A Fault Taxonomy for Service-Oriented Architecture

Stefan Brüning, Stephan Weißleder and Miroslaw Malek
Humboldt-Universität zu Berlin, Germany
{bruening weissled malek}@informatik.hu-berlin.de

***Abstract*—Service-Oriented Architecture (SOA) is a popular design paradigm for distributed systems today. Its dynamics and loose coupling are predestined for self-adaptive systems. This adaptivity and complexity, however, opens many chances for a failure. Therefore, a clear understanding of the kinds of faults that may occur is necessary for detection, tolerance, and testing, e.g. by fault injection. For this purpose, we present a fault taxonomy for service-oriented architecture. Starting with a definition of service and SOA, we describe and analyze different structures and the process of service invocation with a five step model. Furthermore, we describe possible faults and define SOA-specific faults as a new fault class that extends the known fault set for distributed systems. The application of our fault taxonomy is demonstrated with the use case of an online airline reservation system. We conclude with the benefits of our fault taxonomy for designing and testing SOA-based systems.**

***Index Terms*—SOA, dependability, testing, fault taxonomy, fault injection**

I. Introduction

Computer-based infrastructures are a necessity for many companies to handle their daily work today [1]. A big part of their turnover is dependent on the system's availability, reliability, safety, security or short: its working ability. Downtime of the system is costly because it slows down the business or brings it to a halt. Thus, the companies strongly rely on the correctness of the underlying software and hardware. This confidence needs some backup, i.e. justification for the reliance on the system. Some means to verify correctness of the system or to strengthen the company's trust in the system are formal methods like static analysis or model checking but they usually do not scale well to industrial complexity levels so mostly testing is used in practice.

Additionally, the information system infrastructure of most companies is based on distributed systems, which consist of multiple independent computers connected by a network. A common design paradigm for distributed systems is Service-Oriented Architecture (SOA). In SOA, service providers and service consumers are loosely coupled. New services can be discovered and used on the fly using service brokers such as Universal Description, Discovery and Integration (UDDI). Another important aspect about SOA is its ability to connect systems running on different platforms and to react to changes in the system infrastructure. Connecting different platforms through a common communication technology increases the interoperability among systems. Dynamic binding of services enables their self-adaptivity and self-management. Nevertheless, all these options do not only bear prospects of improved infrastructure - they can also be sources of serious failures.

Therefore, much effort has to be put in detecting, analyzing and handling faults while testing the system. The running system is subject to the same caution. As we already stated, the technology of SOA brings many possibilities of dealing with different servers, clients or services. We have to categorize the possible faults to be generally aware of them because the corresponding system parts can be faulty. This is very important in guaranteeing a certain level of quality. Finally, this is not just important for ordinary applications - it is also critical for security-relevant tasks.

For verification issues, there are many formal methods available. However, because services and software are most often informal and the complexity of the distributed system is too high, most formal approaches fail here. So, we resort to testing. One wide-spread technique in this area is fault injection. Therefore, we need to know about the possible types of faults. In this paper, we list several steps that are essential for SOA and use them to build up a SOA-specific fault taxonomy. With knowledge about possible faults, we can test our services by observing their interaction with faulty services that are created via fault injection. Thus, we can stepwise improve the system's robustness already at the development stage and we can test the single components more thoroughly.

Another way to increase availability and reliability of SOA components is to refine possible reactions to faults occurring while the system is operating. In general, this can be done either through redundancy in space (e.g. duplication of systems) or redundancy in time (e.g. detecting faults and exception handling). Either way, the detection of faults and the knowledge about these faults are essential. Therefore, we present a fault taxonomy for service-oriented architectures, which is sub-divided into five steps according to the SOA service invocation process. For each of these steps, possible faults are shown and supported by an example.

The paper is organized as follows: Related work is discussed in Section 2. In Section 3, we justify the necessity of testing. The following Section 4 gives a definition of SOA with a conceptual view on it with five steps required to do a service invocation in an SOA. In the 5th Section, we describe the fault taxonomy which is followed by an example application for our taxonomy in Section 6. Finally, we present conclusions and an outlook on future work in Section 7.

II. Related Work

Fault injection is a common method to test computer system for fault tolerance and reliability [2] [3]. In literature about distributed systems, the focus is usually set on one kind of fault: hardware, software, or network. Cristian [4] proposed a fault taxonomy for computer systems which was extended by Laranjeira et al. [5] and applied to parallel and distributed systems. Hayes [6] developed a fault taxonomy for the NASA used in the development for the ISS project. His taxonomy is NASA-specific but can be generalized to be applied to any other software system. The experiences gained from using the taxonomy are described as well. In our work, we focus on SOA and introduce a new class of SOA-specific faults and define a fault taxonomy for services.

Many methods and technologies have been developed to describe and analyze hardware faults. Hardware architectural issues are discussed, in [7], [8]. The authors describe hardware faults and fault tolerant distributed systems. Arlat et al. [9] describe levels of abstraction for fault injection and make use of different models. Hardware faults can be the root cause for faults in service-oriented architecture as they cause crashes or execution faults.

Software faults are difficult to describe as they cause side effects that cannot be foreseen and models are often incomplete. A short section tackling that topic can be found in [8] where the authors claim that software faults often interact and errors are grouped together. Madeira et al. [10] classify software faults into assignment, checking, interface, timing/serialization, algorithm, and function faults. They use these types to emulate software faults by fault injection. Li et al. [11] examine the effect of software to the risk of failure of a system. They compiled a taxonomy and provide a validation. Bondavalli et al. are assessing and reducing the cost of software fault tolerance in [12]. Some software faults in services cannot be discovered before execution. This makes them impossible to detect for service discovery and composition engines. Beizer [3] collects statistics from various software projects. He deduces a fault taxonomy subdividing the reasons for faults in requirements, features, system structure, input data, software coding, interfaces, integration, and testing. Beizer focuses on all possible sources of error. Instead, we concentrate just on observable behavior because our system should be able to react automatically.

Network faults are well described and analyzed. Nowadays, computer networks are packet based. Possible faults range from routing, missing, or abandoned packets caused by congestion via overflow and wrong routing to authorization conflicts. Some network protocols like TCP already provide fault tolerance. TCP provides a reliable connection which guarantees correct transmission of data. Applications can use these protocols and therefore do not have to handle these faults themselves. As application in SOA are distributed applications that are connected by a network, all network faults will effect the service execution.

In the area of fault injection for web services the focus usually is on altering the Simple Object Access Protocol (SOAP) [13] messages exchanged among client and provider. Looker and Xu are assessing the dependability of web services in [14]. Faults are injected into the SOAP XML files and the dependability of the web services is assessed at the network level.

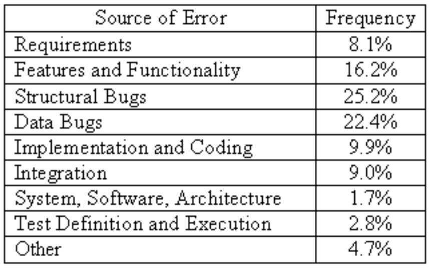
III. Necessity of Testing

Software testing is one of the most important techniques to find faults or increase the dependability, i.e. the programmer's and user's faith in the correctness of the tested system. There are many formal approaches to guarantee a system's correctness like static analysis or model checking. But their application is limited because they assume a formally specified system. Unfortunately, most systems are described only informally. The corresponding problems are described in [3] by the following statements: 1) We can never be sure that our specification is correct. 2) No exerciser can verify every correct program. 3) We can never be sure, that the exerciser is correct.

Thus, we rely on testing and therefore we have to ensure that our test suite covers a big part of the possible fault classes. This is usually achieved by using test coverage criteria that describe certain criteria that need to be fulfilled by the test suite. For instance, the most important control-flow based coverage criteria are statement coverage, branch coverage, or Modified Condition / Decision Coverage (MC/DC) [15]. These can almost exclusively be used for white-box testing or model-based black-box testing. Details are omitted here but more information can be found in [3], [16], [17]. There are also already some commercial tools that support automated test case generation [18], [19], [20], [21]. However, we follow another way to guarantee a certain quality of the test suite: First, we identify possible or expected faults following our taxonomy. Then, we inject the faults in the system, and, finally, we design test cases that are specially designed to find these faults. Obviously, the success of such test cases strongly depends on the quality of the underlying fault taxonomy.

Beizer [3] gathers information about reasons for faults from various software projects. The faults can be found in requirements, features, system structure, input data, software coding, interfaces, integration, or testing. Beizer focuses on every possible source of error, whereas we just focus on the observable errors. We restrict our taxonomy because the system should be able to react automatically to such errors and it can only react to things it can observe. Furthermore, Beizer also weighted these sources of errors according to the frequency in which they occur (see Fig.1). Structural faults, functionality faults, implementation and integration faults are very important for SOA and together they reach a frequency of about 60%.

Once we obtain such a taxonomy, we enable testers to check the robustness of distributed systems by systematically injecting all the described faults (*fault injection*) and watch the consequences on the system behavior. In this paper, we present a corresponding fault taxonomy for web services.



Source of Error	Frequency
Requirements	8.1%
Features and Functionality	16.2%
Structural Bugs	25.2%
Data Bugs	22.4%
Implementation and Coding	9.9%
Integration	9.0%
System, Software, Architecture	1.7%
Test Definition and Execution	2.8%
Other	4.7%

Fig. 1. Fault frequency by Beizer

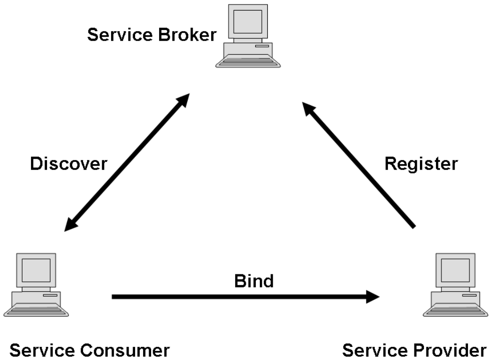


Fig. 2. SOA structure

IV. Service-Oriented Architecture

Service-Oriented Architecture ([22], [23], [24]) is widely accepted as design paradigm for distributed systems. The Web Services standard [25] uses the Web Service Description Language (WSDL) [26] to describe services and SOAP for communication with services.

A. SOA Structure

The basic assumption of SOA is that there are many consumers that require services. In literature, consumers are also referred to as clients or customers. These terms are used interchangeably here. On the other side, there are many providers that provide services on the network. These two groups have to be linked together in a dynamic and adaptive way. This is usually done by a service broker [27], [28].

Service providers register their services at the broker, service consumers request a service from the service broker, which returns a known provider for the requested service. Consumer and provider agree on the semantics. The consumer then binds himself to the service provider and uses the service. The structure of this architecture is shown in Fig. 2.

However, this is not the only possible architecture for a SOA. The Adaptive Service Grid (ASG) project [29] sees SOA as a grid which handles all service discovery and binding. Consumers ask the grid for a specific service. The grid then tries to find the specified service among its registered service providers. In case it cannot find the service the grid tries to compose the service out of the available ones based on the semantic description. The only interaction partner for consumers respectively service providers is the grid (Fig. 3).

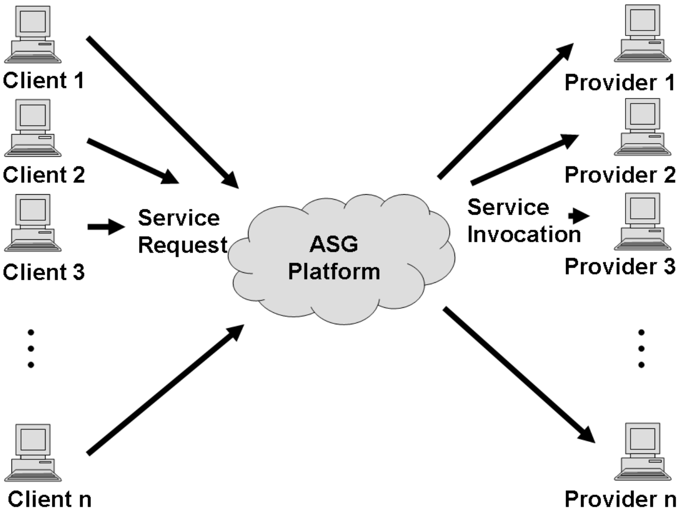


Fig. 3. ASG platform structure

In the end, all SOA architectures provide a way to dynamically link service consumers and service providers. Therefore, SOA basically consists of five steps: service publishing, service discovery, service composition, service binding, and service execution. These steps are explained in the next section.

B. SOA Steps

A clear understanding of all steps for SOA is necessary to establish a comprehensive fault taxonomy of possible faults. Essentially, all five steps have to be undertaken each time a service is executed. Following this paradigm is necessary for the loosely coupled components required for SOA. However, the first steps before the service execution can be "cached". For faster service delivery, they do not have to be repeated each time. However, this would make the quick reaction to changing service environments impossible, i.e. caused by new service providers or crashed services.

1) Publishing: Service providers offer services on the network. They have to provide the services and the corresponsing service descriptions. All services have to be self-descriptive.

Web services are described by WSDL documents. The first version of WSDL was insufficient for a full SOA as it covers the functional interface specification only. A full SOA service description has to cover all service properties and semantics. For instance, it may be a textual description which results in a loss of the capability to automatically discover and compose services. Therefore, structured approaches like WSDL extensions are used for the description of *non-functional properties*. Ontology languages like OWL-S [30] are also used for the semantic description.

2) Discovery: A consumer looking for an appropriate service to perform a desired task has to discover a matching service among all the service providers. He must have a clear understanding of the required service in form of a service description as it is offered by the service providers. Subsequently, the description of the desired service and the description of the offered service have to be compared. If they match, the required service has been successfully discovered.

3) Composition: Perhaps no service matching to the consumer's needs exists. In this case, it may still be possible to compose the service out of the existing ones. Service composition describes the combination of two or more services to one complex service either by service choreography or service orchestration. Functional and non-functional properties have to be taken into account to create the composition [31]. The result of service choreography and service orchestration is the same: a new service providing a more complex functionality. The approaches however, are quite different.

Service Choreography describes the composition of services by defining rules for the collaboration of services (declarative). There is no central instance controlling the composition process. The rules within the service environment describe the externally observable behavior (compare [32], [33], [24]). An attempt to describe service composition using choreography is the Web Service Choreography Interface (WSCI) [34], [32].

Service Orchestration describes the collaboration of services controlled by an external component called orchestration engine. This engine sees the services from an external perspective and knows the rules to compose them (compare [31], [33], [24]). The Business Process Execution Language (BPEL) uses this approach [35].

4) Binding: After discovering an appropriate service delivering the desired functionality, the consumer service binds itself to the service for the execution. At this point, side aspects of the service can be set. Security and AAA (authentication, authorization, and accounting) issues fall into this category.

5) Execution: Once the service is bound to a consumer, the service or the service composition can be executed. The service input parameters are transmitted to the service provider, the service is executed and the output parameters are returned to the consumer.

C. Shortcomings of Current SOA Implementations

The most commonly used implementation of SOA is the web services standard. Web services are described by WSDL files, discovered by UDDI, and communication is usually done via SOAP. Composition usually works using BPEL or web service choreography. However, these standards do not quite fulfill the promise of a dynamic system with loosely coupled components.

WSDL files usually contain the functional specification of the service only. Therefore, UDDI search requests allow service discovery based on this specification. Without further information about the service, the actual service actions and their circumstances (semantics and contract) remain unclear. Several extensions to the WS standard cover other properties but no common basis exist.

This leads to the next problem: automatic service composition. Without the essential information about the service's actual implementation, automatic composition is impossible. Until now, only semi-automatic and manual approaches exist (e.g. BPEL).

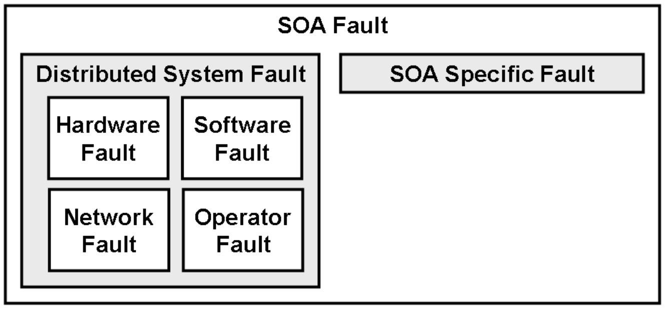


Fig. 4. Fault classes

The need for further information about the service is usually circumvented by using only services that are provided by the developer team that also implements the client. This limits the use of the SOA extremely as these services are rather closely coupled and no dynamic service binding is done.

V. Fault Taxonomy

Now, faults may occur during all the steps of the SOA process. If detected, they will cause errors which will lead to a failure unless the SOA structure is capable of handling those errors, e.g. service crashes can be handled by duplicating those services or using an equivalent service by a different provider.

Besides the SOA specific faults, all presented faults can occur in a distributed system. These are hardware faults, software faults, network faults, and faults caused by the operator (Fig. 4).

While these faults apply to distributed systems in general, this paper focuses on typical faults applying to SOA. So, we leave all general elements of distributed systems aside and focus solely on the essential steps of SOA and all faults correlated to them. The other faults are described in more detail in the related work.

SOA faults apply to service-oriented architectures only, independently of the used technology (e.g. web services). As described in section IV, SOA implementations dynamic link service providers and consumers, which leads to a loosely coupled distributed system. This dynamic linking allows SOA implementations to adapt to quickly changing environments and situations. Of course, this dynamic behavior brings in new sources of failures and possible faults. Faults can occur in all of the described five steps. Our taxonomy starts very general but is refined within each category. This generalization allows a complete coverage of all possible faults. Of course, the taxonomy has to be refined for each domain to allow the system to also react to the domain-specific faults in a reasonable way and allow fault detection. Fig. 5 gives an overview of possible faults. They are explained in the following sections. The names printed in italic correspond to the names in the figure.

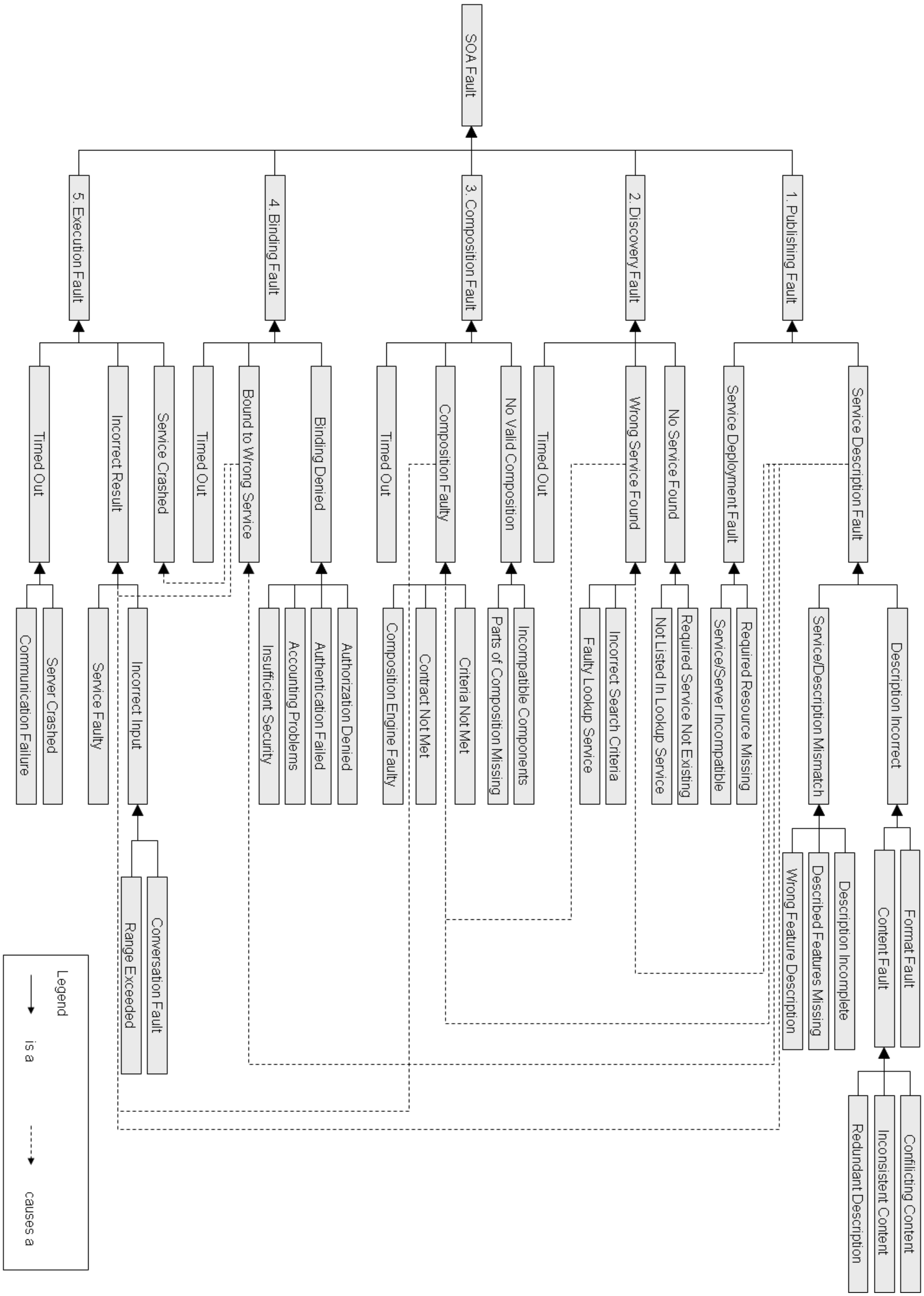


Fig. 5. Taxonomy of SOA-specific faults

A. Publishing Faults

During publishing, the service is deployed on a server so it can be executed and the service description is made public. *Service Description Faults* are similar to specification faults that occur when the description of the service is incorrect. The description may be faulty itself (*Description Incorrect*) or it just may not match the deployed service (*Service Description Mismatch*). Faults caused by an incorrect description can be detected by checking the description file only. It is a *Format Fault* when the format of the description is incorrect. For instance, a XML file may be not well formed because it misses some tags. The content may be wrong (*Content Fault*) if, for example, prescribed security algorithms do not exist at all.

The deployed service may not match the provided description. There is an *Incomplete Description* if the service provides more features than are published ones in the description. If the description mentions features, which are not provided by the deployed service, these are called *Missing Features*. If the feature described does not match the feature actually provided it is an *Incorrect Feature Description*.

Service Deployment Faults occur when the service is not successfully deployed on the target platform. If the software versions of service and server do not match and therefore the execution will be not possible or limited there is an *Incompatibility of Service/Server*. In case the service is *Missing a Required Resource* (e.g. a link to a data base), the service may be deployed successfully but will fail to perform.

Format faults can be detected using verification techniques, e.g. XML checkers. Content faults can be detected by validating using predefined criteria. Deployment faults cannot be detected before the execution. So, only tests can reveal them.

B. Discovery Faults

Faults during discovery can happen either on search invocation or returning the found services.

The relatively easy fault to detect is *No Service Found*. This may be for several reasons: Either the *Required Service* is *Not Existing* or it is *Not Listed In Lookup Service*.

A *Wrong Service Found* fault will be difficult to detect. The fault can only be detected in Step 5 when the service is really executed. The found service may be wrong if either the client specified *Incorrect Search Criteria*, or there is a *Faulty Lookup Service*, or the provided specification does not match the actual provided service.

Any step from 2 to 5 may *Time Out*. This can be due to a *Server Crash* which can be caused by hardware or software or a *Communication Failure* which is a network fault. For simplification, this is only shown in Step 5.

C. Composition Faults

The ability to compose services to create new services is an essential part of SOA. If the composition process fails the composition engine will return *No Valid Composition*. This can be due to various reasons. There may be *Incompatible Components* that cannot be connected. For instance, two location based services use different coordinate systems for specifying the position. If *Parts of the Composition Are Missing*, required services to translate between services are missing. For example, a converter service from Euro to Dollar may be missing.

If the composition engine is unable to detect the faults during composition, the returned composition may not meet the specified requirements (*Composition Faulty*). This may either be because certain properties are not supported by all parts of the composition. For example, security may only be guaranteed by the first and last service, but not in between. Then, *Criteria* are *Not Met*. Or, the used *Composition Engine is Faulty* and does not produce the desired service composition. Another possibility is if preconditions, postconditions, or invariants are not fulfilled. Then, the contract of the service is violated (*Contract Not Met*).

D. Binding Faults

During binding, the service consumer and service provider negotiate the conditions to execute the service. The *Binding* may be *Denied* if the AAA component denies the access. That is, either *Authorization Denied*, *Authentication Failed*, or *Accounting Problems* occur. *Insufficient Security* may also be a reason, e.g. one side does not trust the certificate of the other.

Also, the client may be bound to the wrong service without noticing. This can either be caused by a wrong service description, failed service discovery, or even malicious methods like spoofing.

E. Execution Faults

Execution faults occur when the service is executed but the result does not match the expected outcome.

The service may just crash. The server will usually notice the *Crashed Service* and notify the client about the failure.

If the service delivers an *Incorrect Result* this can be either due to a software fault (*Service Faulty*) or *Incorrect Input*. Incorrect inputs may be caused by a *Conversion Fault* or the *Input Range* may be *Exceeded*.

VI. An Example

We demonstrate the application of our taxonomy with a typical example of a travel agency. In this example, a customer books a trip using a travel agency service. His trip booking consists of a flight reservation to his destination, a rental car to be picked up at the airport, and the hotel reservation. This service is usually realized by one travel agency service that uses three other sub-services. As we want to show the benefits our SOA-specific fault taxonomy, we focus only on one part of the whole example and describe this part in more detail. We chose the airline reservation service (Fig. 6). All listed faults can likewise occur in the other two services.

Suppose, a traveler wants to fly from London to Paris. Consequently, he needs to find an appropriate flight and has to pay for it. He specifies the date and time he wants to fly and provides his credentials (e.g. name, credit card number). The

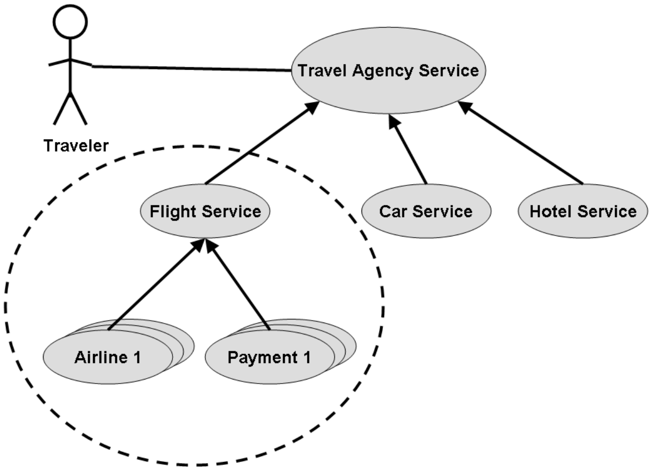


Fig. 6. Travel agency example with focus on airline reservation service

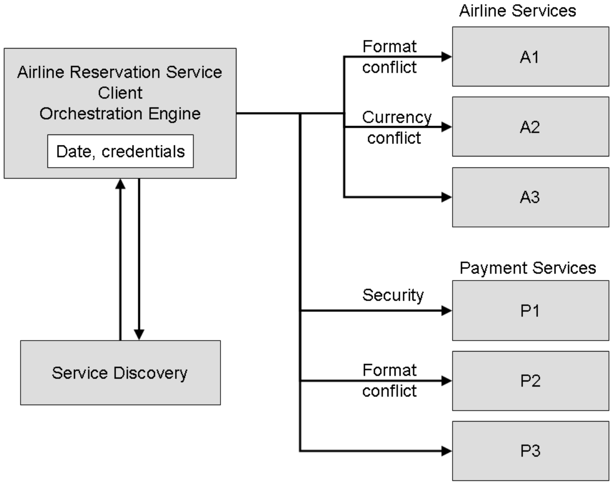


Fig. 7. Airline reservation service

airline reservation service will then try different airlines until it finds an appropriate flight and then call a payment service.

The traveler enters his personal information into a form on a web server he accessed using a standard browser. In our context, this web server has two roles: it is the client that uses web services to perform the booking and it is the service orchestration engine as it composes the airline and payment services (see Fig. 7).

First, the *Airline Reservation Service* has to find an appropriate flight. The discovery service returns three different airline reservation services. The first returned service (A1) is executed without problems but returns no available flights. Apparently, this is wrong. This fault was caused by an incorrect date format: The airline did offer some flights but the service specified the format of the date with (DDMMYY) but the actual implementation required the format (YYMMDD). In our fault taxonomy, this is a *Service/Description Mismatch*.

The second airline reservation service (A2) is checked for a valid composition with a payment service. Here, our test person is not able to pay in English Pound Sterling £. To find a valid service composition, the orchestration engine tries to find a currency converter service. As there is no currency converter service, the engine is unable to use this service and temporarily generates a *No Valid Composition*, or more specific, a *Parts of Composition Missing* error.

Finally, the third airline (A3) offers a matching flight. Additionally, we need a payment service to pay for the flight. Again, our orchestration engine can choose between three different services. While building up a secure channel, the first payment service (P1) denies the binding because it does not trust the security certificates provided by the airline service (*Insufficient Security* fault).

The second payment service (P2) accepts the certificate but returns an error during execution because the parameter specified cannot be interpreted. Instead of a decimal point, the amount of Euro and Cent is separated by a comma. This is a fault during specification (*Service/Description Mismatch*) that cannot be discovered before the actual service execution and therefore causes an *Incorrect Input* fault.

The third payment service (P3) finally accepts the credentials of our person and the flight can be booked and charged.

As we showed, the presented typical faults in such a scenario are covered by our fault taxonomy. Consequently, we can now test our web services more thoroughly for these faults. Next, we can develop services which can react to such faults properly.

VII. Conclusions and Outlook

SOA is a popular approach in industry and science to design distributed systems. However, the promises of SOA to allow loose coupling, dynamic discovery, and composition of independent services from different sources have not been quite fulfilled. Several WSDL extensions and ontology approaches aim to fill the gap but have not found their way into typical SOA implementations at this time. It is essential for a truly dynamic SOA that the properties, semantics, and functional aspects of services are fully described.

The process of service invocation in SOA has five different steps that have to be taken to allow dynamic service discovery and loose coupling. As in distributed systems, faults can occur at each one of those steps. We proposed a fault taxonomy for a systematic description of possible faults. Additionally, we stated which faults can cause other ones. This knowledge is essential for testing and building dependable systems as well as for testing the system via fault injection.

Due to the complexity of SOA systems, testing is the commonly used support measure for high assurance of reliability, availability, and security. Testing via fault injection aims at covering as many fault classes as possible with as few test cases as possible. Our fault taxonomy facilitates such approach by abstracting from concrete implementations to a general understanding of SOA and the steps that are required for service invocation.

An important aspect is the detectability of the described faults. This aspect is getting even more difficult because some faults can be detected by syntax checks (e.g. description mismatches), whereas other faults can only be detected during runtime. Fault injection is a valid approach to examine the fault detection mechanisms. The main benefit of our work is the identification of typical possible SOA-specific faults.

Next steps in our work will be the setting up of a test environment and the development of appropriate fault injection mechanism (e.g. description manipulators, SOAP message interception mechanisms). This environment will be used to deploy a feasible test scenario with multiple service providers, clients, and service invocation scenarios. Subsequently, the behavior of existing SOA structure like ASG can be examined and evaluated.

REFERENCES

[1] M. Papazoglou and P. Ribbers, *e-Business: Organizational and Technical Foundations*. Chichester: John Wiley & Sons, 2006.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 01, no. 1, pp. 11–33, 2004.

[3] B. Beizer, *Software Testing Techniques*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[4] F. Cristian, "A rigorous approach for fault-tolerant programming," *IEEE Transaction on Software Engineering*, vol. 11, no. 1, pp. 23–31, January 1985.

[5] L. Laranjeira, M. Malek, and R. Jenevein, "Nest: A nested-predicate scheme for fault tolerance," *IEEE Transactions on Computers*, vol. 42, no. 11, pp. 1303–1324, 1993.

[6] J. H. Hayes, "Building a requirement fault taxonomy: Experiences from a nasa verification and validation research project," in *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*. Denver, CA, USA: IEEE Computer Society, 2003, p. 49.

[7] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[8] D. P. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, 3rd ed. Digital Press, 1992.

[9] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, 1990.

[10] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," in *Proceedings of the International Conference on Dependable Systems and Networks*. New York, NY, USA: IEEE, June 2000, pp. 417–426. [Online]. Available: citeseer.ist.psu.edu/madeira00emulation.html

[11] B. Li, M. Li, K. Chen, and C. Smidts, "Integrating software into pra: A software-related failure mode taxonomy," *RISK ANALYSIS*, vol. 26, no. 4, pp. 997–1012, August 2006. [Online]. Available: http://www.blackwell-synergy.com/doi/abs/10.1111/j.1539-6924.2006.00795.x

[12] A. Bondavalli, F. Di Giandomenico, and J. Xu, "A cost-effective and flexible scheme for software fault tolerance," *Journal of Computer Systems Science and Engineering*, vol. 8, no. 4, pp. 234–244, 1993, cRL Publishing.

[13] W3C, "SOAP version 1.2 part 0: Primer," June 2003. [Online]. Available: http://www.w3.org/TR/soap12-part0/

[14] N. Looker and J. Xu, "Assessing the dependability of SOAP RPC-based web services by fault injection," in *Object-Oriented Real-Time Dependable Systems, 2003. Proceedings. Ninth IEEE International Workshop on*, 1-3 Oct. 2003, pp. 163–170.

[15] J. Chilenski and S. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing," in *Software Engineering Journal, Sept. 1994, Volume 9, Issue*, 1994, pp. 193–200.

[16] R. V. Binder, *Testing object-oriented systems: models, patterns, and tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[18] MIT, "Korat - a framework for automated testing of java programs." [Online]. Available: http://mulsaw.lcs.mit.edu

[19] Reactive Systems Inc., "Reactis." [Online]. Available: http://www.reactive-systems.com

[20] Telelogic, "Rhapsody automated test generation (ATG)." [Online]. Available: http://www.telelogic.com/products/rhapsody

[21] open source, "ModelJUnit." [Online]. Available: http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/

[22] S. Burbeck, "The tao of e-business services," 2000. [Online]. Available: http://www-128.ibm.com/developerworks/webservices/library/ws-tao/

[23] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications*, ser. Data-Centric Systems and Applications. Berlin: Springer, 2003.

[24] F. Leymann, "Workflow-based coordination and cooperation in a service world," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, ser. Lecture Notes in Computer Science, vol. Volume 4275/2006. Springer Berlin / Heidelberg, 2006.

[25] W3C, "Web services," 2007. [Online]. Available: http://www.w3.org/2002/ws/

[26] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, "Web services description language (WSDL) 1.1," March 2001. [Online]. Available: http://www.w3.org/TR/wsdl

[27] W3C, "Web services architecture," Febuary 2004. [Online]. Available: http://www.w3.org/TR/ws-arch/

[28] G. Denaro, M. Pezzé, D. Tosi, and D. Schilling, "Towards self-adaptive service-oriented architectures," in *TAV-WEB '06: Proceedings of the 2006 workshop on Testing, analysis, and verification of web services and applications*. New York, NY, USA: ACM Press, 2006, pp. 10–16.

[29] D. Kuropka, A. Bog, and M. Weske, "Semantic enterprise services platform: Motivation, potential, functionality and application scenarios," *edoc*, vol. 0, pp. 253–264, 2006. [Online]. Available: http://asg-platform.org

[30] DAML Services. (2004, 11) OWL-S 1.1 specification. [Online]. Available: http://www.daml.org/services/owl-s/1.1/

[31] N. Milanovic, "Contract-based web service composition," Ph.D. dissertation, Humboldt-Universität zu Berlin, 2006.

[32] W3C, "Web services choreography requirements." [Online]. Available: http://www.w3.org/TR/ws-chor-reqs/

[33] lad Tanasescu, A. Gugliotta, J. Domingue, L. G. Villarías, R. Davies, M. Rowlatt, M. Richardson, and S. Stincic, "Spatial integration of semantic web services: the e-merges approach," in *ISWC*, Athens, GA, 2006.

[34] W3C. (2002, August) Web service choreography interface (WSCI) 1.0. [Online]. Available: http://www.w3.org/TR/wsci/

[35] OASIS. (2007, April) Web service business process execution language version 2.0. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf