Service Integration: A Soft Spot in the SOA Testing Stack

Интеграция сервисов: слабое место в стек тестировании SOA

Sebastian Wieczorek
SAP Research
Darmstadt, Germany
sebastian.wieczorek@sap.com

Alin Stefanescu
SAP Research
Darmstadt, Germany
alin.stefanescu@sap.com

Abstract

The concepts of Service Oriented Architectures (SOA) have manifested themselves as the next evolutionary step for software development and the industry is fast in adopting SOA. The gained freedom of combining loosely coupled services is countered by resulting challenges for testing. While current research is mainly focusing on the investigation of advanced SOA feature like semantic service retrieval and dynamic binding, industrial projects still face fundamental problems in assuring software quality of SOA applications. In this paper we describe the SOA testing stack and the different objectives of its layers. We further explain why traditional and currently discussed testing techniques alone are not sufficient to cover all relevant testing layers. The unique SOA challenges posed by message based communication are usually not considered in research and industry. Therefore we discuss them in more detail and give advice of how to address them in a holistic development process.

Концепция сервис-ориентированных архитектур (SOA) заявила себя как следующий эволюционный шаг в разработке программного обеспечения и индустрия торопится в принятии концепций SOA. Однако полученную свободу объединять слабосвязанные сервисы можно противопоставить вызванным этой свободой проблемам тестирования. Хотя данное исследование ориентируется в основном на изучении передовых функций SOA, таких как семантический поиск сервисов и динамическое связывание, промышленные проекты по-прежнему сталкиваются с существенными проблемами в обеспечении качества SOA приложений. В этой статье мы описываем стек SOA тестирования и различные цели его уровней. Мы также объясним, почему традиционные и теперешние методы тестирования сами по себе не является достаточным для покрытия всех необходимых слоев тестирования. Уникальные проблемы, вызванные процессом передачи информации, основанном на передаче сообщений, как правило, не рассматриваются ни в научных исследованиях, ни в промышленных подходах. Поэтому мы рассмотрим эти проблемы более подробно и дадим советы, как их решать в общем процессе разработки.

***Keywords**: testing; SOA; service integration; message-based communication.*

1. Introduction

According to Forrester [5] service-oriented architectures (SOA) are becoming mainstream in the software industry. Their survey states that 2/3 of companies taking part in the survey (i.e., 2200 across North-America and Europe) expect to be using SOA by the end of 2009 while 60% of those currently using it are expanding their usage. This trend was triggered by the promise of getting powerful, flexible, customizable, and extensible applications with less development effort by applying the SOA concept of loosely coupled services that communicate by message passing. As a consequence the world's largest enterprise software vendors IBM, SAP, Microsoft and Oracle all are offering SOA enabled products and platforms.

Paradigm shifts in the software development inevitably demand considerations regarding the methods and efficiency of testing. Traditionally three different testing layers have been advertised to enforce functional correctness of monolithic, component-based software systems (see for instance [8]):

- *Unit testing.* This is carried out alongside the development of single software units like methods, procedures or classes, which are then tested in isolation.

- *Integration testing.* This deals with the testing of aggregated functionality like clusters of classes or sub-systems.
- *System testing*. This comprises the fully integrated application, usually using its externally exposed interfaces.

It has been argued in [1] that everything apart from unit testing can be seen as integration testing and we agree. However it still makes sense to distinguish between the levels of integration testing, if they demand different strategies and techniques (like *integration testing* and *system testing* in the traditional approach).

The main contribution of the paper is to uncover several issues of integration testing for service-based applications and the provision of possible improvements of the SOA development process using state-of-the-art techniques.

In Section 2 we will introduce the four layers of SOA application testing incorporating different layers of integration and explain their aim and necessity. In Section 3 we introduce an example to show the effects of the concept of service composition. From these effects we derive some challenges for service integration testing in Section 4. By presenting related work in Section 6 we show that the identified challenges have not gained the necessary awareness in research and industry. Section 7 concludes the paper and gives an outlook to further work directions of us.

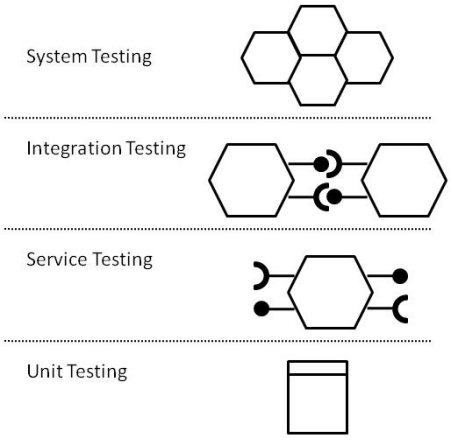

Figure 1: Scope of SOA testing layers

2. SOA testing layers

As explained previously, integration testing of SOA applications takes place on different layers. In [12], the traditional testing layers have been adjusted towards the support of component-based systems (CBS). As the general idea of partitioning applications into logical units is somehow similar to the SOA approach of encapsulating related functional units in a service, we align our definition of testing layers with the one for CBS. As a result we identified four distinct testing layers, namely *unit testing*, *service testing*, *integration testing*, and *system testing* that we illustrate in Figure 1. In the following sub-sections we explain each layer in more detail. This definition will be use throughout the remaining paper.

2.1. Unit testing

Unit testing is the best understood testing layer in research and practice. As explained in the classical definition of Section 1, it deals with the single software units in isolation. In contrast to all other testing layers, unit testing focuses on getting confidence in the functional correctness and hence in the correct implementation of the algorithms. Unit testing in SOA systems can therefore be carried out just like in CBS implementations or any other software architecture, using all available tools and techniques (i.e. white-box and black-box, manual and automated, code-based and model-based techniques). A good overview of unit testing techniques can be found in [1].

2.2. Service testing

Service testing for SOA is somehow analog to the testing of components in the CBS world. The focus here is less on correct implementation of algorithms but on the integration of the functional units inside the component (respectively service) and on the fulfillment of the contractual obligations. Applying the classical definition of testing layers, service testing is part of integration testing as introduced in Section 1. In SOA, the static aspects of services are defined in WSDL and basically correspond to the UML component or class diagrams that are used to describe the components in the CBS world. Therefore the same methods as for component testing can be used (for more details see [7]).

2.3. Integration testing

As mentioned before the loose coupling of service components is one of the distinguishing factors of SOA. In contrast to the CBS approach, integration testing cannot rely on homogenous components with tightly connected interfaces, therefore the adaptability and distribution of SOA demands additional considerations for integration testing. Especially the effects of message racing and its implications have to be considered during system development and should be tested thoroughly. Message racing in this context refers to situations where messages are not received in the same order as they are sent. This topic will be covered in our paper in detail.

2.4. System testing

System testing can be defined analog to the classical definition from Section 1. As the faultless interplay of the services can be assured on the integration testing level, in practice system testing is normally based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is therefore most appropriate to carry out the tests, as the system should be validated as a whole and only using access points that are available to the prospect user. Most commercial testing tools focus on UI level tests and offer certain degrees of automation (for more details see for instance [6]).

3. Aspects of service integration

As showed in the previous section, the major difference between CBS and SOA testing can be found on the level of integration testing. In this section we will explain in more detail, what considerations are taken during the service integration using the message based communication. We will do this on the base of the following illustrating example.

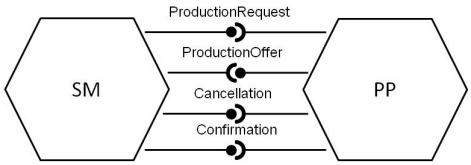

Figure 2: Structural information for sales management (SM) and production planning (PP)

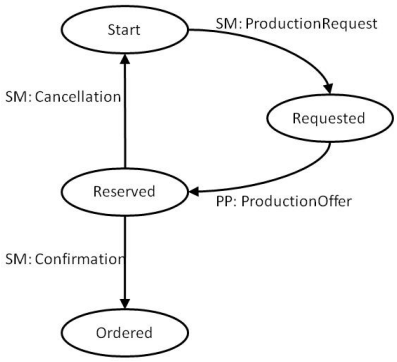

Figure 3: Basic interaction protocol of sales management (SM) and production planning (PP)

Two components usually occurring in large enterprise applications are the components offering *production planning* (PP) and *sales management* (SM). In the SOA world, these components communicate via their service interfaces. The structural information of the example is sketched in Figure 2. Furthermore we consider an interaction protocol as described in Figure 3.

When a customer places an order the SM starts a conversation with PP by sending a P*roductionRequest* message that provides the details of the order and asks for a production of the desired goods. PP processes the request, reserves the necessary resources for production and replies with a *ProductionOffer* message that includes information like the calculated delivery date. The SM can now decide whether the offered production planning is acceptable or not. In the first case, SM will send a *Confirmation* message that will trigger the execution of the production. In the second case, SM will send a *Cancellation* message that rolls back the previously sent request and signals to PP that the reserved resources can be released.

In a CBS implementation, the communication would most likely be handled by synchronous calls between the components. Synchronous communication means that the initiator is blocked from further computation until the requested component is providing the desired answer. The SOA approach however demands a loose coupling that allows more flexibility and better distribution of the components. Therefore asynchronous channels are used in addition to the synchronous ones.

Asynchronous channels provide different reliability degrees (see also [13]), for instance they can guarantee that every message is received either *exactly once* (EO) or even *exactly once in order* (EOIO). Both types of channels have mechanisms to ensure that each sent message is received once, thus preventing message loss and multiple receiving of the same message. While messages on an EOIO channel are received in the same order as they are sent, messages on the EO channel may overtake each other.

In practice this change in the message order happens when dynamic routing strategies are applied to the messages or when data corruption occurred thus forcing a message to be sent again. On EOIO channels, message racing is usually prevented by re-sorting the messages at the receiver side. In the following subsections we describe common considerations when assigning channels for SOA.

3.1. Synchronous vs. asynchronous channels

Synchronous channels are usually used when the sender of a request message demands an immediate response because its consequent actions depend on it. The blocking of the sender until the response message arrives is therefore acceptable. In the given example a low waiting time between the *ProductionRequest* and *ProductionOffer* messages is crucial for SM because the decision about accepting or cancelling the offer obviously depends on PP's response.

In contrast, asynchronous channels are preferred over synchronous ones if the sender is not dependent on an immediate response. On one hand asynchronous communication does not block the sender until it finally receives the response; on the other side this gives the receiver the freedom to delay the computation of the incoming messages in favor of more urgent tasks that demand low latency. In the given example, SM does not need an immediate assurance that its *Confirmation* message has been processed because the PP guarantees the availability of the offered resources. Also the *Cancellation* message can be sent asynchronously, as the future behavior of the SM does not depend on the feedback of the PP.

The example shows that the decision about using synchronous or asynchronous communication should be taken on message level and not for the whole SOA application. For the remainder of the paper we assume that the *ProductionRequest* and *ProductionOffer* are sent on a synchronous channel, while *Confirmation* and *Cancellation* are sent on an asynchronous channel. A more detailed discussion on synchronous vs. asynchronous SOA communication can be found in [11].

3.2. Exactly Once vs. Exactly Once in Order

In the case of asynchronous communication, EO channels have the advantage compared to EOIO that their protocol overhead is much smaller and hence the latency is reduced a lot, especially in environments with bad transmission quality. However, EO communication should only be applied if the final result of computation is independent of the order in which the messages are received. In our example, the order in which *Confirmation* and *Cancellation* messages arrive (and hence are computed) at PP might be crucial. Imagine the scenarios given in Figure 4 where SM sends two *ProductionRequest* messages during the interaction. While both depicted scenarios are valid for an asynchronous EO channel between SM and PP, the scenario on the right cannot be observed in the case of an EOIO channel because it prevents the message racing of *Cancellation* and *Confirmation*.

In the described case, an EO channel could lead to problems because the PP might for example be unable to determine, which of the two previously sent production offer has been cancelled when a *Cancellation* is received. However, if the *Cancellation* and *Confirmation* messages contain information about the corresponding offer, the PP will be able to take the right decision even if the messages arrive in changed order. Hence an EO channel with lower latency can be used in this case.

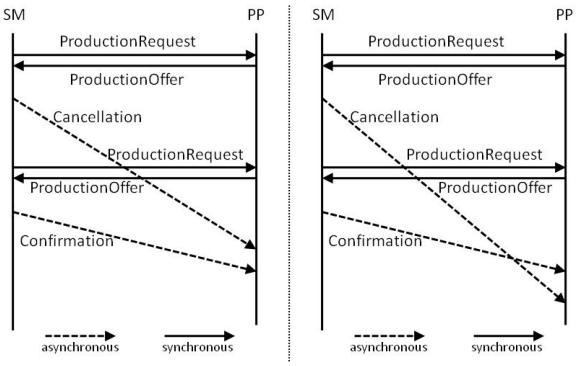

Figure 4: Example scenarios of asynchronous communication

4. Challenges of testing service integration

As we argued in the Section 2, the integration testing of SOA demands special attention as the classical approaches are not applicable "off the shelf". In Section 3 we explained the particularities of SOA service integration. In this section we will discuss the resulting challenges of the integration testing due to message exchange.

We identified three reasons why in SOA systems it can be impossible to predict in which order messages are received:

- **Usage of an EO channel.** The effect of message racing on EO channels has already been explained in Section 3. In our example of the communication between SM and PP, SM sends the *Confirmation* and *Cancellation* messages on an EO channel which may lead to the changed receive order depicted on the right of Figure 4.
- **Multiple channels.** As we discussed in Section 3.1, synchronous communication is usually much faster than asynchronous. Therefore messages sent over different channels experience different latency and hence may be received in different order. In Figure 4 the second synchronous *ProductionRequest* message is sent later by SM than the asynchronous *Cancellation* but PP receives them in the inverse order.
- **Concurrent communication.** In many business scenarios, situations occur where both participants are allowed to send messages concurrently. In the given example this would be the case if the SM was allowed to send a *Cancellation* message before receiving the *ProductionOffer* from the PP. In this case, after SM sent the *ProductionRequest*, PP and SM could act simultaneously and hence no prediction about who receives the message first can be made.

The three reasons for non-determinism in the SOA message delivery described above imply that robustness against message racing has to be considered in the system design and also checked during integration testing. It can be argued that a system design using synchronous communication only will avoid the non-determinism to a certain extend (the effects concurrent communication cannot be addressed though) but this contradicts the paradigms of SOA and also affects its performance, as explained in Section 3.1. An integration testing approach for SOA therefore clearly has to address the faults related to message racing. Only then it will possible to build confidence in the correct functioning of the system.

The practice shows that the three reasons for non-deterministic message delivery mentioned above rarely manifest themselves in test situations, because on the one hand current test frameworks are not very good at emulating the distributed nature of the productive systems and on the other hand are not exposed to high communication loads during functional tests. Therefore neither the channels nor the message queues are put under stress for functional and integration testing. But even if the system is exposed to such situations, the message racing will still be rather randomly, which also does not allow a systematic and controlled test execution.

5. Discussion on possible solutions

After describing the specific challenges of SOA integration testing in Section 4, we discuss now how to tackle them within the SOA development process. Therefore we will focus on the different development stages, i.e., design, implementation, and testing, and propose actions that ensure correct integration of the underlying services of the SOA application.

5.1. Design

In order to avoid any ambiguity, the specification of the desired service communication should be done using a formal specification and in the different means of modeling of service communication have been discussed already [4]. [15] examines the most common of them regarding their suitability in the context of integration testing. It was concluded that most of them do not consider multiple or asynchronous communication channels and therefore lack the needed expressiveness. WS-CDL[1] however supports detailed channel specifications and hence can be recommended for choreography modeling.

Having an unambiguous definition of the communication protocol also allows applying automatic verification techniques. For example the absence of deadlocks, livelocks, inconsumable messages or local enforceability can be proven at this early development stage. This effectively ensures that an infeasible design decision is identified and corrected before implementation has started and a necessary correction becomes costly.

[1] http://www.w3.org/TR/ws-cdl-10/

5.2. Implementation

One of the SOA development objectives is to decompose the software in order to carry out the implementation in a distributed setting, i.e. having multiple development teams working on the service components in parallel. A key factor for successful integration of these services is however a common understanding of the shared interfaces and the particularities of the SOA communication. Therefore developers should have enough modeling skills such that they are able to correctly interpret the system design documents.

The study of [9] shows that another factor of success is to apply continuous integration throughout the development. The idea of continuous integration is to try out the developed functionality very frequently in order to spot problems as soon as possible [9]. Another effect in practice is that the awareness and understanding of the overall design among developers is much higher.

5.3. Testing

As explained the challenges of SOA integration testing are associated with the particularities of the used communication channels and their specific properties. Another challenge described in Section 4 is that the service components usually do not expose message racing during integration tests because the test environment is rather idealistic. This implies the need of tools with the ability to simulate changes in the receive order of sent messages. One possibility to achieve this is to delay the actual sending or receiving of specific messages inside the service components. A better possibility is to use mechanisms to control the message channels during integration testing. In this way, the implementation itself can be validated without interference with its internal behavior. Moreover, there are many cases where the testers have no control over the involved components.

Once we have tools that can simulate message racing in the system, one needs to have a testing strategy such that the coverage of such situations is as high as possible. Here again the models describing the service communication can be used to generate test suites with a high model coverage. This can be done with model-based testing (MBT) techniques and tools like the ones described in [12]. Best practice for integration testing is to achieve transition coverage of the components [1]. If a modeling

approach that explicitly considers message racing was used also transition coverage of the given interaction protocol can be used [14].

6. Related work

The most comprehensive survey on SOA testing can be found in [3]. The authors group the existing research in three main categories: unit testing (based on WSDL), unit testing of service compositions (based on orchestrations) and integration testing. The approach of our paper would fall then under their integration testing category, but to the best of our knowledge, our work is not covered by the existing literature. We found only the work in [4] that addresses at their "interface behavior level" the one-to-one service communication as proposed in our paper. However, the authors there study the problem only from architectural point of view ignoring totally the testing aspects.

The existing SOA testing tools (Green Hat[2], Parasoft[3], Soasta[4], Progress Software[5], iTKO[6], etc.) mainly provide support for the service testing and system testing levels (see Figure 1) and not the service integration level. This is due to the simple fact that the models describing the service communication are usually missing. It would be beneficial to see commercial tools taking into account models that able to capture service communications protocols like WS-CDL, BPMN, BPEL4Chor, or UML-S. Unfortunately, these modeling languages are still not regularly used in large scale SOA implementations.

7. Conclusion

Applying a holistic service integration testing strategy is a key factor for successful SOA projects. In this paper we pinpointed subtle errors due to different types of message racing that could appear in a SOA implementation and some possible ways to tackle them. We used an example from the Enterprise Application Integration (EAI) area, so an intra-organizational SOA implementation, but our considerations are general enough to cover also inter-organizational SOA implementations using B2B communication.

Testing concurrent and distributed systems is difficult but our belief is that one can use the particularities and flexibility of SOA together with a formal testing approach to overcome the integration testing challenges.

[2] *http://www.greenhat.com*
[3] *http://www.parasoft.com*
[4] *http://www.soasta.com*
[5] *http://www.progress.com*
[6] *http://www.itko.com*

References

[1] S. Ali, S., Briand, L., Jaffar-Ur Rehman, M., Asghar, H., Iqbal, M.Z., and Nadeem, A., "A State-Based Approach to Integration Testing Based on UML Models", *Information & Software Technology*, 49 (11–12), pp. 1087–1106. Elsevier, 2007

[2] Binder, R.V., *Testing object-oriented systems: models, patterns, and tools.* Adison-Wesley, 1999.

[3] Canfora, G. and Di Penta, M.: *Service-oriented architectures testing: A survey.* Software Engineering. Springer, pp. 78-105, 2009.

[4] Dijkman, R. and Dumas, M., "Service-Oriented Design: A Multi-Viewpoint Approach". In: *Int. J. Cooperative Inf. Syst.* 13(4), pp. 337–368. World Scientific, 2004.

[5] Forrester, *Enterprise and SMB software survey, North America and Europe, Q4 2008*. Forrester Research, 2008.

[6] Forrester, *The Forrester Wave: Functional Testing Solutions, Q2 2006.* Forrester Research, 2006.

[7] Gao, J.Z., Tsao, H.S.J., and Wu, Y., *Testing and Quality Assurance for Component-Based Software*, Artech House, 2003.

[8] Jorgesen, P.C., *Software Testing: A Craftsman's Approach*, 3rd Edition, Auerbach Publications, 2008.

[9] Murphy, T., *Using Continuous Build to Drive Quality*. Gartner Research Report No. G00166848, 2009.

[10] Thomson, C.D., Holcombe, M., and Simons A., "What Makes Testing Work: Nine Case Studies of Software". In *Proc. of TAICPART'09*, pp. 167-175, IEEE Computer Society, 2009.

[11] Torry Harry Business Solutions, *Web Services in SOA - Synchronous or Asynchronous?* Whitepaper, THBS, 2006. Online: *http://www.thbs.com/pdfs/sync_or_async.pdf*

[12] Utting, M. and Legeard, B., *Practical model-based testing, a tools approach.* Morgan Kaufmann Publ., 2007.

[13] Web Services Reliable Messaging (WS-Reliable-Messaging), Vers. 1.1. *OASIS Consortiom.* Online at: *http://docs.oasis-open.org/ws-rx /wsrm/v1.1/wsrm.pdf*

[14] Wieczorek, S., Kozyura, V., Roth, A., Leuschel, M., Bendisposto, J., Plagge, D., and Schieferdecker, I, "Applying model checking to generate model-based integration tests from choreography models." In: *Proc. of TESTCOM/FATES'09*, LNCS, Springer, 2009. To appear.

[15] Wieczorek, S., Roth, A., Stefanescu, A., and Charfi, A., "Precise Steps for Choreography Modeling for SOA Validation and Verification". In: *Proc. of SOSE'08*, pp. 148-153. IEEE Computer Society, 2008.