Architecture-based Dependability Prediction for Service-oriented Computing

Vincenzo Grassi
Università di Roma "Tor Vergata", Italy
vgrassi@info.uniroma2.it

Abstract

In service-oriented computing, services are built as an assembly of pre-existing, independently developed services. Hence, predicting their dependability is important to appropriately drive the selection and assembly of services, to get some required dependability level. We present an approach to the dependability prediction of such services, exploiting ideas from the Software Architecture- and component-based approaches to software design.

1. Introduction[1]

In the service-oriented computing (SOC) paradigm, an application is built as composition of components and services (including both basic services, e.g. computing, storage, communication, and "advanced" services that incorporate some complex business logic) provided by several independent providers [4]. A basic requirement for SOC is that support should be given to automatically discover and select the services to be assembled. The "Web services" and "Grid computing" frameworks represent standardization efforts in this area.

An important issue for applications built in this way is how to assess, as much as possible automatically to remain compliant with the SOC requirements, their quality, for instance their performance or dependability characteristics. In this paper, we focus on dependability aspects, and provide an approach that lends itself to automatization to predict the service *reliability*, defined as a measure of its ability to successfully carry out its own task. The main goal of this approach is to define a compositional way for predicting the service reliability, that reflects the underlying structure of a service realized within the SOC framework. To this purpose, we exploit ideas taken from Software Architecture- and Component-based approaches to software design.

Approaches to the reliability analysis of component-based systems have been already presented (e.g. [5, 6, 8]). What distinguishes our approach is the exploitation of a "unified" service model that helps in modeling and analyzing different architectural alternatives, where the characteristics of both "high level" services (typically offered by software components) and "low level" services (typically offered by physical devices) are explicitly taken into consideration. This model allows us to explicitly deal also with the impact on the overall reliability of the infrastructure used to assemble the components and make them interact. Moreover, we point out the importance of considering the impact on reliability of service sharing, that could typically happen in a SOC framework, when we assemble originally independent services in such a way that they exploit some common service, so being no longer independent. Finally, to better support compositional analysis, we also point out the need of explicitly dealing with the dependency between the input parameters for some service and the input parameters of cascading service requests that the service itself generates, as also pointed out in [2]. The paper is organized as follows. In section 2 we discuss general issues for an architecture-based approach to QoS prediction in a SOC framework. In section 3 we focus on reliability, and present ideas for its architecture-based prediction. In section 4 we outline a simple example, while section 5 concludes the paper.

2. Architectural approach to predictive QoS analysis

According to the Software Architecture approach [1], an application is seen as consisting of a set of *components* that offer and require services, connected through suitable *connectors*. In particular, special emphasis is given to the connector concept, that embodies all the issues concerning the connection between offered and required services; hence, a connector can also represent a complex architectural element carrying out tasks that are not limited to the mere transmission of some information, but could also include middleware services such as security and fault-tolerance. Using different types of connectors to assemble the same set of services we can easily experiment the impact on the overall system QoS of different ways of architecting the service assembly.

For these reasons we adopt this vision, but extend it to deal more thoroughly with services realized within the SOC framework, where a service can also be a low level service offered by a physical resource (e.g. a processing service). To this purpose we adopt a broad concept of *component* and *connector*; in this respect, we prefer to talk of *resource* rather than *component*, since the latter term seems too strictly tied to the idea of software resource. Hence, our *resource* concept encompasses both software components, devices (like printers and sensors), and physical resources (like processors and communication links). Analogously, our *connector* concept encompasses issues pertaining to resource composition, where "resource" has the broad sense just defined, thus including both assembly relationships

[1] Work partially supported by the PRIN project: "SAHARA: Software Architectures for heterogeneous access network infrastructures" and by the FIRB project "PERF: Performance evaluation of complex systems: techniques, methodologies and tools".

among "peer" resources (typically, software components) and deployment relationships among software resources and physical resources (e.g., processors, communication links, batteries). Note that, since a connector can in general model some complex "interaction service", it can also require other services/resources by itself; as a consequence, a connector shares several characteristics with our general concept of resource; we will exploit this similarity modeling in the same way the resource requests associated with the services they offer.

In a SOC framework (but, also, more generally, in component-based approaches) a resource is expected to publish a description not only of its set of offered services (that includes their signature), but also a description of a related set of required services, plus a set of attributes and constraints that further specify conditions for a correct matching between offered and provided services. To support predictive analysis of some non-functional property of a service composition, for example its reliability, it has been argued that each resource should also publish some *analytic interface* [3], that is a representation at a suitable abstraction level of the actual resource behavior and requirements, that lends itself to the application of some analysis methodology.

We assume that this analytic interface is associated with the offered services of both resources and connectors, and includes: (i) an abstract description of the service, (ii) an abstract description of the flow of requests that will be addressed to other resources to carry out that service (abstract usage profile). We now give some detail about how to define these abstract descriptions.

With regard to point (i), the abstraction concerns both the service itself and the domains where its formal parameters, used to specify a particular service request, can take value; for example, a processing service can be abstractly defined as a service that executes a single kind of "average" operation (at some constant speed) and whose formal parameter is the number of such operations that must be executed. In general, the abstraction with respect to the real service parameter domains can be achieved by partitioning a real domain into a (possibly finite) set of disjoint subdomains, and then collapsing all the elements in each subdomain into a single "representative" element [2]. The processing service example is an extreme case, where the entire set of operations is collapsed into a single average operation.

With regard to point (ii), the abstraction consists in giving a probabilistic description of the flow of requests. For this purpose, we assume that the "abstract" flow of requests is modeled by a discrete time Markov chain, where each state models a set of actual (abstract) service requests, with the underlying assumption that the requests in this set must be fulfilled according to some completion models before a transition to the next state can take place. Moreover, a special Start state represents the entry point for the modeled flow, while an End absorbing state represents the successful completion of the service task.

Finally, we note that in general a dependence could exist between the kind of request a particular service has to fulfil and the cascading service requests that it addresses to other resources; for example, the size of a list to be ordered sent as input parameter to some sorting service has an impact on the request of processing service addressed by the sorting service itself to some processing resource. Hence, we argue that modeling this dependency is necessary to achieve a real compositional analysis. To model it, we suggest that both the transition probabilities and the actual parameters of the service requests in a flow are defined as parametric with respect to the formal parameters of the offered service they are associated with.

3. Architecture-based dependability prediction

As stated before, the reliability of a service is a measure of its ability of completing its task. Let us denote by *Pfail*(*S*,*ap**) the probability that a service *S* is unable of completing its task when called with actual parameters *ap**. Hence, the reliability of *S* can be expressed as 1 - *Pfail*(*S*,*ap**). In the following, we show how to calculate *Pfail*(*S*,*ap**), exploiting information about how *S* has been architected. We distinguish two possible cases for the resource offering that service:

- *basic resources* that do not require the services of any other resource to carry out their own service *S* (e.g., a "cpu" resource); the reliability of such services depends only on the resource internal characteristics/operations; we assume that it is known, and is expressed by a failure rate (failure/time-unit);
- *complex resources*, that do require the services of other resources to carry out their own service *S* (typically, software components);[2] hence each of their services is characterized by a flow modeling the usage profile of other services; the reliability of these services depends on both the resource internal characteristics/operations and on the reliability of the services they require; moreover, it also depends on the reliability of the connectors used to connect required and offered services; we assume that a complex resource can only provide information about its "internal reliability", expressed by some suitable reliability measure (e.g., software failure rate in the typical case of software components).

Finally, in the following discussion we assume a "fail-stop" behavior (i.e. each failure causes a service interruption), and that no repair occurs.

3.1 Reliability of services offered by basic resources

We limit ourselves to consider processing and communication services. Let us consider first a cpu-type resource offering a processing service with speed *s* (operation/time-unit) and failure rate λ: assuming an exponential failure rate, the probability of a failure during the execution of N operations is:

$$Pfail(cpu,\mathrm{N}) = 1 - e^{-\lambda \mathrm{N}/s} \qquad (1)$$

[2] Note that in our framework even a software component that does not call any other component is a complex resource, since it at least must exploit a processing service.

Then, let us consider a network-type resource offering a communication service with bandwidth *b* (byte/time-unit) and failure rate λ: analogously to (1), the probability of a failure in transmitting B bytes is:

$$Pfail(net,B) = 1 - e^{-\lambda B/b} \qquad (2)$$

Within this type of resources, we also include the case of connectors that do not have any flow of service requests associated with them; this is the case of connectors simply modeling an association between resources, like a "local processing" connector between a software component and the cpu resource of the same node where the component is located. Besides not making use of any resource, these connectors do not correspond to any tangible artifact, and hence we assume that their failure probability is zero.

3.2 Reliability of services offered by complex resources

Under this case, we also include "interaction" services offered by connectors that exploit other resources (typically communication and, possibly, processing resources) to carry out them; in the following we do not make any basic distinction between such connectors and generic complex resources, from the viewpoint of the reliability evaluation of the service they offer. Given our flow model, we have:

$$Pfail(S,ap^*) = 1 - p^*(\underline{Start},\underline{End})$$

where p*(<u>Start</u>,<u>End</u>) denotes the probability of reaching in any number of steps the <u>End</u> absorbing state for the flow associated with *S*, starting from the <u>Start</u> state. To calculate a non trivial value for such a probability, we must specialize the flow model to the dependability domain, taking into consideration the possibility that a failure may occur at any flow stage. Under the fail-stop and no repair assumptions, this corresponds to adding an additional transition from each state *i* of the flow to a new <u>Fail</u> absorbing state, with probability p(*i*,<u>Fail</u>), weighing with a probability 1-p(*i*,<u>Fail</u>) the already existing transitions to other states (except transitions from the <u>Start</u> state, since we assume that it does not represent any real behavior). Standard Markov methods can be exploited to evaluate p*(<u>Start</u>,<u>End</u>) once the failure probabilities have been added to the original flow model [8]. In the following, we focus on issues related to the evaluation of p(*i*,<u>Fail</u>), exploiting architectural information.

We recall that in our flow model we have assumed that each state *i* models a set of activities *A1*, ..., *An*, where each activity *Aj* consists of a request for a service *Sj* with actual parameters *apj** (i.e., *Aj* ≡ *call*(*Sj*,*apj**)). Hence, to evaluate p(*i*,<u>Fail</u>) we must take into account: *(i)* the failure probability of each *Aj*; *(ii)* an overall completion model for *A1*, ... *An*, to determine when a successful transition to the next flow stage is enabled, even if some *Aj* has failed; *(iii)* the existence of possible dependencies among the *A1*, ... *An*, that can affect their failure probability.

Let us consider separately the above three points.

(i) Failure probability of each Aj: to evaluate the probability of a failure of *Aj*, that we denote by Pr{*fail*(*Aj*)}, we must take into account the following failure probabilities:

- *Pfail_int*(*Aj*): probability of "internal" failure (i.e. depending on the internal characteristics of the resource requesting the service) related to the service request for (*Sj*,*apj**); we discuss at the end of this section issues concerning the evaluation of this probability;
- *Pfail_ext*(*Aj*): probability of "external" failure related to the service request (*Sj*,*apj**); this probability depends in its turn on the probability of failure of the service *Sj* itself, and on the probability of a failure in the connector *Cj* that "transports" the request, that is on *Pfail*(*Sj*,*apj**) and *Pfail*(*Cj*,[*Sj*,*apj**]) (where [*Sj*,*apj**] is the parameter for the connection service offered by *Cj*).[3]

(ii) Completion model for A1, ... An: we consider two possible completion models for the activities in state *i*:

- *AND* model: all the activities *A1*, ... *An* must be completed to enable a transition to the next state;
- *OR* model: at least one of *A1*, ... *An* must be completed to enable a transition to the next state.

(iii) Dependency model for A1, ... An: with this model we take into account the sharing of some common service, that could typically occur in a SOC framework:

- *no sharing*: *A1*, ... *An* do not share any common service, and hence are assumed independent of each other;
- *sharing*: *A1*, ... *An* do share a common service, and hence their failure probabilities are not independent.

In particular, in the *sharing* model we consider the case where all the *A1*, ... *An* are actually requests for the same service offered by a single resource. This case should be taken into account in particular in the *OR* completion model that could be used, for example, to model some kind of "retry" operation. For this reason, in the following we will provide expressions for the *sharing* model under the *OR* completion model only.

Note that the component reliability model presented in [5] considers only a single activity in each flow state (hence, for example, it does not consider fault-tolerance features), while the model in [8] take into consideration the *AND* and *OR* completion models for multiple activities in a flow state, but does not consider the possible dependency caused by service sharing; moreover, both models do not introduce explicitly the service actual parameters in the reliability evaluation, so that compositional analysis is not completely supported.

Using the above defined probabilities, and completion and dependency models, we can define expression for the probability p(*i*,<u>Fail</u>) of a failure in state *i* of the flow. Let us start with the calculation of p(*i*,<u>Fail</u>) under the two completion models:

[3] Note that *Pfail*(*Sj*,*apj**) and Pr{*fail*(*Aj*)} denote the probability of two different events, where the former corresponds to a failure in the execution of service *Sj*, while the latter correspond to a failure in the "call" of *Sj*, that includes a failure in the execution of *Sj*, or in other related activities, like the transport of the request and result.

• *AND* model: the failure of any activity in state *i* causes a failure of that state; hence we have:

$$p(i, \underline{\text{Fail}}) = 1 - \Pr\{ \bigcap_{j=1}^{n} nofail(Aj)\} \quad (3)$$

• *OR* model: the failure of all the activities in state *i* causes a failure of that state; hence we have:

$$p(i, \underline{\text{Fail}}) = \Pr\{ \bigcap_{j=1}^{n} fail(Aj)\} \quad (4)$$

Now, let us see how we can calculate the two probabilities in the right hand side of (3) and (4) under the two defined dependency models.

• *no sharing* model: thanks to the independence assumption, we can rewrite (3) and (4), respectively, as:

$$p(i, \underline{\text{Fail}}) = 1 - \prod_{j=1}^{n} (1 - \Pr\{fail(Aj)\}) \quad (5)$$

$$p(i, \underline{\text{Fail}}) = \prod_{j=1}^{n} \Pr\{fail(Aj)\} \quad (6)$$

where each Pr{*fail*(*Aj*)} can be calculated as follows:

$$\begin{aligned}\Pr\{fail(Aj)\} &= 1 - (1-Pfail\_int(Aj))\cdot(1-Pfail\_ext(Aj))\\ &= 1 - (1-Pfail\_int(Aj))\cdot(1-Pfail(Sj,apj^*))\\ &\quad\cdot(1-Pfail(Cj,[Sj,apj^*])) \quad (7)\end{aligned}$$

since *Aj* does not fail only if neither an internal nor an external failure occurs, and, in its turn, the external failure does not occur if neither the used connector nor the requested service fail.

• *sharing* model: in this case *A1*, ... *An* are no longer independent. Limiting ourselves to the *OR* case, as stated above, we rewrite (4) as:

$$\begin{aligned}p(i, \underline{\text{Fail}}) &= \Pr\{ \bigcap_{j=1}^{n} fail(Aj) \mid noextfail\}\Pr\{noextfail\}\\ &+ \Pr\{ \bigcap_{j=1}^{n} fail(Aj) \mid extfail\}\Pr\{extfail\} \quad (8)\end{aligned}$$

where *extfail* and *noextfail* denote the events “external failure occurrence” and “no external failure occurrence” in *A1*, ... *An*, respectively. Now, note that when no external failure occurs, each *Aj* can only fail because of an internal failure; in the opposite case, that is when an external failure occurs for some *Aj*, it causes the failure of all the *A1*, … *An* with probability one, since they share the same external service and we have assumed that no repair occurs. Hence, reasonably assuming that the internal failures are independent, we can refine equation (8) as follows:

$$\begin{aligned}p(i,\underline{\text{Fail}}) &= \prod_{j=1}^{n} Pfail\_int(Aj) \cdot \prod_{j=1}^{n} (1 - Pfail\_ext(Aj))\\ &+ 1\cdot(1 - \prod_{j=1}^{n} (1 - Pfail\_ext(Aj))) \quad (9)\end{aligned}$$

If we compare equations (6) and (7) with equation (9), we can see the different results obtained for the *OR* completion model under the *no sharing* and *sharing* models; this remarks the importance of considering service sharing.

To conclude this section, we give some suggestion about how the internal failure probability *Pfail_int*(*Aj*) can be evaluated. For this purpose, we distinguish two cases:

a) *Aj* is the request for a service offered by some complex resource, and hence typically corresponds to an actual method call; in this case, the internal operations related to this request just consist of the “call” of such service, while other operations connected to the request (e.g., parameters marshaling/unmarshaling) in our architectural vision are captured under the connector concept; hence, we must give some suitable value to *Pfail_int*(*Aj*) reflecting the reliability of the call operation only; this value could also be set equal to zero, if we assume that a method call is a reliable operation that does not cause failure by itself;

b) *Aj* is the request made by a software component to a processing service to execute N operations (i.e. *Aj* = *call*(*cpu*,N)), then *Pfail_int*(*Aj*) must depend on N and represents the probability that the software code for the N operations causes a failure; assuming that the probability of a software failure in an operation is ϕ, we can write:

$$Pfail\_int(Aj) = Pfail\_int(call(cpu,N)) = 1-(1-\phi)^N \quad (10)$$

Corresponding expressions can be written under different software failure models.

Finally, considering automatization issues, we remark that the reliability evaluation methodology presented above basically defines an iterative procedure, where the reliability of a service is calculated in terms of the reliability of the services it requires (according to a given usage profile). The bottom of this iteration is given by services offered by basic resources, whose reliability can be directly calculated.

4. Example

We use a very simple example to illustrate the above ideas. For this purpose, we consider a resource that offers a search service for an item in a list; to carry out this service, it requires in its turn a sorting service (to possibly sort the list before performing the search) and a processing service (for its internal operations). The search service has three parameters, with the former two used to provide the item to be searched and the list, respectively, while the third is used to return the search result. In an abstract characterization of this service, the abstract domain of the former two parameters is the set of integer numbers, used to specify, respectively, the size of the element to be searched and the list size. Moreover, this service is characterized by a software failure rate $\phi 1$. The sort service has one parameter, used to provide the list to be sorted. This service is characterized by a software failure rate $\phi 2$.

Figure 1 depicts the flows associated with these two services, where beside each state it is shown the associated service request with its parameters.

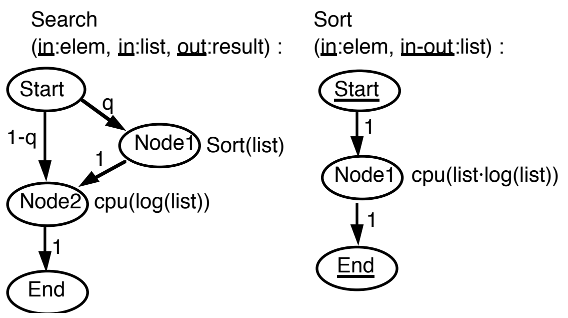

Figure 1. Flows of the search and sorting services.

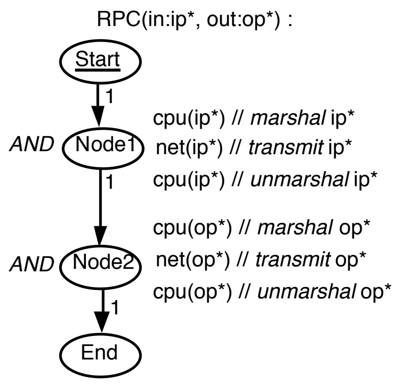
RPC(in:ip*, out:op*) :
Start
1
cpu(ip*) // *marshal* ip*
AND
Node1
net(ip*) // *transmit* ip*
1
cpu(ip*) // *unmarshal* ip*
cpu(op*) // *marshal* op*
AND
Node2
net(op*) // *transmit* op*
1
cpu(op*) // *unmarshal* op*
End

Figure 2. Flow of a RPC connector.

We assume that the components providing these two services are allocated to two different processing nodes (*cpu1* and *cpu2*), connected by a communication network (*net1-2*). In an abstract characterization of these physical devices, they can be modeled as basic resources, offering processing and communication services, respectively, each with only one abstract parameter used to provide the number of operations to be processed, and the bytes to be transmitted. The two processing resources (*cpu1* and *cpu2*) are also characterized by speed attributes *s1* and *s2*, and by failure rates λ1 and λ2. Analogously, the communication resource has a bandwidth attribute *b*, and failure rate λ.

The search and sorting services are connected through a RPC connector. From a reliability viewpoint, this connector plays a role similar to a complex resource, as it requires other services (processing and communication) to carry out its own interaction service. Figure 2 shows the flow associated with this connector, where the input and output parameters correspond to the data transmitted from the client to the server and vice-versa, respectively.

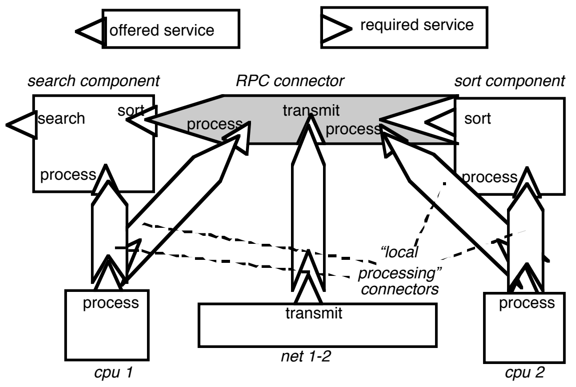

Figure 3. Model of the search service on a platform with two processing resources.

Finally, figure 3 shows how all these services are assembled together. To evaluate the reliability of this assembly, we must add failure information to the service flows, as discussed in section 3. Then, we can apply the methodology of section 3. Let us denote by n the size of the list to be searched. For space limits, we only show the evaluation of *Pfail*(*sort*,n). Using (1) and (10) in (7), and assuming that a "local processing" connector does not cause a failure, we have, from the sort service flow:

$$\begin{aligned} Pfail(sort,\mathrm{n}) &= \Pr\{fail(call(cpu2,\mathrm{nlog(n)}))\} \\ &= 1 - (1-Pfail\_int(call(cpu2,\mathrm{nlog(n)})) \\ &\quad \cdot(1-Pfail(cpu2,\mathrm{nlog(n)})) \\ &= 1 - (1-\lambda 2)^{\mathrm{nlog(n)}} \cdot e^{-\lambda 2 \cdot \mathrm{nlog(n)}/s2} \qquad (11) \end{aligned}$$

5. Conclusions

We have presented an approach to the reliability prediction of an assembly of services, that allows to take into account in an explicit and compositional way the reliability characteristics of both the resources and interaction infrastructure used in the assembly. However, several points require further investigation. They include, for example, how to actually make automatic, as much as possible, the reliability prediction of a service assembly, that is important in a true SOC perspective. This point involves the precise definition of reliability evaluation algorithms, and the inclusion in a machine-processable language of appropriate constructs to express the dependability-related characteristics of resources and connectors. It should also be noted that the iterative evaluation procedure outlined at the end of section 3 does not work in the case of a service assembly where some services recursively call each other. In this case, the assembly reliability would be expressed by a fixed point equation, for which appropriate evaluation methods should be devised. Another point concerns the dependency model, that should be extended to deal with more complex dependencies, and the fail-stop assumption, that should be released to deal also with failure propagation aspects.

Finally, we would like to remark that, even if our focus is on dependability issues, the presented ideas can also be extended to other service quality aspects (e.g. performance).

References

[1] L. Bass, P. Clements, R. Kazman, *Software Architectures in Practice*, Addison-Wesley, 1998.

[2] D. Hamlet, D. Mason, D. Woit "Properties of Software Systems Synthesized from Components", June 2003, on line at: http://www.cs.pdx.edu/~hamlet/lau.pdf (to appear as a book chapter).

[3] S. Hissam et al. "Enabling predictable assembly" *Journal of Systems and Software*, vol. 65, 2003, pp. 185-198.

[4] M.P. Papazoglou, D. Georgakopoulos "Service oriented computing" *ACM Communications*, vol. 46, no. 10, Oct. 2003, pp. 24-28.

[5] R.H. Reussner, H.W. Schmidt, I.H. Poernomo "Reliability prediction for component-based software architectures" *Journal of Systems and Software*, vol. 66, 2003, pp. 241-252.

[6] J. Stafford, J.D. McGregor "Issues in predicting the reliability of composed components" *5th ICSE CBSE Workshop*, Orlando, Florida, May 2002.

[7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison Wesley, 2002.

[8] W.-L. Wang, Y. Wu, M.-H. Chen "An architecture-based software reliability model" *IEEE Pacific Rim Int. Symp. on Dependable Computing*, Hong Kong, 1999.