Towards Systematic Design of Adaptive Fault Tolerant Systems

Elena Troubitsyna, Kashif Javed
Åbo Akademi University, Finland
e-mails: {Elena.Troubitsyna, Kashif.Javed}@abo.fi

***Abstract*—The development of modern distributed software systems poses a significant engineering challenge. The system architecture should exhibit plasticity and high degree of reconfigurability to enable an automated adaptation to continuously changing operating conditions and component failures. Traditional engineering approaches are inefficient to cope with complexity of such systems to ensure their robustness and fault tolerance. Therefore, there is a clear need for the approaches explicitly addressing the problem of designing adaptive fault tolerance mechanisms. In this paper, we propose a systematic approach to the development of adaptive fault tolerant systems. We discuss the main principles of architecting such systems to enable plasticity and reconfigurability. We demonstrate how deployment of the predictive adaptation allows us to ensure that the system would be able to continuously deliver its services with the acceptable quality despite occurrence of component failures.**

Keywords-adaptable systems; fault tolerance, predictive adaptation; reconfiguration.

I. INTRODUCTION

The complexity of modern large-scale systems requires solutions that ensure that systems autonomously adapt to the operating environment and internal conditions. Often, such systems are put into a wide class of autonomic systems -- the software-intensive systems that, besides providing their intended functionality, are also capable to diagnose and recover from errors caused either by external faults or unforeseen state of environment in which the system is operating [3]. In this paper, we focus on the fault tolerance aspect of such systems.

Fault tolerance is an ability of a system to deliver its services in a predictable way despite faults [8]. The generic principle underlying design of fault tolerant systems is to detect a discrepancy between a model representing fault free system behaviour and the observed state, and implement error recovery [8] .

In this paper, we propose a general pattern for architecting and developing the adaptive fault tolerant systems. The proposed pattern supports a layered design approach [6] that enables separation of concerns and facilitates structured design of fault tolerance mechanisms. In our representation of the architectural pattern, we define the interfaces between the components at different levels of abstraction to ensure correct propagation of fault tolerance related data. The high-level coordination of the fault tolerance mechanisms is implemented by an adaptation manager – a component that is responsible for implementing predictive fault tolerance. To specify the adaptation manager, we propose an algorithm that allows the adaptation manager to monitor state of the system at the run time and implement proactive adaptation. Such an approach ensures that the overall system would continuously deliver the services with the acceptable quality. We believe that the proposed approach ensures a systematic development of adaptive fault tolerant systems.

The paper is structured as follows: in Section II, we overview the state-of-the-art in designing adaptive fault tolerant systems. In Section III, we describe general principles of achieving fault tolerance, and, in particular, proactive fault tolerance. In Section IV, we present our proposal for structuring adaptive fault tolerant system. In Section V, we present our proposal for algorithms that implement proactive fault tolerance. Finally, in Section VI, we discuss the proposed approach and future work.

II. RELATED WORK

The need for high performance and continuous service provisioning demands novel solutions for achieving system fault tolerance. We are increasingly observing deployment of proactive fault tolerance techniques that replace traditional reactive approaches [10]. In modern large-scale systems, error rate is increasing and reliance on traditional "error-detection – error-recovery" pattern leads to poor performance and prolonged system downtime, which is often unacceptable. The approaches for proactive fault tolerance are based on preventive treatment of faults aiming at precluding failures and minimising recovery time [10]. The main mechanism of achieving proactive fault tolerance is adaptation.

The problem of software adaptation has been extensively studied at the implementation level, (see e.g., [2] for an overview). However, there is a lack of approaches that attempt to derive appropriate adaptation mechanisms from system-level goals as well as support layered reasoning needed to efficiently cope with system complexity. A prominent work on formal modelling of adaptive systems has been done within the HATS project [2]. In [13][14], an approach to quantitative assessment of reconfiguration strategy has been proposed. In our previous work, we also investigated the impact of faults on dependability, as well as structured approach to designing fault tolerant distributed systems [7][11].

Current engineering practice takes an architecture-centric perspective on adaptive systems. Among the most prominent examples are the Rainbow framework proposed at Carnegie Mellon University [12] and the autonomic computing initiative by IBM [3]. These frameworks outline the main abstractions for describing and managing dynamic system changes. However, currently, the approaches to proactive fault tolerance are not well-integrated into the system development process [10]. In this paper, we will address this problem by proposing a structured approach to architecting adaptive fault tolerant systems. Our approach aims at facilitating design space exploration at the early development stages and enabling explicit representation of the mechanisms for proactive fault tolerance.

III. FAULT TOLERANCE

The main goal of introducing fault tolerance is to design a system in such a way that faults of components do not result in a system failure. A fault cannot be detected by a system until the manifestation of the fault generates *errors* in the component function. The first step in implementing fault tolerance is error processing [10]. *Error processing* aims at removing errors from the computational state.

The first step in error processing is *error detection*. An error is a manifestation of a fault. The general mechanism of error detection is to intercept outputs produced by a system (or a component) and to check whether those outputs conform to the specification of fault free behaviour. Discrepancy between produced outputs and the specification indicates an occurrence of an error. The next step in error processing – *damage confinement* – is concerned with structuring the system to minimise the spread of errors. Once the damage is assessed and confined the error recovery can be performed. *Error recovery* has two main forms – forward and backward error recovery. The forward error recovery mechanisms manipulate the current system state to produce a new system state, which is presumably error free. The success of error recovery strongly depends on how precisely the error is located and how well it is confined. A typical example of forward recovery is *failsafe* [1]. If a system has a safe though non-operational state then it may be possible to recover from an error by forcing the system permanently to that safe state (obviously, this strategy is only appropriate where shut down of the system operation is possible).

By analyzing actions to be undertaken for error processing, we observe that error processing imposes additional requirements on the system design. Namely:

- The system should be specified in such a way that error occurrence conditions are easily deduced and then explicitly checked;
- The system architecture should enable error confinement;
- Error recovery procedures should be identified for every output, which differs from the specified one.

Obviously, an incorporation of error processing in the system design has a strong impact on all levels of the system structure. Hence, fault tolerance should be an intrinsic part of system development and should start from the early stages of the system design.

To embrace complexity challenge, fault tolerance community has been proposing new concepts that can be seen from initiatives and research efforts on autonomic computing [3] and various forums on self-healing [9] or self-protection (see, e.g., [1]). These terms span a wide range of research fields ranging from adaptive memory management to advanced security mechanisms.

A promising direction among them focuses on determining how computer systems can proactively handle failures: if the system knows about a critical situation in advance, it can try to apply countermeasures in order to prevent the occurrence of a failure, or it can prepare repair mechanisms for the upcoming failure, in order to reduce the time-to-repair.

Such an approach can be called proactive fault tolerance. It encompasses three main steps:

1. Failure prediction: it aims at identifying failure-prone situations, i.e., the situations that will probably evolve into a failure. The result of failure prediction is an evaluation of whether the current situation is failure-prone.

2. Proactive reconfiguration: based on the outcome of failure prediction, a system should make a decision and implement the countermeasures to be executed in order to remedy the problem. These decisions are based on an objective function taking into account the cost of the actions, the confidence in the prediction, and the effectiveness and complexity of the actions to determine the optimal tradeoff. Challenges for action execution include online reconfiguration of globally distributed systems, data synchronization of distributed data centers, and many more.

3. Recovery: this stage enables graceful degradation of services while the resources are insufficient for mitigating the failures. For instance, the predictive reconfiguration might not be completed as promptly as expected and the system should compensate for insufficient resources. Another example would be a sudden simultaneous failure of several components due to unexpectedly adverse situations in the environment.

Each one of these stages is important for an efficient implementation of the proactive fault tolerance. Hence, novel architectural solutions, algorithms and development approaches are needed to attain the goal of building adaptive fault tolerant systems.

To build a proactive fault tolerance solution that is able to boost system dependability, the best techniques from all fields for the given surrounding conditions have to be combined.

In this paper, we consider the proactive fault tolerance to be the main adaptation mechanism to achieve system dependability. In the next section, we present our approach to structuring an adaptive fault tolerant system. Then, we focus on designing the proactive adaptation mechanisms. Our proposal aims at enhancing self-adaptation system capabilities. Our goal is to design the mechanisms that allow a system to autonomously adapt to changing operating conditions without human intervention. Essentially, our proposal follows a spirit of the autonomic computing paradigm.

IV. ARCHITECTURE OF ADAPTIVE FAULT TOLERANT SYSTEMS

In this paper, we propose to structure an adaptive fault tolerant system in a layered manner [6]. The layered architecture significantly simplifies the development of complex software-intensive systems. Each layer becomes responsible for a certain aspect of the system behaviour. It facilitates a clear separation of concerns and simplifies the interfaces between the layers. The main issue is to device a well-structured clean architecture that does not introduce tangled interdependencies between layers. In this paper, we propose to structure the architecture of a fault tolerant adaptive system in four layers:

- Application layer
- Adaptation layer
- Fault tolerance layer
- Physical layer

The physical layer represents the environment whose state should be monitored. It might be a complex control system that uses sensors to monitor the health of its components. Another example might be an indoor sensor network that monitors such conditions as temperature, humidity, the level of CO, etc. Finally, it might also be a sensor network for monitoring the outdoor environment, e.g., such as used for forest fire detection, air pollution etc.

The fault tolerance layer performs the data aggregation and evaluation of the quality of monitoring. This information is supplied to the adaptation layer that is responsible for defining the proactive adaptation policy. The aim of the application and fault tolerance layer is to continuously supply the application with the monitoring data of an acceptable quality. The design of the application is defined by its purpose – it varies from the complex control functions to collecting data intelligence. The graphical representation of the system architecture is given in Fig.1.

The physical layer consists of the component to be controlled by the application software. In order to implement proactive fault tolerance, the software should continuously monitor the state of the controlled components.

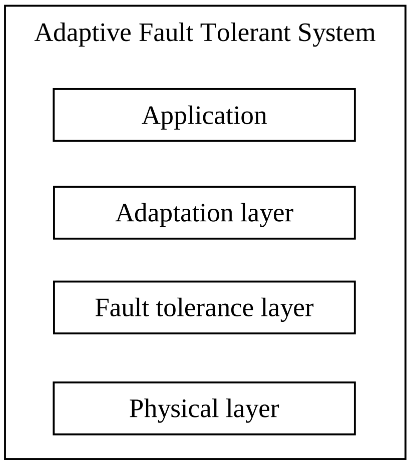

Figure 1. Structure of an adaptive fault tolerant system.

The monitoring capabilities are achieved by integrating sensors that measure the parameters required to observe the behaviour of the system in real-time. Usually, complex systems contain a large number of sensors. Hence, from the fault tolerance perspective, the physical layer can be considered as a sensor network.

It generates raw data. Each sensor produces the data in the following format

<value, timestamp>

We consider two most typical failure modes of the sensors: *stuck at previous value* and producing a (detectably) *incorrect value*. In the former case, the sensor fails silently by failing to update its reading, i.e., the timestamp indicates that the produced data is old. In the latter case, the sensor produces the value that is outside of the feasible range.

At the fault tolerance layer resides fault tolerance manager. The goal of the fault tolerance manager is

- To periodically read the sensor data,
- To filter out faulty data,
- To compute the average value of valid data together with defining the quality level.

The fault tolerance manager produces the input for the adaptation manager as a tuple

<value, level>

To compute the quality level, the fault tolerance manager keeps track of the number of sensors that have produced valid data. There are two thresholds: *lim1* and *lim2* such that *lim2 > lim1*. They determine the quality level. If the number of the sensors that produced the valid data is greater than *lim2* then the quality level is set to *Level 3*. If the number of sensors produced valid data is between *lim1* and *lim2* then the quality level is set to *Level 2*. If the number of valid readings is between *1* and *lim1* then the quality level is set to *Level 1*. Finally, if none of the sensors have produced valid results then the quality level is assigned value *Level 0*.

The adaptation manager and deployment manager constitute the adaptation layer. The adaptation manager receives the data from the fault tolerance manager in the format

<value, level>

where *level* is an integer between *0* and *3*. If the level has value *3*, then, the value has a good quality and the adaptation manager simply forwards the received value to the applications. However, if the quality level is below *3* but greater that *0* then the adaptation manager still forwards the received data to the application but starts an observation period.

The aim of the observation period is to establish whether the decline in the quality of data is temporal or permanent. Assume that, after receiving a value with the levels *1* or *2*, the adaptation manager observes a continuous period of receiving data with quality level *3*. Then, the observation period terminates and no reconfiguration is initiated, i.e., the adaptation manager treats the decline in the quality of data as a temporal one and considers the system to be healthy.

If, during the observation period the adaptation manager continuously receives data with quality level *1* or *2* then after the observation period expires, it initiates reconfiguration, i.e., considers the quality deterioration to be the permanent one.

The reconfiguration is triggered by sending a request to the *deployment manager* to deploy a new set of sensors. The deployment can be achieved in several different ways. For instance, if we consider a wireless sensor network that is used to monitor the state of the environment then the deployment is performed via a distribution of a set of fresh sensors (e.g., from an airplane). If the sensors are used to monitor an indoor environment then the deployment triggers a request to the maintenance company. The same principle applies if the sensor network is used to monitor the behaviour of a complex control system. In any case, the main advantage of the proposed approach is a possibility to preventively react on the deterioration of the quality of monitoring and avoid the loss of the observability of the physical layer.

The requested number of new sensors to be deployed depends on how deeply the level of data quality has deteriorated. If the quality level has value *1* then the deployment manager requests *n* new sensors to be deployed. If the quality level has the value *2* then *m* new sensors are to be deployed, where *m<n*.

In general, we could design a more sophisticated deployment mechanism. For instance, if each sensor or a group of sensors is assigned an id then the failures can be diagnosed precisely. This would allow the adaptation manager to communicate the exact requirements for the deployment of new sensors.

When the new sensors are deployed, the deployment manager acknowledges the completion of the reconfiguration and the adaptation manager notifies the fault tolerance manager about availability of the new sensors. The fault tolerance manager closes the connection with the failed sensors and establishes connection with the newly deployed ones.

An important aspect to be considered is how to define the behaviour of the adaptation manager when the quality level keeps fluctuating between the values 2 and 3. On the one hand, the adaptation manager should not trigger the reconfiguration prematurely. On the other hand, delaying a reaction on such an unstable situation might result in an abrupt deterioration of the quality of data that should be prevented.

To resolve this issue, we let the adaptation manager to maintain the observation period as long as no continuous improvement in quality has been observed. Every time when the data are received with the quality threshold lower than *3*, the adaptation manager increments the counter of the observation period. When this counter exceeds the predefined threshold, the adaptation manager triggers the reconfiguration. This approach is taken to ensure that the preventive reconfiguration will be initiated even if the system keeps fluctuating between quality levels.

Finally, if the adaptation manager receives data with the quality level equal to *0*, then it immediately initiates reconfiguration of the data flow. In this case, it starts to send to the application data received at the previous cycle. It continues to send the last data with an acceptable quality value until the reconfiguration is completed and the fault tolerance manager starts to send the data with an acceptable quality level.

In the next section, we define the main behavioural patterns of adaptation manager and fault tolerance manager.

V. ALGORITMS FOR PROACTIVE FAULT TOLERANCE

Let us focus first on defining the module specifying the fault tolerance manager.

The module should implement the procedures of

- Reading the sensor data,
- Checking validity of sensor data with respect to time and feasibility
- Calculating the average of the received valid data and the quality level.

In our definition of the fault tolerance manager, we used two abstract functions *fresh* and *valid*. The function *fresh* relies on the specific parameters to determine whether the produced data is fresh. Since the clocks of the sensors might fluctuate, the function checks whether the timestamp is within certain boundaries.

The function *valid* checks feasibility of the data produced by a sensor. It returns the Boolean value *True* if the data is valid and *False* otherwise.

```
Module Fault Tolerance Manager
Global Variables
   in_buffers: array of  <float, INT>
   out_buffer: seq of <float, INT>

Local Variables
   count: INT /*counter of healthy sensors
   sum : float /*sum of readings
   avg: float /*average value
   level:  [0..3]

Initialisation:
   count:= 0;
   sum:= 0;
   avg:= 0;
   level:= 0

Begin

 for i = 1 to k do
    read (data, time_stamp, in_buffer[i]);
    if
         fresh (time_stamp) = True & valid(data)= True
    then  count:= count +1; sum := sum +data
   end;

 if counter > 0 then avg:= sum/count;

 case count = 0 then level:= 0
   elseif count>0 & count<lim1 then level:=1
   elseif count>lim1 & count<lim2 then level:=2
   else level:=3;

 out_buf:= out_buf^<avg,level>;
 count:= 0;
 sum:= 0;
 avg:= 0

End
```

Figure 2. Fault Tolerance Manager.

Reliance of the abstract functions allows us to parameterise the definition of the module and reuse the proposed definition in different contexts.

In our definition of the module, we have abstracted away from the implementation details of the communication between the fault tolerance manager and the sensors. We assume that they communicate by shared variables -- data and time stamps that are stored in the *in_buf* array of pairs.

The proposed algorithm implements the procedure of reading the sensor data, checking their validity with respect to time and feasibility and calculates the average of the received valid data.

By keeping track of the number of valid readings, the fault tolerance manager calculates the quality level. It compares this number with two constants – *lim1* and *lim2*. The pair of calculated data and the quality level is appended to the output buffer that is read by the Adaptation Manager. The specification of the Fault Tolerance Manager module is given in Fig. 2 and the Adaptation Manager in Fig. 3.

```
Module Adaptation Manager

Global Variables
     a_out_buf: float

Local variables:
     observ : Bool
     cur_level : INT
     cur_data:float
     fault_count : INT
     suc_count : INT
     mode: {Normal, Adapt, Adapt_Compl, Adapt_activ}

Initialisation:
     observ :=0;
     cur_level :=0;
     fault_count :=0;
     suc_count :=0;

Begin

 cur_level, cur_data := head(out_buf);

if observ= False & cur_level= 3 then out_buf:= cur_data

if observ= False & cur_level= 2 & fault_count<thr
 then fault_count:= fault_count+1; out_buf:= cur_data;

if observ= False & cur_level<3 & cur_level>0 &
   fault_count>thr-1
 then mode := adapt_active, adapt_req:= True;

if observ= False & cur_level=3 & fault_count>0 &
    fault_count<thr-1
then observ:= True; suc_count := suc_count +1;
     observ:= 0;

if observ= True & cur_level=3 & fault_count>0 &
   fault_count<thr-1 & suc_count<thr_s
then suc_count:= suc_count+1;
     observ_s_iter:= observ_s_iter:=+1;

if observ= True & cur_level=3 & fault_count>0 &
   fault_count<thr-1 & suc_count>thr_s-1 &
   suc_count =observ_s_iter
then observ:= False ; suc_count:= 0; fault_count:= 0;
      observ_s_iter:= 0;

if observ= True & cur_level<3 & fault_count>0 &
   fault_count<thr-1 & suc_count<thr_s
then suc_count:= suc_count+1;
    observ_s_iter:= observ_s_iter+1;

if mode= adapt_activ then adapt_req ;

if adapt_conf  then mode:= normal
End
```

Figure 3. Adaptation Manager.

In the specification of the Adaptation manager, the variable *observ* indicates whether the observation period has started. The variable obtains the value *True* when the first data with the quality level below *3* is received. The variable is reset to *True* if the quality has recovered or a new period of observation is initiated.

The variables *cur_level* and *cur_data* designate the data and the quality level received from the fault tolerance manager. The variable *fault_count* is used to keep track of the number of iterations, in which the data with the quality level lower than 3 have been received. When the value of *fault_count* exceeds the predefined threshold *thr*, the reconfiguration is triggered.

The variable *suc_count* is used to keep track of the iterations that produced data with the quality level 3 after the observation period has been initiated. When the value of *suc_count* exceeds the predefined threshold *thr_s* the adaptation manager has continuously received the data with the quality level 3 for sufficiently long period of time. Therefore, the quality level has recovered and the observation period can be deactivated.

The adaptation manager provides the application with the latest data by updating the global variable *a_out_buf*. It forwards the data received from the fault tolerance manager if the quality level is higher than zero. Otherwise, it simply does not update the variable.

The adaptation manager triggers the reconfiguration by issuing the adaptation request *adapt_req* that is received by the deployment manager. When the new sensors are deployed the deployment manager confirms the reconfiguration by issuing the signal *adapt_conf.*

After triggering the reconfiguration, the adaptation manager enters the mode *Adapt*. After the reconfiguration is completed, the adaptation manager enters the mode *Adapt_Compl*. In this mode [4] [5], it notifies the fault tolerance manager about availability of new healthy sensors. As a response to this, the fault tolerance manager shuts down the connection with the failed sensors and establishes a new connection with the newly deployed sensors. After this procedure is completed, the fault tolerance manager notifies the adaptation manager. It enables transition to the mode *Normal*.

The general scheme of an implementation of the mode transition is given in Fig. 4. The main principle that underlies the mode transition is as follows: the mode is stable and unchanged until a fluctuation in the quality level is registered. We show the snippet implementing this principle as a generic mode changing procedure.

The proposed architecture ensures a separation of concerns and clear allocation of responsibilities between the components. Indeed, the fault tolerance manager is responsible for collecting data and validating them. It encapsulates the failures of sensors and gives only the high-level indication of the current health of the system by annotating the data with the quality level. The adaptation manager is responsible for diagnosing the situation and executing the preventive reconfiguration – requesting the new sensors to be deployed before the quality of data deteriorates below the acceptable level. At the same time, it also ensures remedial actions when no data is produced – it outputs to the application the last healthy value. Such behaviour ensures graceful degradation of quality of service.

Procedure *ModeTransition*

```
Variables
  last_mode: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  next_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  prev_target: {Normal, Adapt, Adapt_Compl, Adapt_activ}
  level: int

Begin

if adaptation completed
then initiate a forward transition
       to next_target according to
       the predefined scenario;

if level dropped
then initiate a backward transition to next_target
       adaptation mode
      The choice of target mode depends on severity
      of level decrease;

if the conditions for entering the target
  mode are satisfied
then complete a transition to next_target mode
       and become stable ;

if neither the conditions for entering
  the next global mode are satisfied nor the level dropped
then maintain the current mode

End
```

Figure 4. Mode transition procedure.

VI. Conclusion and future work

In this paper, we have proposed a systematic approach to architecting adaptive fault tolerant systems. We have demonstrated how to structure the system to facilitate layered design of proactive fault tolerant mechanisms. We defined the information flow between the layers of the system architecture that enables adaptation and guarantees a continuous delivery of services with an acceptable quality level.

Proactive fault tolerance is a promising research direction that aims at providing systems with capabilities of executing preventive reconfiguration to preclude occurrence of failure and disruption in service provision. In our paper, the main mechanism of achieving proactive fault tolerance relies on several levels of error detection and monitoring of system health.

As a future work, we are planning to investigate alternative approaches to preventive reconfiguration as well as conduct quantitative assessment of various system characteristics, e.g., correlation between frequency of the network rejuvenation with new sensors and quality of data, proportion between periods of low quality data and different thresholds etc. Such a work, would allow us to define heuristics for designing proactive fault tolerance.

REFERENCES

[1] O. Babaouglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen (Eds.) *Self-Star Properties in Complex Information Systems*. LNCS 3460. Springer-Verlag, 2005.

[2] *HATS Project*: Highly Adaptable and Trustworthy Software using formal models. *www.hats-project.eu/*.Accessed 20.03.2014

[3] P. Horn, Autonomic Computing*:* IBM's perspective on the State of Information Technology. http://researchweb.watson.ibm.com /autonomic/. Accessed 20.03.2014

[4] A .Iliasov, E. Troubitsyna, L. Laibinis, A.Romanovsky, and K.Varpaaniemi. Verfifying Mode Consistency for On-Board Satellite Softyware. In.Proc. SAFECOMP 2010, LNCS 6351, pp. 126-141, Springer, 2004.

[5] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K.Varpaaniemi, D. Ilic, T. Latvala, Developing Mode-Rich Satellite Software by Refinement in Event B . In: *Proc. of FMICS 2010*, LNCS 6371, pp. 50-66, Springer, 2010.

[6] L. Laibinis and E.Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In Proc. of SEFM 2004. pp. 346-355, IEEE Computer Press, 2004.

[7] L. Laibinis, E. Troubitsyna, A. Iliasov and A. Romanovsky. Rigorous Development of Fault-Tolerant Agent Systems. *Rigorous Development of Complex Fault-Tolerant Systems*. LNCS 4157, pp. 241-260, Springer, 2006.

[8] J. C. Laprie, *Dependability: Basic Concepts and Terminology*. New York, Springer-Verlag, 1991.

[9] M. Salehie, L. Tahvildari: Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4(2). ACM, 2009.

[10] F. Salfner, M. Lenk, and M. Malek: A survey of online failure prediction methods. *ACM Comput. Surv*. 42(3), 2010.

[11] K. Sere and E. Troubitsyna. Safety Analysis in Formal Specification. In *Proc. of FM'99,* LNCS 1709, pp. 1564 – 1583, Springer, 1999.

[12] B. Schmerl, J. Aldrich, D. Garlan, R. Kazman, and H. Yan. Discovering Architectures from Running Systems. In *IEEE Transactions on Software Engineering*, Vol. 32(7), July 2006.

[13] A. Tarasyuk, I. Pereverzeva, E. Troubitsyna, T. Latvala, and L. Nummila, Formal Development and Assessment of a Reconfigurable On-Board Satellite System. In Proc. of *SAFECOMP 2012*, LNCS 7612, pp. 210–222, Springer-Verlag, 2012.

[14] E. Troubitsyna. Reliability assessment through probabilistic refinement. Nordic Journal of Computing 6(3), 320-342, 1999.