A Structured Approach to Architecting Fault Tolerant Services

Elena Troubitsyna, Kashif Javed
Åbo Akademi University, Finland
Elena.Troubitsyna@abo.fi, Kashif.Javed@abo.fi

***Abstract*— Service-oriented computing offers an attractive paradigm to designing complex composite services by assembling readily-available services. The approach enables rapid service development and significantly increases productivity of the development. However, it also poses a significant challenge in ensuring quality of created services and in particular their fault tolerance. In this paper, we propose a systematic approach to architecting complex fault tolerant services. We demonstrate how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We propose an approach facilitating a systematic analysis of possible failures of the services, recovery actions and alternative solutions for achieving fault tolerance. Our approach supports structured guided reasoning about fault tolerance at different levels of abstraction. It allows the designers evaluate various architectural solutions at the design stage that helps to derive clean architectures and improve fault tolerance of developed complex services.**

Keywords - services; fault tolerance, architecture, service composition, service orchestration; failure modes and effect analysis

I. INTRODUCTION

Web-services [13] constitute one of the fastest growing areas of software engineering. With a strong support for compositionality, the process of developing an application essentially becomes a process of composing available services. Services – the basic building blocks of complex applications are platform and network independent components implementing computations that can be invoked by clients or other services.

To enable a rapid service composition, services define their properties in a standard and machine readable format. It enables service discovery, selection and binding. Service composition introduces the orchestration of the basic services to build applications. However, usually research on service orchestration focuses on defining the language for service composition that does not support reasoning about such essential features as fault tolerance. Such reasoning can be supported by dependability analysis and architectural modelling [5].

In this paper, we propose a systematic approach to architecting fault tolerant services. We demonstrate how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We propose static and dynamic solutions for introducing fault tolerance into the service composition. The structural solutions rely on availability of redundant service providers that can be requested to provide services in case of failures of the main service providers. This mechanism allows the designers to mask failures of the individual service providers. The dynamic solutions rely on re-execution of failed services to recover from the transient faults of services. This solution requires modifications of the service execution flow.

To facilitate design of complex fault tolerant services, in this paper, we introduce a systematic approach to analysing possible failure modes of services and defining fault tolerance measures. Our approach is inductive – it progressively analyses one component after another in the service execution flow, explores possible fault tolerance alternatives and systematically introduces them into the service architecture.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space. It allows the designers to evaluate various architectural solutions at the design stage that helps to derive clean architectures and improve fault tolerance of developed complex services.

The paper is structured as follows: in Section II, we demonstrate how to model a fault tolerant service from a service user's perspective. In Section III, we demonstrate how to unfold service architecture, i.e., explicitly represent the service composition and the service execution flow. We also propose different fault tolerance mechanisms that can be introduced to enhance fault tolerance. In Section IV, we introduce a structured approach to designing a fault tolerant architecture. Finally, in Section V, we overview the related work and discuss the presented work.

II. ABSTRACT MODELING OF FAULT-TOLERANT SERVICES

The main goal of introducing fault tolerance in the service architecture is to prevent a propagation of faults to the service interface level, i.e., to avoid a service failure [7] [9]. A fault manifests itself as *error* – an incorrect service

 ISBN: 978-1-61208-361-2

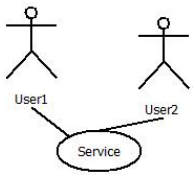

Figure1. Use case representation of a service.

state [9]. Once an error is detected, an error recovery should be initiated. Error recovery is an attempt to restore a fault-free state or at least to preclude system failure.

Error recovery aims at masking error occurrence or ensuring deterministic failure behaviour if the error cannot be masked. In the former case, upon detection of error, software executes certain actions to restore a fault-free system states and then guarantee normal service provisioning. In the latter case, the service provisioning is aborted and failure response is returned.

In this paper, we focus on the architectural graphical modelling [12] of fault tolerant services [13]. We demonstrate how to explicitly introduce handling of faulty behaviour into the service architecture. We follow the model-driven development paradigm and start our modelling from a high level of abstraction [8]. The consecutive model transformations introduce the detailed representation of the service architecture.

The high-level model of a fault tolerant service is given in Fig.1. The service is defined via its interactions with different service users. Each association connecting an external user and a service corresponds to a logical interface, as shown in Fig.2. The logical interfaces are attached to the class with ports. At the abstract modelling level, we treat a service as a black box with the defined logical interfaces.

The UML2 interfaces *I_ToService* and *I_FromService* define the request and request parameters of the service user. We formally describe the communication between a service and its user(s) in the *I_Communication* state machine as illustrated in Fig.3. The request *ser_req* received from the user is always replied: with the *ser_cnf* in case of success, with the *ser_fail_cnf* in case of unrecoverable failure and with the *ser_tfail_cnf* in case of a recoverable failure. Let us point out, that already at the abstract level of modelling, we explicitly introduce representation of faulty behaviour and reaction on it.

To exemplify an abstract modelling of a fault tolerant service, let us consider a *positioning service*. It provides the services for calculating the physical location of the service user.

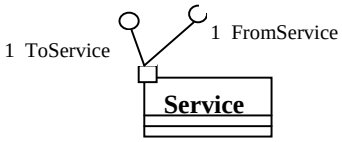

Figure 2. Abstract architectural diagram.

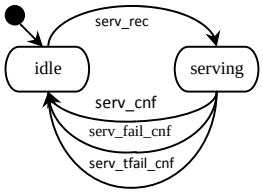

Figure 3. State diagram of communication.

As shown in Fig.4, the abstract model represents an interaction of the service with a user. An abstract architectural diagram defines an interface for communicating with the user. The state diagram formally defines the communication between the user and the service.

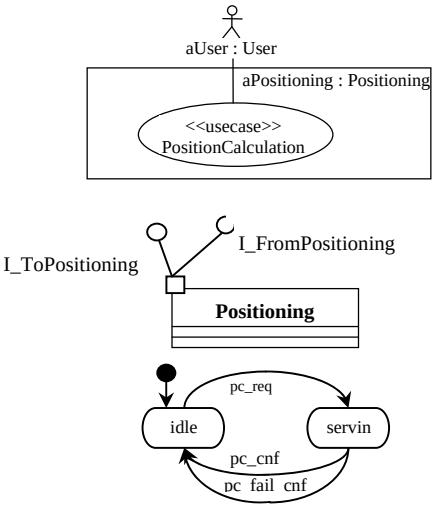

Fig.4. Modelling positioning service

The request to calculate the position is modelled by the event *pc_req*. In case of a normal execution, the positioning service returns the reply *pc_cnf*. Let us observe, that in our modelling we explicitly define the possibility of a service failure following the pattern proposed above. Indeed, in case of the unrecoverable failure, the positioning service returns *pc_fail_cnf*. In case of a recoverable failure, the service returns *pc_tfail_cnf*. Such a fault-tolerance explicit approach to modelling ensures that the service execution always terminates, i.e., the service never becomes unresponsive.

III. ARCHITECTURAL DECOMPOSITION

Our abstract modelling has defined the service from the service user's point of view. The model transformation presented next focuses on defining the composition that constitutes the overall service.

An execution of a composite service consists of executing several subservices. Coordination of a service execution is performed by a *service manager* (sometimes

called *service composer*). It is a dedicated software component that on the one hand, communicates with a service user and on the other hand, orchestrates the service execution flow.

To coordinate service execution, the service manager keeps the information about subservices and their execution order. It requests the corresponding service components to provide the required subservices and monitors the results of their execution.

Let us note, that any subservice might also be composed of several subservices, i.e., in its turn, the subservice execution might be orchestrated by its (sub)service manager. Hence, in general, a composite service might have several layers of hierarchy [5].

To model a composite service, we introduce the providers of the subservices into the abstract architectural service model. The model includes the external service

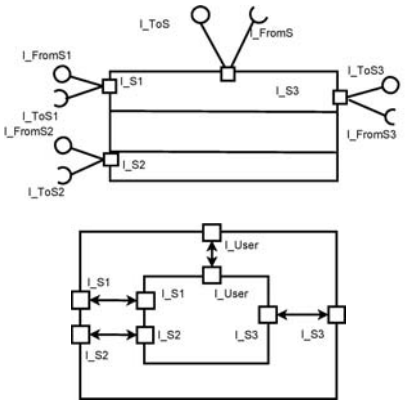

Figure 5. Architecture of a positioning service.

providers communicating with the aggregated service via their service director. For each association between the main service and the corresponding subservice, we define a logical interface. The logical interfaces are attached to the corresponding classes via the corresponding ports. This enables a structured representation of the modular structure of the composite service. The functional architecture is defined in terms of the service components, which encapsulate the functionality related to a single execution stage of another logical piece of functionality.

The architectural diagram of the position calculation [5] [14] – the composite service example described above is presented in Fig. 5. The service manager role is two-fold: it orchestrates service execution flow and handles communication with the service user. The dynamics of the execution flow is refined by introducing the corresponding sub-states in the service state as shown in Fig. 6.

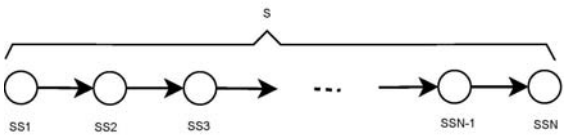

Figure 6. Unfolded dynamic behaviour.

Now, let us discuss the fault tolerant aspect of the composite services. Execution of any subservice can fail. To ensure fault tolerance of composite services, we propose a two-fold approach. On the one hand, we define a set of patterns [11] that allow us to introduce structural means for fault tolerance using various forms of redundancy. On the other hand, we propose to extend the responsibilities of a service manager, to implement dynamic error recovery. Next, we propose the architectural patterns for introducing structural fault tolerance and define the corresponding modeling artifacts.

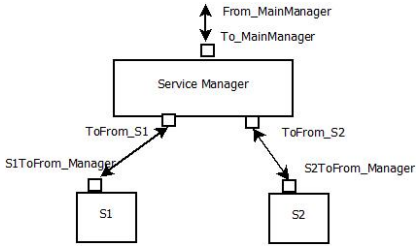

Figure 7. Duplication scheme.

Duplication pattern. The duplication is a simplest arrangement for structural fault tolerance. It can be introduced if there are two service components available that provide the same functionality. In this case, the services can be executed in parallel. A successful execution of a service by any out of two service components suffices for the successful service provisioning.

An architectural diagram of the duplication arrangement is given in Fig. 7. We introduce a dedicated service manager to take care of the execution of the duplicated service. The dynamical behavior of the duplication pattern is shown in Fig. 8. An alternative architectural approach would be to allow the main service manager to orchestrate this arrangement.

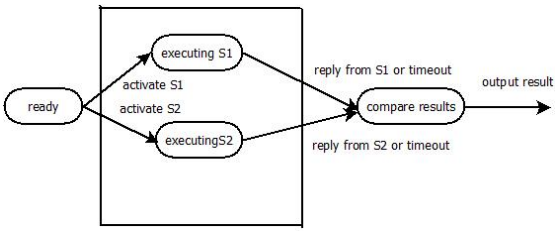

Figure 8. Dynamic behavior of duplication pattern.

Stand-by spare. This arrangement relies on availability of a spare service component implementing the desirable service. The spare is used only if the execution of the service by the main component fails. If the main service component succeeds in executing a service, the spare service component remains inactive. However, if the main service component fails to execute a service then the spare service component is requested to provide the service.

The stand-by spare arrangement can be implemented with and without an introduction of the dedicated service

 ISBN: 978-1-61208-361-2

director. The design decision depends on the complexity of the composite service, i.e., whether the design of the main service manager would become too complex with the introduction of this additional responsibility.

The architecture of the stand-by-spare implemented with the dedicated service manager coincides with the duplication pattern. However, the dynamic behavior is different as shown in Fig.9.

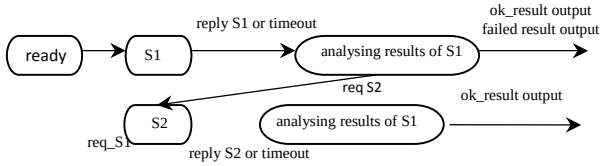

Figure 9. Dynamic behavior of stand-by spare.

Triple modular redundancy pattern. A more complicated scheme for structural redundancy – triple modular redundancy is shown in Fig.10. The precondition for implementing it is that we have three service components available that provide identical services with the same functionality. All three service components receive the same service request and work in parallel. The results of the service execution are sent to a voting element.

The voting element is a dedicated software component that performs comparison of the results and produces the final result. The voting element takes a majority view over the produced results of the successfully executed services and outputs it as the final result of the service execution.

In the context of the service-oriented computing, the voting component might be implemented in two different ways: it might output the results after receiving the first two replies or it might start to act only after the certain deadline when all non-failed services have replied.

Let us discuss a difference between triple modular redundancy scheme adopted in hardware and services. In hardware context, the scheme can mask failure of a single component by adopting the majority view. In the service-oriented context, it gives more fault tolerance options. Indeed, if two out of three services failed to reply within the timeout, the voter component can be design to simply output the result of the non-failed service. Obviously, in case of a failure of a single service, it gives better fault tolerance guarantees, because it can compare the results of two non-failed services and take the one, which is more accurate as the output.

Since the triple modular redundancy scheme has a rather complex architecture by itself, we propose to introduce a dedicated service manager to integrate the arrangement in the architecture of a composite service. The proposal is depicted in Fig. 10.

The dynamic behavior of the triple modular arrangement is depicted in Fig.11. Here, the dedicated service manager performs voting before outputting the service result.

The static redundancy schemes require availability of redundant service components and hence, sometimes,

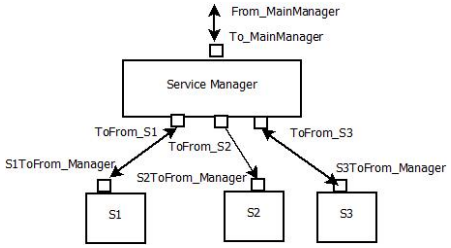

Figure 10. Architecture of triple modular redundancy.

might be non-implementable. However, they provide an efficient means to cope with permanent service failure. In contrast, dynamic fault tolerance relies on service re-execu-

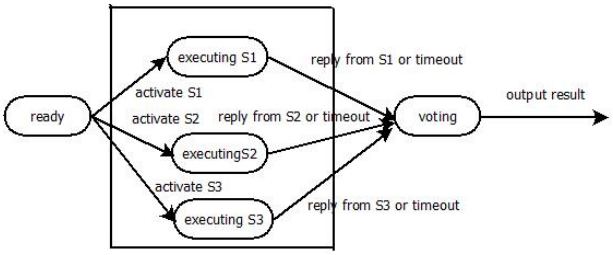

Figure 11. Dynamic behaviour of triple modular redundancy.

tion to increase the chances of the successful service execution and does not require an availability of the redundant service components. Obviously, the dynamic fault tolerance solutions can cope with transient failures.

To leverage fault tolerance of a composite service, the service manager might alter the normal flow of service execution to dynamically cope with failures. For instance, it might repeat service execution, roll-back or abort service execution.

If service execution failed, but the returned exception indicates that the error is transient then by re-executing the failed subservice, the service manager might recover from the error. The service execution flow is shown in Fig.12.

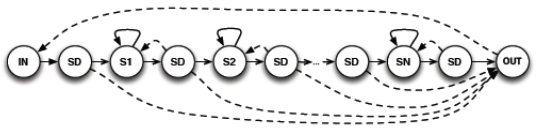

Figure 12. Service execution flow.

If service execution failed but the returned exception indicates that the error is unrecoverable and there are no alternative services available, then the service manager can abort the entire service execution and return failure response.

Obviously, designing fault tolerant composite services is a non-trivial task that requires a systematic support. In the next section, we propose an approach to systematic development of fault tolerant architecture by a structured

 ISBN: 978-1-61208-361-2

analysis of failure modes of the services and fault tolerance schemes.

IV. DEVELOPMENT OF A FAULT TOLERANT SERVICE ARCHITECTURE

The main motivation behind our approach is to facilitate a structured disciplined derivation of fault tolerant service architecture. Essentially, we define the guidelines for analyzing faulty behavior of the services and deciding on the mechanisms for fault tolerance.

Our approach is inspired by the Failure Modes and Effect Analysis (FMEA) technique. FMEA [16] is an inductive analysis method, which allows designers to systematically study the causes of components faults, their effects and means to cope with these faults. FMEA is used to assess the effects of each failure mode of a component on the various functions of the system as well as to identify the failure modes significantly affecting dependability of the system.

FMEA step-by-step selects the individual components of the system, identifies possible causes of each failure mode, assesses consequences and suggests remedial actions. The results of FMEA are usually represented in the tabular form that contains the following fields: component name, failure mode, possible cause, local effect, system effect, detection, and remedial action.

Let us exemplify the proposed approach. Assume that a service *S1* is a part of the composite service *S*. The services *S11* and *S12* have identical functionality. Assume that the service *S1* might experience transient silent failures, i.e., become temporally irresponsive. Such failures can be detected by timeout. Then we can arrange services into a triple modular redundancy scheme. The structured analysis of the fault tolerance arrangement around the service *S1* according to the proposed approach is shown in Table I.

TABLE I. TRANSIENT FAILURE ANALYSIS

Service	S1
Failure mode	Transient silent failure
Detection	Timeout
Available redundancy	S11, S12
Structural redundancy	Triple modular redundancy arrangement. Result is produced upon timeout
Recovery	Masking failure by use of triple modular redundancy arrangement. In case of simultaneous failure of S1, S11 and S12 repeat execution

Let us now assume that a service *S2* is also part of the composite service *S*. Assume that the service *S2* might experience transient failures that are identified by receiving the response *S2_tfail_cnf* from it. Since no redundant service components are available for this case and the service failure is detectable with the corresponding notification, we can rely on dynamic redundancy to cope with failures of S2. The structured analysis of the fault tolerance arrangement around the service *S2* according to the proposed approach presented in Table II.

TABLE II. FAILURE MODE ANALYSIS

Service	S2
Failure mode	Transient detectable failure
Detection	*S2_tfail_cnf* response
Available redundancy	No
Structural redundancy	No
Recovery	Re-execute service. Maximal allowed number of retries is 3.

It easy to observe that reliance on the proposed approach facilitates structured derivation of fault tolerance architecture for both structured and dynamic fault tolerance schemes.

As a result of introducing various means for fault tolerance, we also should modify the design of the service manager. Fig 13 depicts the modified flow with a retry.

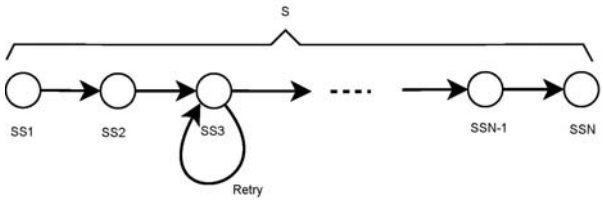

Figure 13. Execution flow with retry.

The process of introducing fault tolerance mechanisms can be iteratively applied to unfold all the architectural layers. As a result of this process, we obtain a hierarchical structure of service managers augmented with fault tolerance properties.

V. RELATED WORK AND CONCLUSIONS

While the topic of service orchestration and composition has received significant research attention, the fault tolerance aspect is not so well addressed. Liang [10] proposes a fault-tolerant web service on SOAP (called FT-SOAP) using the service approach. It extends the standard WSDL by proposing a new element to describe the replicated web services. The client side SOAP engine searches for the next available backup from the group WSDL and redirects the request to the replica if the primary server failed. It is a rather complex mechanism that hinders interoperability.

Artix [2] is IONA's Web services integration product. It provides a WSDL-based naming service by Artix Locator. Multiple instances of the same service can be registered under the same name with an Artix Locator.

 ISBN: 978-1-61208-361-2

When service consumers request a service, the Artix Locator selects the service instance based on a load-balancing algorithm from the pool of service instances. It provides useable services for the service consumers. An active UDDI mechanism [4] enables an extension of UDDI's invocation API to enable fault-tolerant and dynamic service invocation. Its function is similar to the Artix Locator. A dependable Web services framework is proposed in [1]. Once a failure for one specific service occurs, the proxy raises a "WebServiceNotFound" exception and downloads its handler from DeW. The exception handling chooses another location that hosts the same service and re-invoks the method automatically. The main goal of DeW is to realize physical-location-independence. Providing fault-tolerance capability for composite Web service has also been discussed in [3].

A formal approach [15] [17] to introducing fault tolerance to the service architecture has been proposed in [5] [6]. This work extends the set of architectural patterns that can be introduced to achieve fault tolerance as well as propose a systematic support for deriving fault tolerance solutions.

In this paper, we have proposed a systematic approach to architecting fault tolerant services. We demonstrated how to graphically model the architecture of composite services and augment it with various fault tolerance mechanisms. We defined a set of static and dynamic solutions for introducing fault tolerance into the service composition. The proposed mechanisms can cope with different types of failures to increase reliability of complex composite services.

To facilitate design of fault tolerance mechanisms, we proposed an approach to a structured analysis of possible failure modes of services and introducing fault tolerance measures. The proposed approach is inductive – it progressively analyses services in the execution flow, explores possible fault tolerance alternatives and systematically introduces them into the service architecture.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space while architecting complex composite services.

ACKNOWLEDGMENT

Troubitsyna thanks the Need for Speed program. http://www.digile.fi/N4S for a financial support.

REFERENCES

[1] E. Alwagait, S. Ghandeharizadeh, "A Dependable Web Services Framework" 14th International Workshop on Research Issues on Data Engineering 2004. [Online]. Available from http://fac.ksu.edu.sa/alwagait/publication/31143 2014.30.05.

[2] Artix Technical Brief. [Online]. Available from http://www.iona.com/artix 2014.30.05.

[3] V. Dialani, S. Miles, L.Moreau, D. Roure, M. Dialani, "Transparent fault tolerance for web services based architectures". 8th Europar Conference (EULRO-PAR02), Springer 2002, pp. 889-898. ISBN:3-540-44049-6.

[4] M. Jeckle, B. Zengler, "Active UDDI-An Extension to UDDI for Dynamic and Fault Tolerant Service Invocation" 2nd International Workshop on Web and Databases, Springer 2002, pp. 91-99. ISBN:3-540-00745-8.

[5] L. Laibinis, E. Troubitsyna and S. Leppänen, "Service-Oriented Development of Fault Tolerant Communicating Systems: Refinement Approach" International Journal on Embedded and Real-Time Communication Systems, vol. 1, pp. 61-85, Oct. 2010, DOI: 10.4018/jertcs.2010040104.

[6] L. Laibinis, E. Troubitsyna, A. Iliasov, A. Romanovsky, "Rigorous development of fault-tolerant agent systems", In in M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna (Eds.), Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, pp. 241-260, Springer 2006, ISBN 978-3-642-00867-2.

[7] L. Laibinis, E. Troubitsyna, S. Leppänen, J. Lilius and Q. Malik, "Formal Service-Oriented Development of Fault Tolerant Communicating Systems", in M. Butler, C. Jones, A. Romanovsky and E. Troubitsyna (Eds.), Rigorous Development of Complex Fault-Tolerant Systems, LNCS 4157, pp. 261-287, Springer 2006, ISBN 978-3-642-00867-2.

[8] L. Laibinis, E. Troubitsyna "Fault Tolerance in use-case modelling", In Workshop on Requirements for High Assurance Systems (RHAS 05), [Online]. Available from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.4950 2014.01.05.

[9] J. C. Laprie. Dependability: Basic Concepts and Terminology. Springer-Verlag, 1991.

[10] D. Liang, C. L. Fang, C. Chen, F. X, Lin. "Fault-tolerant web service". Tenth Asia-Pacific Software Engineering Conference, IEEE Press, Dec. 2003, pp.56-61, ISBN 973-4-642-01867-1.

[11] I. Lopatkin, A. Iliasov, A. Romanovsky, Y. Prokhorova, E. Troubitsyna, "Patterns for representing FMEA in formal specification of control systems" High-Assurance Systems Engineering Conference (HASE), IEEE Nov 2011, pp. 146 – 151, ISBN 978-1-4673-0107-7.

[12] J. Rumbaugh, I. Jakobson, and G .Booch, The Unified Modelling Language Reference Manual. Addison-Wesley, 1998.

[13] Web Services Architecture Requirements. [Online] Available from http://www.w3.org/TR/wsareqs. 2014.01.05.

[14] 3GPP. Technical specification 25.305: Stage 2 functional specification of UE positioning in UTRAN. Available at http://www.3gpp.org/ftp/Specs/html-info/25305.htm. Accessed 01.05.2014.

[15] K. Sere, E. Troubitsyna, "Safety analysis in formal specification" In Formal Methods (FM'1999), Springer Sep. 1999, pp 1564-1583, ISBN:3-540-66588-9.

[16] N. Storey. Safety-critical computer systems. Addison-Wesley, 1996.

[17] E. Troubitsyna. "Elicitation and specification of safety requirements". In Third International Conference on Systems (ICONS 08), IEEE Apr. 2008, pp. 202-207, ISBN978-0-7695-3105-2.

 ISBN: 978-1-61208-361-2