Model-Driven Engineering of Fault Tolerant Microservices

Elena Troubitsyna
Åbo Akademi University
Turku, Finland
email: Elena.Troubitsyna@abo.fi

***Abstract*—The microservices architectural style has gained a significant popularity over the last few years. It promotes structuring applications as a composition of independent services of small granularity – microservices. Such an approach supports agile development and continuous integration and deployment. However, it also poses a significant challenge in ensuring the required quality of service and, in particular, fault tolerance. It requires a systematic analysis of possible failure scenarios and the use of structured techniques to implementing fault tolerance mechanisms capable of coping with the various types of failures. In this paper, we propose a structured approach to model-driven engineering of fault tolerant applications developed in the microservice architectural style. We define modelling patterns to facilitate the design of appropriate fault tolerance mechanisms. We also discuss how to integrate fault tolerance into the design of complex applications. We demonstrate how to graphically model the micorservice architectures and augment them with various fault tolerance mechanisms. The proposed approach facilitates a systematic analysis of possible failures, recovery actions and design alternatives. Our approach supports structured guided reasoning about fault tolerance at different levels of abstraction and enables efficient exploration of 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 applications.**

Keywords*— *microservices; fault tolerance; architecture; graphical modelling; fault tolerance pattern component.

I. INTRODUCTION

Microservices architectural style [11] has gained a significant attention over the last few years. The style builds on the service-oriented computing paradigm [10]. It supports continuous integration and deployment software engineering approach, which makes it a good fit for ubiquitous cloud-based environments. The microservices style was motivated by the need of small autonomous teams of developers, who do not own the full life-cycle of the application development, to continuously integrate and deliver, provide on-demand virtualisation and infrastructure automation.

Microservices aim at overcoming the drawbacks associated with developing, refactoring and maintaining monolithic applications. While supporting a quick and efficient development, integration and modification, the style introduces the additional complexity caused by the need to correctly orchestrate the distributed microservices as well as ensure the desired degree of Quality of Service (QoS).This is a challenging task because the microservices, in general, are developed using different languages and rely on lightweight communication to implement complex application-level scenarios. Therefore, to ensure the desired high degree of QoS and in particular, reliability, we should create an approach that enables a systematic analysis of different failures that might occur in the microservices architectures. Moreover, we should provide the developers with structured techniques to integrate different fault-tolerance mechanisms into the developed applications and analyse their impact.

In this paper, we propose a structured model-driven approach to modelling fault tolerance in the microservice architecture. We rely on Unified Modelling Language (UML) [9] – a popular graphical modelling language – to define patterns for representing different fault tolerance mechanisms and support their structured integration into the application architecture. We define the modelling patterns for representing fault tolerance mechanisms at different levels of abstraction.

We propose static and dynamic fault tolerance mechanisms. The static mechanisms are the structural solutions, which rely on availability of redundant service providers that can be requested to provide services in the case of failures of the main service providers. This mechanism allows the designers to mask failures of the individual service providers. The dynamic fault tolerance mechanisms rely on different monitoring solutions that enable more efficient handling of microservices and communication failures.

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 discuss the microservices architectural style. In Section III, we propose several fault tolerance mechanisms suitable for the microservice architectures. In Section IV, we introduce modelling of complex composite patterns. Finally, in Section V, we overview the related work and discuss the proposed approach.

II. MICROSERVICES ARCHITECTURAL STYLE

Microservice architectures [11] have emerged as a new architectural style, which aims at overcoming the problems associated with monolithic architecture. In monolithic applications, all functionality is put together to be distributed as a single file. Monolithic applications are simpler to deploy because they usually run on a single machine. Moreover, they are easier to develop because a programmer does not need to deal with abstractions associated with distributed architectures. However, large monolithic applications are hard to maintain, because even a simple refactoring requires rebuilding and redeploying the entire application. Moreover, since a monolith application is usually tightly coupled, failure of even a small part of it leads to the failure of the entire application. Handling runtime failures is especially cumbersome, because all components run in the same environment.

Another approach is taken from the service-oriented architectural style. In Service-Oriented Architecture (SOA) an application consists of independent, interoperable and reusable services, usually implemented as Web services. To facilitate achieving a loose coupling between the components, SOA aims at abstracting of the overall business logic [12].

Each service publishes its description, where it defines its capabilities. A service in SOA typically has one of two main roles – a service provider or service consumer. A service provider is invoked via an external source to provide some services according to its published capabilities. A service consumer (sometimes called service requestor) invokes service providers by sending them corresponding messages. A service can play just a single role in the composed application. It can also play both roles, e.g., if it functions as an intermediary that routes and processes messages or as a service director, which needs to invoke other services to provide a composite service, which is a part of application.

Typically, an application is composed of services that are hosted on different servers. SOA promotes an asynchronous communication to ensure stateless nature of services. Obviously, highly distributed nature of SOA makes development and deployment of services more challenging.

Microservice architecture builds on the concept of SOA. It promotes building an architecture consisting of autonomous and, hence, independently replaceable and upgradable services. Each microservice represents a small component specialising in implementing a certain functionality. Usually, microservices run in their own processes distributed across the network.

Microservice architectures have many benefits including availability, scalability as well as continuous integration and deployment [11]. Next, we discuss a few main characteristics of microservices, which we find particularly useful for achieving QoS of complex applications built in the microservice architectural style.

Single responsibility: the functionality of each microservice is narrowly focused. The main goal is to keep the code base as small as possible and ensure that each microservice can be redeveloped and redeployed in short time. Microservices emphasise the modularity principle, which, nevertheless, allows us to build large applications composed of numerous services.

Autonomy. As mentioned above, in a micorservice architecture, each microservice is typically run on its own process and the processes are distributed across the network. This introduces additional complexity but allows an application to cope with different performance demands and avoid tight coupling.

Heterogeneity. The microservice architecture supports technological independence in implementing each individual microservice, e.g., a programmer is free to choose any programming language to implement a microservice. The API of a microservice should be language-agnostic to ensure that the services can communicate with each other on different platforms.

Scaling. Microservice can be replicated if there is high performance demand or used differently in different parts of the application. Such an approach ensures good scalability of the microservices applications. Microservices have only run-time dependencies on each other and, hence, can be replaced or deployed independently, which further improves scalability and flexibility in developing complex applications.

The growing popularity of the microservices architectural style has led to creating several specialised platforms. Among the most popular platforms are Spring Boot and building on it Spring Cloud, WildFly Swarm, Payara Micro and SilverWare [11]. In addition, there are different libraries, frameworks and application servers, which allow the developers to create the applications in the microservice style.

Since microservices should run in highly distributed environments, their developers should deal with the complexity inherent to all distributed systems, in particular, complexity of achieving fault tolerance and reliable behaviour of the overall developed application. To achieve this goal, we should utilise the knowledge and best practices – design patterns – created in the area of fault tolerant computing and adapt them to the microservices style. In the next section, we focus on discussing various fault tolerance mechanisms and model-driven approach to designing them.

III. Fault-Tolerance In Microservice Architecture

The main goal of introducing fault tolerance in the microservice architecture (and SOA in general) is to prevent a propagation of faults to the application interface level, i.e., to avoid an application failure [7][8]. A fault manifests itself as *error* – an incorrect service state [7][8]. 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, certain actions are executed 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 [9] of fault tolerance mechanisms, which can be integrated into the microservice architecture [11]. We demonstrate how to explicitly introduce handling of faulty behaviour into the microservice architecture.

To model a microservice, we should analyse its interactions with the other microservices in the application under development. At the abstract modelling level, we treat a microservice as a black box with the defined logical interfaces. As we mentioned in Section II, in general, each microservice can play one of two roles – service consumer or service provider. Figure 1 and Figure 2 show the patterns for modelling service provider and service consumer correspondingly.

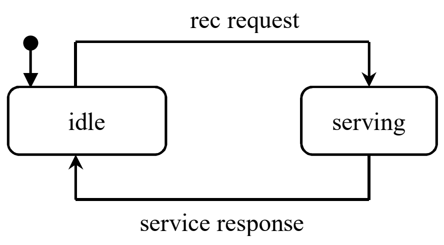

Figure 1. Behavior of service provider

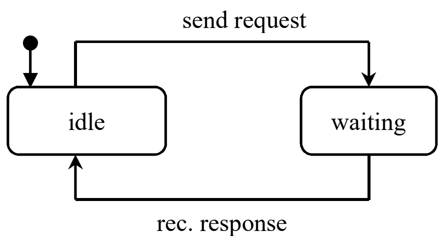

Figure 2. Behavior of service consumer

To model a microservice, we should analyse its interactions with the other microservices in the application under development. At the abstract modelling level, we treat a microservi

A high-level state diagram of the service provider is depicted in Figure 1. The microservice is idle and upon receiving a service request enters the state serving. When the requested computation is completed, the service provides the requested outcome and returns to the state idle.

A high-level state diagram of the service consumer is depicted in Figure 2. Similarly, the data consumer is activated after is issues the service request and upon receiving the requested results returns to the state *idle*.

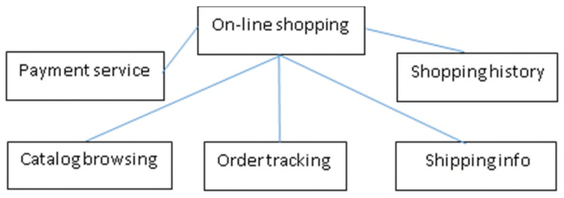

Figure 3. Example of microservice architecture

A generic microservice architecture can be represented by a diagram similar to the one shown in Figure 3. The scenarios to be supported can be modelled as a sequence of service requests and replies. Correspondingly, the microservices involved into an execution of the scenario play the roles of service providers and service consumers.

Let us analyse the possible failures that might occur while executing an application composed of microservices. We can identify three classes of failures:

1. Invalid service request
2. Invalid service response
3. Network failure

The first class of failures caused by an error in the request parameters. This failure can be easily detected during the scenario execution by the explicit response indicating the error. The second class of failures – the invalid service response – is caused by a logical error in implementing a certain microservice. This type of error can be detected by integrating the corresponding functionality into the service consumer that checks the validity of the obtained response.

Finally, the network failures are not caused by the microservices themselves. They can only be detected by integrating the corresponding monitoring mechanisms, for instance, timeouts, into the microservice architectures as well as the appropriate fault tolerance patterns, which we discuss next.

Timeout pattern. This pattern aim at coping with unreliable networks. As we discussed previously, a microservice architectural style promotes loose coupling of services, which results in a highly distributed execution of scenarios. In general, the network is unreliable and hence, connections might fail or become slow. Such network behaviour negatively effects the executions relying on the synchronous remote calls, i.e., can deadlock the scenario execution.

To prevent this, timeouts can be used to bound the waiting time. Despite the fact that timeout mechanisms are actively used at the operating systems level, their use at the application level is less common.

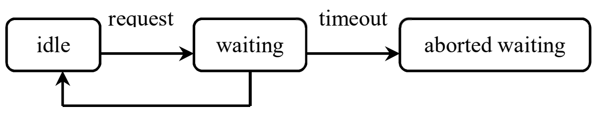

Figure 4. Timeout pattern for service consumer

Figure 4 graphically depicts the timeout pattern with respect to the service consumer. We have decided to single out the state of failed response, since it would allow us to explicitly collect data about failed responses when we study more complex fault tolerance patterns.

The interactions between the service consumer and service provider in the timeout pattern are shown in Figure 5.

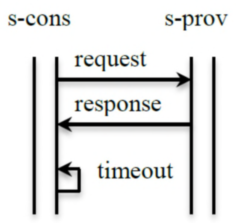

Figure 5. Interaction in timeout pattern

Circuit breaker. Circuit breakers detect excessive current in electric circuits and by failing open the circuit to prevent the connected appliances from damage. When the excessive current is removed, the circuit breaker can be reset and the circuit becomes closed and functioning again.

The idea of circuit breaker in the microservice architecture is similar to the electric one. It is a wrapper, which intercepts the erroneous (and potentially dangerous) calls to make sure that they do not harm the entire application. Hence, the main purpose of the circuit breaker is to monitor the behaviour of the network and services and if the failure rate exceeds certain threshold, make the calls to the remote destinations to fail immediately. After certain timeout, the circuit breaker sends some testing call to the quarantined server to check whether it has recovered. If the calls succeed then the circuit breaker stops failing the calls to this sever, i.e., it "closes" the circuit.

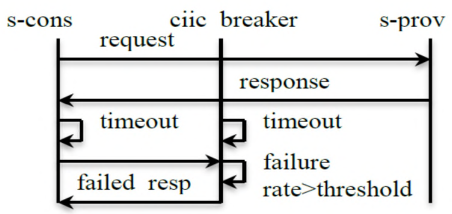

Figure 6. Interactions in the circuit breaker pattern

The graphical model for representing the interactions in the circuit breaker pattern is shown in Figure 6. The state diagram representing the dynamic behaviour is given in Figure 7. The circuit breaker is activated upon receiving a request from a service consumer. It monitors the request execution by the service provider and collects the corresponding data. If request fails, it checks whether the failure rate has exceeded the predefined threshold. If not then the circuit breaker continues to function in the monitoring mode, i.e., does not block the calls to the corresponding service provider.

However, if the failure rate threshold is exceeded, then the circuit breaker changes its mode to block, i.e., fail all the calls to the corresponding service provider. The service provider becomes quarantined. After the quarantine time expires, the circuit breaker sends a test request to the quarantined service provider. If the request is successfully returned the quarantine is removed and the circuit breaker returns to the monitoring mode.

Circuit breakers provide us with an efficient way to prevent cascading failures. They rely on timeout pattern to detect failures and correspondingly, recoveries of the service providers. The circuit breakers collect data about the behaviour of the microservices, which can be used to refactor them and continuously improve the reliability of the microservices architectures.

IV. Composite Fault Tolerance Patterns

In this section, we overview more complex fault tolerance patterns. They built on the patterns introduced in Section III as well as classic fault tolerance techniques.

Proactive fault tolerance. This pattern aims at preventing an execution of complex scenarios that cannot be executed to the completion. Proactive fault tolerance identifies potential failures before the scenario or part of it is executed, signals about the possible deadlock and either proposes an alternative way to execute a scenario or fails it.

The proactive fault tolerance pattern is a composite pattern that relies on timeout and circuit breaker patterns as well as other standard fault tolerance techniques.

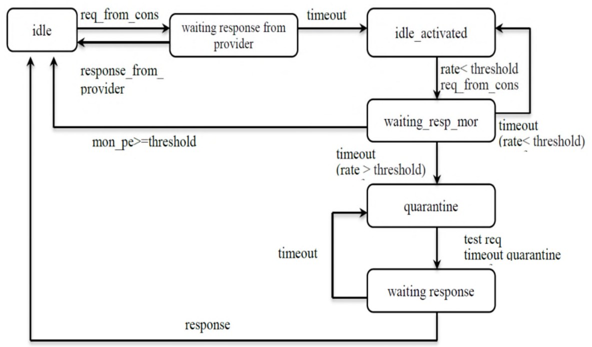

Figure 7. Dynamic behaviour of circuit breaker pattern

The graphical model of proactive fault tolerance pattern is shown in Figure 8.

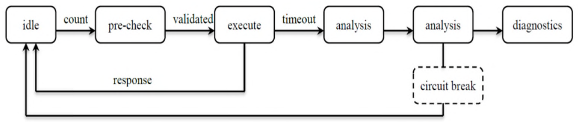

Figure 8. Dynamic behaviour of proactive fault tolerance pattern

Composite services. Some microservices are composite, i.e., to provide a requested service they need to request services from several other microservices, process the responses and finally provide the requested result. Let us note, that any other microservice might also be "composed" of several microservices, i.e., in its turn, the requested microservice execution might be orchestrated by its (sub)service director. Hence, in general, a composite microservice might have several layers of hierarchy.

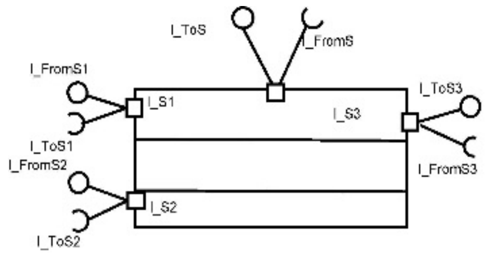

Figure 9. Service director: static view

To model a composite microservice, we introduce the providers of the microservices into the abstract architectural service model. The model includes the external service providers communicating with the microservice director via their service director, as shown in Figure 9.

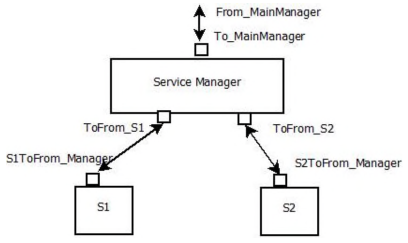

Figure 10. Duplication pattern: static view

Now, let us discuss the patterns that allow us to introduce structural means for fault tolerance using various forms of redundancy.

Duplication pattern. The duplication is a simplest arrangement for structural fault tolerance. It can be introduced if there are two microservice providers, which provide functionally identical microservices. In this case, the request from a service director of a service consumer can be duplicated and microservice providers activated in parallel. An execution is successful if any out of two microservice providers successfully completes the request.

An architectural diagram of the duplication arrangement is given in Figure 10 and the dynamic behavior shown in Figure 11.

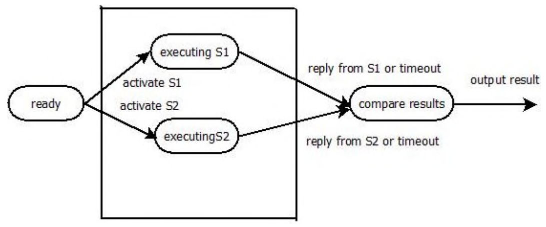

Figure 11. Dynamic behavior of duplication pattern.

Triple modular redundancy pattern. A more complicated scheme for structural redundancy – triple modular redundancy (TMR) is shown in Figure 12. The precondition for implementing it is that we have three microservice providers that provide functionally identical microservices. In this case, a service request from a service consumer or service director should be triplicated. All three microservice providers 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 microservice 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.

The voting microservice 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

The proposed patterns offer suitable solution for achieving fault tolerance in developing applications in the microservice architectural style.

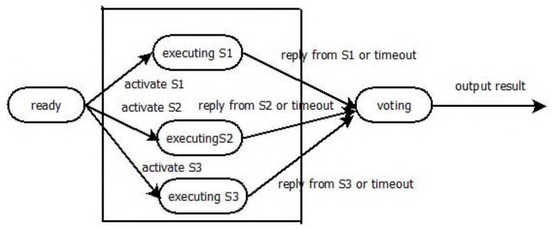

Figure 12. Dynamic behavior of TMR pattern.

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 [8] 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. 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 to introducing fault tolerance to the service architecture in the telecommunication domain has been proposed in [6][7][13][14]. 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.

The fault tolerance means are often assessed quantitatively. The techniques for probabilistic assessment of fault tolerance have been proposed in [15]-[18]. These techniques can be applied together with the proposed fault tolerance patterns.

In this paper, we have proposed a systematic model-driven approach to achieving fault tolerance in microservice architectures. We have defined generic modelling patterns, which can be utilised in model-driven engineering in the microservice architectures. Our patterns help to analyse possible failures and propose efficient solutions to cope with them. By integrating the proposed patterns into the architecture of a microservice, we can improve QoS and achieve higher reliability. Our patterns propose both dynamic and static means for achieving fault tolerance. The dynamic patterns rely on run-time monitoring behaviour and activating patterns if certain failure detection conditions occur. The static pattern help the designers to systematically utilise the redundancies present in the provisioning of microservices.

We believe that our approach supports structured guided reasoning about fault tolerance and enables efficient exploration of the design space while developing complex microservice architectures.

REFERENCES

[1] E. Alwagait, S. and Ghandeharizadeh, “A Dependable Web Services Framework” 14th International Workshop on Research Issues on Data Engineering 2004, http://fac.ksu.edu.sa/alwagait/publication/31143 retrieved January 2018.

[2] Artix Technical Brief. http://www.iona.com/artix, retrieved January 2018.

[3] V. Dialani, S. Miles, L.Moreau, D. Roure, and 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 and 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, 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.

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

[8] 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

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

[10] Web Services Architecture Requirements http://www.w3.org/TR/wsareqs, retrieved January.2018.

[11] M. Fowler and J. Lewis. Microservices: a definition of this new architectural term. In [Online]. Available https://martinfowler.com/articles/microservices.ml. Accessed: 01-April- 2019.

[12] T. Erl. Serivce-Oriented Architecture (SOA): Concepts, Technology, and Design. Prentice Hall, 2005. ISBN: 978-0131858589.

[13] A. Tarasyuk, E. Troubitsyna, L. Laibinis. Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In Proc. of *IFM 2012*, LNCS 7321, pp.237–252, Springer, 2012.

[14] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, L. Laibinis, Formal Development and Quantitative Assessment of a Resilient Multi-robotic System. In: Proc of *SERENE 2013*, LNCS 8166, pp. 109-124, Springer.

[15] E. Troubitsyna, “Reliability assessment through probabilistic refinement,” Nordic J. of Computing 6 (3), 320-342, 1999.

[16] L. Laibinis, B. Byholm, I. Pereverzeva, E. Troubitsyna, K.E. Tan and I. Porres, Integrating Event-B Modelling and Discrete Event Simulation to Analyse Resilience of Data Stores in the Cloud.The 11th International Conference on Integrated Formal Methods, iFM 2014, LNCS 8739, pp. 103-119, Springer 2014.

[17] I.Pereverzeva, L. Laibinis, E. Troubitsyna, M. Holmberg, M. Pöri, Formal Modelling of Resilient Data Storage in Cloud. In: Lindsay Groves, Jing Sun (Eds.), *Proceedings of 15th International Conference on Formal Engineering Methods*, LNCS 8144, 364–380, Springer-Verlag Berlin Heidelberg, 2013.

[18] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, L. Laibinis, Formal Development and Quantitative Assessment of a Resilient Multi-robotic System. In: A. Gorbenko, A. Romanovsky, V. Kharchenko (Eds.), Proceedings of the 4th International Workshop on Software Engineering for Resilient Systems (SERENE 2013), Lecture Notes in Computer Science 8166, 109–124, Springer-Verlag Berlin Heidelberg, 2013.