him in an interactive way in the selection and booking of everything he could need during this trip.

Figure 12.3 shows a possible workflow of this service. As shown, *TripPlanner* invokes several other services to carry out its task. Given the user preferences and constraints, it first invokes a specialized *HotelSelection* service that returns a possible list of hotels for the site of interest, with their corresponding features. Looking at these features, John selects an hotel. As the information provided by *HotelSelection* could not be up to date, *TripPlanner* checks (by contacting the *HotelInfo* service of that hotel) whether

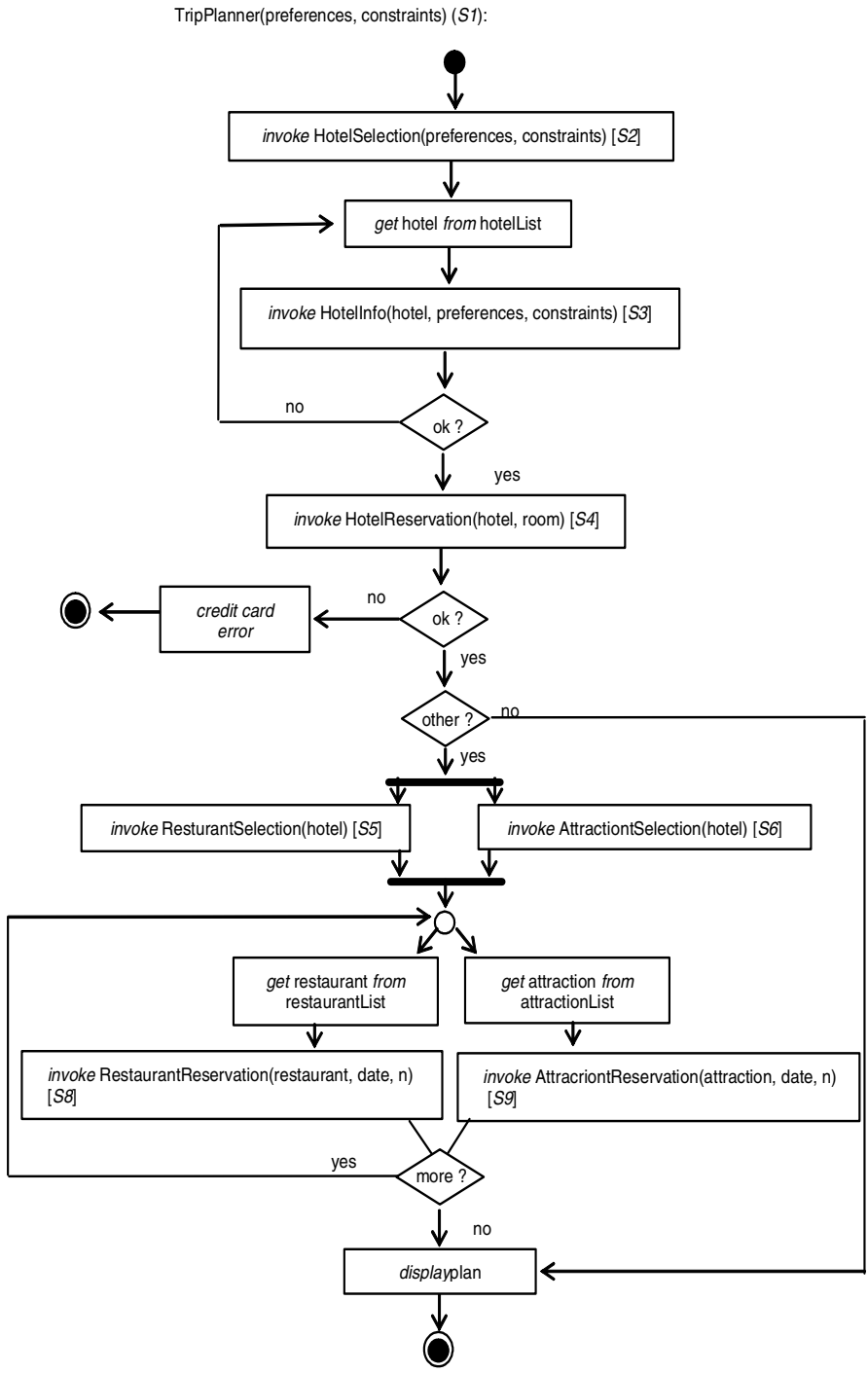

Fig. 12.3. A workflow for the *TripPlanner* service

the listed features are really present (e.g., the swimming pool, even if present, could be closed). When a selection has been finalized, *TripPlanner* invokes the *HotelReservation* service of the selected hotel to make a reservation, providing the John's credit card number. *HotelReservation*, in turn, invokes the *CreditCardManager* service of the credit card company to check whether the card is valid. In the positive case, the booking is completed and *TripPlanner* asks John whether he wants to reserve some attractions or restaurants in the site he is going to visit. Also in this case, *TripPlanner* relies on two specialized services (i.e., *RestaurantSelection* and *AttractionSelection*) to get a list of possible attractions and restaurants close to the hotel that has been selected. Given these lists, John selects one or more attractions or restaurants. After that, *TripPlanner* displays the complete trip plan.

Figure 12.4 depicts the workflows of other services invoked in the *TripPlanner* scenario. *TripPlanner* is in fact a composite service that exploits other services. One of the invoked services (i.e., *HotelReservation*) is, in turn, a composite service, as it invokes another service to carry out its task.

To predict the reliability of a given composition of such services, the provider of each of such services has to "distill" from its workflow the flow graph described in Sect. 12.4, parameterizing it with suitable transition probabilities and failure probabilities (possibly estimated through a monitoring activity).

Figure 12.5 depicts possible flow graphs associated with some of the services included in our scenario. For example, we can see that most (except the last one) of the stages in *TripPlanner* flow graph have an internal failure probability equal to zero. This means that the reliability of this service will mostly depend on the reliability of the services it invokes and of the network it uses to contact them. For example, we could assume that John starts his session with *TripPlanner* while he stays at work (thus using a reliable wired

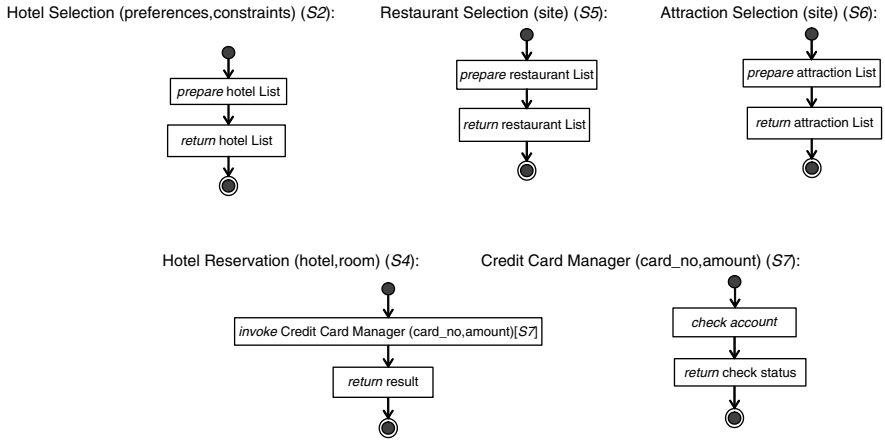

Fig. 12.4. Workflows of services invoked from *TripPlanner*

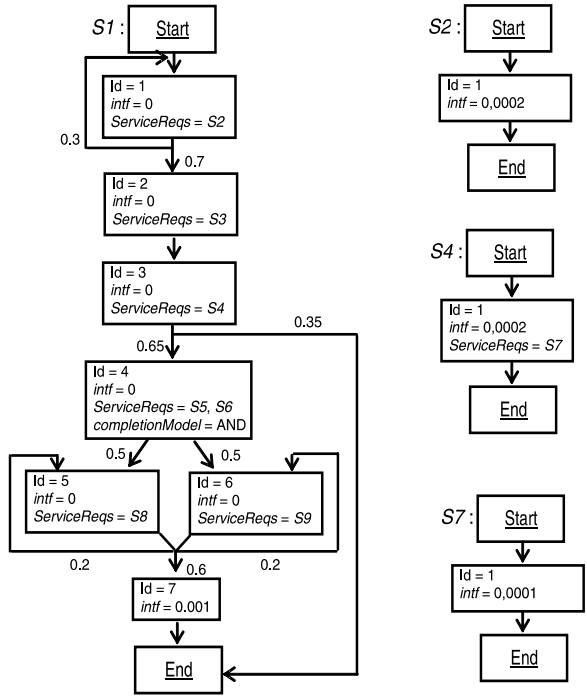

Fig. 12.5. Flowgraphs of services in the *TripPlanner* scenario

Internet connection), but he completes the last part of the session (attraction and restaurants reservation) while he is going back home using public transportation (thus using a less reliable wireless connection).

Given this scenario, the Algorithm 1 can be executed on the flow graphs shown in Fig. 12.5 to retrieve at what extent the reliability of the wireless network affects the overall reliability of the system.

12.7 Conclusions

In this chapter, we have introduced the problem of modeling and analyzing reliability of Service-Oriented Architectures. We have raised the main issues related to this problem that fall, on one hand, in the general problem of composing QoS attributes in modular software systems and, on the other end, in the specific constraints and requirements of the SOA environment such as automation support and provider autonomy.

Upon introducing very basic reliability concepts, we have presented a model for SOA reliability and an algorithm to evaluate our model. Finally, we have described a service implementation of our methodology.

We have tried to open a window on the actual possibility of pursuing model-based automated prediction of QoS attributes in SOAs. In fact we retain that these practices have not yet entered the development and assembly

process mostly for lack of automated supports, and we hope that the research community will spend more efforts in future in this direction, because very good results can be at hand in the next few years in the field of model-based QoS in SOAs.

References

1. H. Ammar , D. Nassar, W. Abdelmoez, M. Shereshevsky, A. Mili, "A Framework for Experimental Error Propagation Analysis of Software Architecture Specifications", Proc. of International Symposium on Software Reliability Engineering (ISSRE'02), 2002.
2. Ardagna, D., Pernici, B., "Global and Local QoS Guarantee in Web Service Selection", Proc. of Business Process Management Workshop, 2005.
3. A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", IEEE Trans. on Dependable and Secure Computing, Vol.1, no.1, January-March 2004, pp. 11–33.
4. L. Baresi, C. Ghezzi, S, Guinea, "Smart monitors for composed services", Proc. of 2nd International Conference on Service Oriented Computing (ICSOC'04), 2004.
5. B. Bhusha, J. Hall, P. Kurtansky, B. Stiller, "Operations Support System for End-to-End QoS Reporting and SLA Violation Monitoring in Mobile Services Environment", Quality of Service in the Emerging Networking Panorama, LNCS 3266, 2004.
6. R. Buyya, D. Abramson, J. Giddy, H. Stockinger, "Economic models for resource management and scheduling in Grid computing", Concurrency and Computation: Practice and Experience, Vol. 14, 2002, pp. 1507–1542.
7. G. Candea, A. Fox, "Crash-only software", Proc. of the 9th Workshop on Hot Topics in Operating Systems, 2003.
8. Canfora, G., Di Penta, M., Esposito, R., Villani, M. L., "An Approach for QoS-aware Service Composition Based on Genetic Algorithms", Proc. of Genetic and Computation Conference, 2005.
9. F. Casati, M. Castellanos, U. Dayal, M.C. Shan, "Probabilistic, Context-sensitive, and Goal-oriented Service Selection", Proc. of 2nd International Conference on Service Oriented Computing (ICSOC'04), 2004.
10. I. Crnkovic, M. Larsson, O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes", Proc. of Workshop on Architecting Dependable Systems (WADS'04), 2004.
11. S. Gokhale, W.E. Wong, J.R. Horgan, K. Trivedi, An analytical approach to architecture-based software performance and reliability prediction, Performance Evaluation, n.58 (2004), pp. 391–412.
12. K. Goseva-Popstojanova, A.P. Mathur, K.S. Trivedi, "Architecture-based approach to reliability assessment of software systems", Performance Evaluation, no. 45 (2001), pp. 179–204.
13. V. Grassi, "Architecture-based Reliability Prediction for Service-oriented Computing", Architecting Dependable Systems III (R. de Lemos, A. Romanovsky, C. Gacek Eds.), LNCS 3549, Springer-Verlag, 2005, pp. 279–299.

14. V. Grassi, V. Cortellessa, "Embedding error propagation in reliability modeling of component-based software systems", Proc. of International Conference on Quality of Software Architectures (NetObjectDays'05), 2005.
15. Grassi, V., Patella, S., "Reliability Prediction for Service-Oriented Computing Environments", IEEE Internet Computing, Volume 10, Issue 3 (2006), pp. 43–49.
16. Inverardi, P., Scriboni, S., "Connectors Synthesis for Deadlock-Free Component-Based Architectures", Proc. of Automated Software Engineering Conference (ASEÃ01), 2001.
17. H. Ludwig, A. Keller, A. Dan, R. Franck, and R.P. King, "Web Service Level Agreement (WSLA) Language Specification", IBM Corporation, July 2002.
18. Ludwig, H., Dan, A., Kearney, R. Cremona, "An Architecture and Library for Creation and Monitoring of WS-Agreements", Proc. of 2nd international conference on service oriented computing (ICSOC'04), 2004.
19. M.R. Lyu (Editor), "Handbook of Software Reliability Engineering", IEEE Computer Society Press, 1996.
20. Maximilien, E.M., Singh, M.P., "Toward Autonomic Web Services Trust and Selection", Proc. of International Conference on Service Oriented Computing (ICSOC'04), 2004.
21. I.V. Papaioannou, D.T. Tsesmetzis, I.G. Roussaki, M.E. Anagnostou, "A QoS Ontology Language for Web-Services", Proc. of the 20th International Conference on Advanced Information Networking and Applications (AINA'06), Vol. 1, 2006.
22. P. Popic, D. Desovski, W. Abdelmoez, B. Cukic, "Error propagation in the reliability analysis of component based systems", Proc. of International Symposium on Software Reliability Engineering (ISSREÃ05), 2005.
23. M. Tian, A. Gramm, T. Naumowicz, H. Ritter, J. Schiller, "A Concept for QoS Integration in Web Services", Proc. of the 4th International Conference on Web Information Systems Engineering Workshops (WISEWÃ03), 2003.
24. H.C. Tijms, "Stochastic models: an algorithmic approach", John Wiley and Sons, 1994.
25. Yu, T. and Lin, K. J., "Service Selection Algorithms for Web Services with End-to-End QoS Constraints", Journal of Information Systems and E-Business Management, vol.3, no.2, July 2005.
26. Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H., "QoS-Aware Middleware for Web Services Composition", IEEE Trans. Software Engineering, vol.30, no.5, August 2004.
27. C. Zhou, L.T. Chia, B.S. Lee, "DAML-QoS Ontology for Web Services", Proc. of IEEE International Conference on Web Services, 2004.
28. "UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms", OMG Adopted Specification, ptc/2004-06-01, 2004.
29. "Parlay Web Services Overview", The Parlay Group: Web Services Working Group, Version 1.0, Oct. 2002, on line at: www.parlay.org.
30. "Web Services Description Language 1.1", W3C Note, March 2001, http://www.w3.org/TR/wsdl.
31. "Business Process Execution Language for Web Services 1.1", http://www-128.ibm.com/developerworks/library/specification/ws-bpel/.
32. www.jdom.org