Towards Formal Specification of Autonomic Control Systems

Elena Troubitsyna
Åbo Akademi University, Dept. of IT
Joukhaisenkatu 3-5A, 20520, Turku, Finland
Turku, Finland
e-mail: Elena.Troubitsyna@abo.fi

***Abstract*— Autonomic systems represent the next generation of software-intensive systems that merge software, computing, communication, sensing and actuating to create intelligent self-aware computing environment. Autonomic systems penetrate majority of critical infrastructures be it air-, rail and road traffic or power supply management. So far, little attention has been paid to theory and techniques for ensuring safety and resilience of such systems. Are autonomic systems to bring benefits or devastating hazards? To ensure harmless deployment of autonomic systems in critical infrastructures we should significantly advance our understanding of principles governing adaptive behaviour of such systems. Therefore, it is important to create formal techniques for modelling adaptive behaviour of autonomic control systems. In this paper, we discuss issues in modelling autonomic control systems that achieve self-adaptation through feedback loops and derive general guidelines for their formal specification.**

Keywords-*autonomic computing; control systems; action systems; formal specification*

I. INTRODUCTION

Autonomic systems are software-intensive systems that besides providing its 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. Autonomic systems are typical examples of self-adaptive systems. The concept of autonomic systems has been introduced in the recognition of complexity crisis. Currently the level of complexity of software has reached unprecedented level and we are no longer can reliably guarantee correct function of the system. Even though complexity is perceived as a major threat to dependability, self-adaptive systems are becoming more and more widely used in critical infrastructures. It is threatening situation that might cause catastrophic consequences.

Originally, autonomic computing paradigm was proposed in a very radical way: autonomic systems were supposed to mimic self-adaptive living organisms that can autonomously take care of themselves. In this paper, we are taking a stand that in the domain of critical systems we should take more moderate view and consider autonomic behavior that converges to a formally verified model that guarantees that the essential properties of the system are preserved despite self-adaptation.

In this paper, we discuss the principles of structuring formal models of autonomic control systems. We demonstrate how to formally specify behaviour of autonomic control system in the action systems formalism [2,3]. The formalism provides us with a unifying framework for developing terminating as well as reactive distributed systems. Our main development technique is stepwise refinement [4]. While developing a system by refinement, we start from an abstract specification and refine it into an executable program in a number of correctness preserving steps – refinements. Stepwise refinement allows us to incorporate system requirements into the specification gradually and eventually arrive at system implementation, which is correct by construction.

In this paper, we propose a general pattern for abstract specification and refinement of autonomic control systems. We present a novel pattern for an abstract specification of autonomic manager -- a components that is responsible for monitoring and adaptation of the control system. Our refinement steps gradually introduce detailed representation of data structures required to model autonomic system with a feedback control loop.

The proposed approach provides the developers with a rigorous framework for systematic development of fault tolerant distributed systems.

The paper is structured as follows: in Section II we describe a general architecture of the autonomic control systems with feedback loop. In Section III we present our formal modelling framework – the Action Systems formalism. In Section IV we demonstrate how to specify the autonomic manager and components of the autonomic control systems. Finally, in Section V we discuss the proposed approach and future work as well as overview the related work.

II. AUTONOMIC CONTROL SYSTEMS

The complexity of modern software systems and volatile environment in which they operate require novel computing paradigms to ensure that the system delivers the desired behaviour, i.e., is capable to adapt to the changing operating conditions. The autonomic computing paradigm is a promising research direction that puts the main emphasis on system self adaptation capabilities. Essentially, self-

 ISBN: 978-1-61208-219-6

adaptation is a capability of the system to adjust its behaviour without any human intervention.

In this paper, we consider issues in modelling systems that achieve self-adaption through feedback loops. In particular we focus on studying autonomic control systems. In general, a control system is a reactive system with two main entities: a plant and a controller. The plant behaviour evolves according to the involved physical processes and the control signals provided by the controller. The controller monitors the behaviour of the plant and adjusts it to provide intended functionality and maintain safety. The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. The general structure of a control system is shown in Fig. 1.

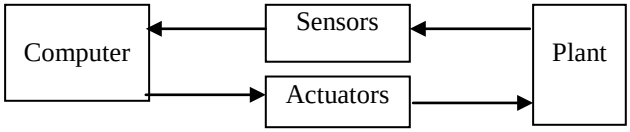

Figure 1. A general structure of a control system

A general structure of an autonomous control system is shown in Fig. 2.

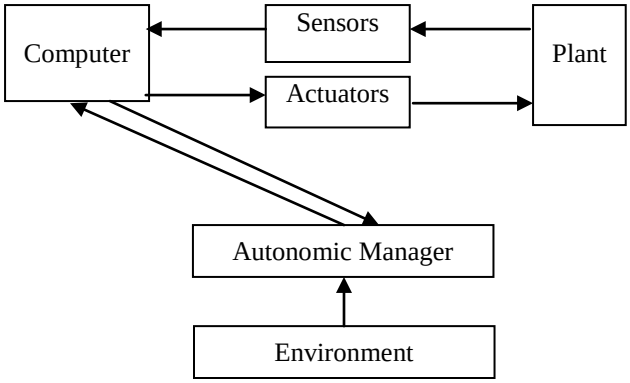

Figure 2. A general structure of an autonomic control system

A self-adaptive control system has an additional feedback control loop – we call it autonomic control loop. The loop has four main functions: monitor, analyse, decide and act. The monitoring activities are implemented via external sensors or monitors that collect data from the system and its environment. Usually the data acquired in the process of monitoring are filtered and stored in a log. The aim of collecting the data is to obtain an accurate model of the system dynamics and its current state. The collected data form the basis for diagnostics of failures, trends in operating environment, etc. There is a large variety of methods used for the analysis of the collected data. There are two general approaches: the first group relies on a reference model – a model of the expected system that is encoded into the analysis procedure at the design phase. Another type of analysis relies on inferring the model of the behaviour at the run time, i.e, no predefined model is given and the system is gradually building the model. In our paper we focus on the modelling the former class of systems.

Once the analysis of the collected data completes the planning phase takes place. The system decides on the strategy along which to continue its function. This strategy is then transformed into the control signals that are communicated to the actuators to implement the chosen strategy. This completed the cycle of the autonomic control loop.

In this paper, we focus on the analysis of autonomic control systems with a centralized autonomic manager. The autonomic manager is responsible for executing autonomic control loop. By communicating with the components of the system it collects the data required for diagnostics of the internal system state. The queues of the internal service requests as well as environmental conditions are monitored to define the usage profile and plan how the system functioning should proceed. The autonomic manager periodically sends diagnostics requests to the system components as well as requests reading from the external monitoring sensors and monitors the external service requests. This information is input to the analyzing component of the autonomic manager. Essentially the analyzing component compares the obtained data with the reference model and passes the control to the planning component. The planning component decides on the further strategy. The developed strategy is passed to the actuating component that sends the required control signals. The static view on the architecture of an autonomic manager is given in Fig. 3.

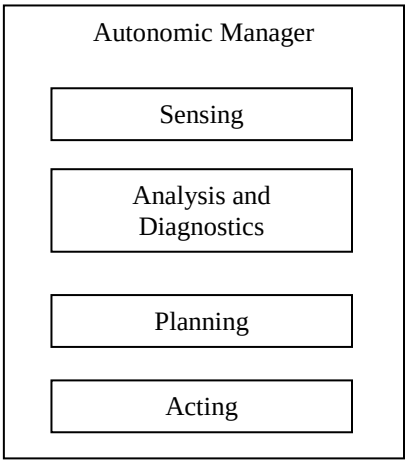

Figure 3.Structure of autonomic manager

As an example of an autonomic control system, let us consider an autonomic robot. Service robots form a quickly growing commercial area as well as research field. Service robots are designed to assist humans in performing services semi or completely automatically. There is a large variety of robots that are used for inspection, housekeeping, office automation and assisting elderly people or people with disabilities. The example that we present in this paper is

inspired by the intelligent service robot developed to assist elderly people. The robot should be able recognize a voice command and bring a desired object (e.g., medicine) from a certain position. We focus on the function of autonomous navigation. A user can command the robot to move to a specific position in the map to perform some task. For instance, the robot can navigate to its destination in the home environment via its sensors, which include laser scanners and ultrasonic sensors. The robot plans a path to the specified position, executes this path, and modifies it as necessary for avoiding obstacles. While the robot is moving, its constantly checks the data from its sensors.

Obviously, despite the complexity the robot should guarantee a high degree of dependability. For instance, we should ensure that the robot does not collide to the obstacles (and gets broken as a consequence leaving the person without the assistance). To facilitate design of dependable autonomous systems we propose to rely on formal modelling that provides us with a rigorous basis for reasoning about system behavior.

III. ACTION SYSTEMS

The action systems formalism [1] is a state-based approach to formal specification and development of parallel and distributed systems. The formalism has proven its worth in the design of complex parallel, distributed and reactive systems [5,13,14]. Below, we briefly describe the action systems.

A. Action Systems

The action system **A** is a set of actions operating on local and global variables:

$$\begin{aligned}\mathbf{A} :: |[\ &\mathbf{proc}\ p_1{*}=P_1;\ \ldots;\ p_N{*}=P_N;\ \ q_1=Q_1;\ \ldots;\ q_M=Q_M;\\ &\mathbf{var}\ v{*},u \bullet Init;\\ &\mathbf{do}\ A_1\ []\ldots[]\ A_K\ \mathbf{od}\ ]| : z\end{aligned}$$

The system **A** describes a computation, in which local variables u and exported global variables v^* are first created and initialised in *Init*. Then, repeatedly, any of the enabled actions *A1*, …, *An* is non-deterministically selected for execution. The computation terminates if no action is enabled, otherwise it continues infinitely. The actions operating on disjoint sets of variables can be executed in any order or in parallel.

The local variables u are only referenced locally in **A**, while the exported global variables v^* also can be referenced by other action systems. The imported global variables z are mentioned in the actions $A_1,\ldots, A_K$ but not declared locally. The identifiers of local, global imported and global exported variables are assumed to be distinct.

A procedure declaration $p=P$ consists of the *procedure header p* and the *procedure body P*. The procedures marked with * are declared as the exported procedures. They can be called from **A** and other action systems. The procedures $q_1,\ldots, q_M$ are the local procedures. They can be called only by **A**. The local and exported procedures are all assumed to be distinct.

The action A is a statement of the form $g(A)\rightarrow s(A)$, where $g(A)$ is a predicate over state variables (*the guard* of *A)* and $s(A)$ is a statement of Dijkstra's language of guarded commands [7] (*the body* of *A)*. The action that establishes any postcondition is said to be miraculous. We take the view that an action is only enabled in those states in which it behaves non-miraculously. The guard of the action characterizes those states for which the action is enabled:

$$g(A) = \neg wp(A, false)$$

The actions are assumed to be *atomic*, meaning that only their input-output behaviour is of interest. They can be arbitrary sequential statements. Their behaviour can therefore be described by the weakest precondition predicate transformer of Dijkstra [7]. In addition to the statements considered by Dijkstra, we use non-deterministic choice A [] B between statements A and B, simultaneous execution of statements $A\|B$ provided A and B do not share state variables and prioritizing composition, $A//B$. Note, that the prioritizing composition selects the first action, if it is enabled, otherwise the second (the choice being deterministic):

$$A//B = A\ []\ (\neg g(A)\rightarrow B)$$

The detail description of these operators can be found elsewhere [3,11].

The procedure bodies and the actions may contain procedure calls. As a parameter passing mechanisms we consider *call-by-value* denoted *p(***val** *x)*, *call-by-result* denoted *p(***res** *x)* and *call-by-value-result* denoted p(**valres** *x)*, where x stands for the formal parameters. We assume that the procedures are not recursive. An extensive study of procedures in the action system formalism has been conducted elsewhere [12].

B. Refinement

The main development technique for the action systems is stepwise refinement [2,3,4]. The action A is refined by the action C, written $A \leq C$, if, whenever A establishes a certain postcondition, so does C:

$$A \leq C \text{ iff for all } p:\ wp(A,p) \Rightarrow wp(C,p)$$

A variation of refinement is if A is (data-) refined by C via the relation R, written $A \leq_R C$. For this, assume A operates on the variables a,u and C operates on the variables c,u. The data refinement is defined as follows [2,3]:

$$A \leq_R C \text{ iff for all } p:\ R \wedge wp(A,p) \Rightarrow wp(C, (\exists a \bullet R \wedge p)) \qquad (2)$$

Data refinement allows us to replace the variables in the actions. The relation R defines the correspondence between the replaced variables a and the newly introduced variables c.

When carrying out refinement in practice, one seldom appeals to the definitions (1) and (2). Instead certain pre-proven refinement rules are used.). Instead certain pre-proven refinement rules are used. For instance, *Rule 1* and *Rule 2* below are examples of derived rules for verifying refinement between actions and action systems. Let assume that *A* operates on variables *a,z* and *C* operates on variables *c,z.* Let *R* be the relation over *a,c,z:*

Rule 1: $A \preceq_R C$ iff

(i) $R \wedge g(C) \Rightarrow g(A)$

(ii) $\forall p.(R \wedge g(C) \wedge wp(sA,p) \Rightarrow wp(sC, \exists a \bullet R \wedge p))$

Rule 2: For action systems ***A*** and ***C***, $\boldsymbol{A} \preceq_R \boldsymbol{C}$, iff

(i) Initialization: $C0 \Rightarrow (\exists a \bullet R \wedge C0)$,

(ii) Actions: $Ai \preceq_R Ci$ for all *i*,

(iii) Exit condition: $R \wedge (\vee_i g(Ai)) \Rightarrow (\vee_i g(Ci))$

The proofs of these rules can be found elsewhere [2,3].

The action system formalism has been successfully used in component-based design. The formalism supports three most important modularization mechanisms: procedures, parallel composition and data encapsulation [3,12]. The components specified as action systems can communicate via shared variables, shared actions or remote procedure calls. Let us consider two action systems **A** and **B**

A :: |[**proc** q_1*=Q_1; … q_N*=Q_M;
$pA_1=PA_1$; … $pA_M=PA_N$;
var v*,a • InitA**;**
do A_1 []…[] A_K **od**]| : *z*

B :: |[**proc** r_1*=R_1; … r_S*=R_S;
$pB_1=PB_1$; … $pB_T=PB_T$;
var *w*,b • InitB**;**
do B_1 []…[] B_L **od**]| : *y*

where *v** and *w**, *a* and *b*, *z* and *y* are pairwise distinct. Moreover, the local procedures pA_1, … pA_M, pB_1, … pB_T declared in **A** and **B** are distinct too. The *parallel composition* **A**||**B** of **A** and **B** is the action system **C**

C :: |[**proc** q_1*=Q_1; … q_N*=Q_M;
r_1*=R_1; … r_S*=R_S;
$pA_1=PA_1$; … $pA_M=PA_N$;
$pB_1=PB_1$; … $pB_T=PB_T$
var *v*, w*,a, b • InitA||InitB**;**
do A_1 []…[] A_K [] B_1 []…[] B_L **od**]| : $z \cup y$

Hence, the parallel composition combines the state spaces of the constituent action systems, merging the global variables and global procedures and keeping the local variables distinct. The prioritizing composition of action systems is defined similarly. However, in the resultant action system **A//B** the preferences are given to the actions of the action system **A.** The refinement of component systems is usually carried out by application of the following refinement rules:

Rule 3: $\boldsymbol{A1||A2} \preceq_R \boldsymbol{C1||C2}$ if $\boldsymbol{A1} \preceq_R \boldsymbol{C1}$ and $\boldsymbol{A2} \preceq_R \boldsymbol{C2}$

Rule 4: $\boldsymbol{A1//A2} \preceq_R \boldsymbol{C1//C2}$ if $\boldsymbol{A1} \preceq_R \boldsymbol{C1}$ and $\boldsymbol{A2} \preceq_R \boldsymbol{C2}$ and

$R \wedge g(\boldsymbol{A1}) \Rightarrow g(\boldsymbol{C1})$

The action systems and refinement provide us with a suitable framework for formal specification and verification of the behaviour of autonomic control systems. The aim of this paper is to propose patterns for modelling autonomic control systems that rely of a feedback loop for their self-adaptation.

IV. SPECIFYING AUTONOMIC CONTROL SYSTEM AS ACTION SYSTEMS

A. Autonomic component

We start deriving a formal specification of an autonomic control system from defining a structure of an autonomic manager. Our specification follows the architecture depicted in Fig. 3. Essentially, it can be seen as a sequential composition of the actions modelling data collection, data analysis and error diagnostics, planning of the next control step and setting actuators accordingly. Formally, we define it as follows:

AM :: |[const
eval: DATA x DATA x DATA x STATE
planning : STATE x DATA x PLAN
acting: PLAN x A_STATE
next_exp_state: STATE x A_STATE x DATA
var *int_data, ext_data, ref_data: DATA*
cur_state :STATE
cur_plan :PLAN
act_state:A_STATE
flag: {sen,an,pl,act}•
INIT *int_data, ext_data, ref_data:: DATA*
cur_state := init_state
cur_plan := nil_PLAN
act_state:=idle
*flag: =sen**;***
do
flag=sen -> int_data, ext_data:: DATA || flag:= an
[] *flag=an ->*
cur_state:= eval(int_data, ext_data,ref_data)
|| flag:=pl
[] *flag=pl ->*
cur_plan := planning(ext_data,cur_state)
|| flag:= ac
[] *flag=ac->*
act_state := acting(cur_plan)
|| ref_data:=next_exp_state(cur_state,act_state)
|| flag:= ac
od]|

In the **cons**t clause, we have defined a number of abstract functions. The function *eval* models the analysis of data. Essentially, it compares the data obtained from the internal and external sensors with the reference model. The function *planning* takes as an input the data obtained from the external sensors and the current system state to decide on the next system behaviour. The behaviour is modelled by an abstract set *PLAN*. The function *acting* defines the new states of actuators based on the defined plan and their current state. Finally, the function *next_exp_state* defines the next value of reference model against which the system state will be compared.

In the **var** clause, we have defined the variables of the model. The variables *int_data* and *ext_data* represent readings of internal and external sensors correspondingly. The variable *ref_data* defines the next expected value of the reference model. The variable *cur_state* defines the current state of the system. Finally, the variable *act_state* abstractly models the state of the actuators. The variable *flag* is an auxiliary variable modelling the progress of system execution.

Let us now describe the actions modelling the behaviour of the autonomic component. The first action models the results of monitoring via the external and internal sensors. The internal sensors supply the information regarding the state of the system components. This information can be used to detect occurrence of failure and deviations from the expected behaviour. The external sensors bring the fresh information about the operating environment of the system. For instance, in the case of the autonomic robot, the external sensors will signal about the obstacles detected on route. These data are compared against the reference data. The general mechanism used for the comparison is to ensure that the detected system state matches the expected system state. Based on that comparison, the variable *cur_state* obtains the new value. The next action relies on the results of the analysis to define the current plan. In general, a plan corresponds to a certain mode of the components. As soon as the plan is defined, the new states of the actuators can be computed. By relying on the assigned states of the actuators, we can also compute the expected value of the reference model. This value will be used in the next cycle to diagnose the state of the system.

The actions are executed cyclically. At each iteration, the autonomic manager repeats the same sequence of actions, as defined by our model *AM*.

B. Specification of the autonomic component

Next, we demonstrate how to specify a self-aware component. A self-aware component besides providing its intended functionality also detects errors in its own functioning and raises corresponding exceptions, if the requested service cannot be executed. The self-awareness capabilities not only allow us to build the system in a well-structured way but also enable an effective fault tolerance.

In our initial specification, we assume that functioning modes of the autonomic components are transparent to the entire system. Such an abstraction allows us to significantly simplify the initial specification. The real communication mechanism is introduced at the consequent refinement steps.

When a component is not involved in executing a request sent by the autonomic manager, it is in the state *idle*. The autonomic manager may request a certain service by placing the corresponding data in *req_buffer*. If a component is idle and a request arrives then the component starts to execute this request and enters the state *executing*. Upon completing the requested service the component becomes idle again.

Therefore, the autonomic component **AC** can be specified as follows:

AC :: |[**var** *req_buffer*: Buffer*
resp_buffer:Buffer*
int_exc:Exceptions
state:{idle,executing,failed}
do N || F od]|

where

N :: |[**do** *state=idle /\ req_buffer ≠empty /\ no_exc →*
state:= executing || req_buffer:=tail(req_buffer)
[] $_{i: 1..N}$ *state:= executing /*
no_exc /\ event_i →
reaction_i
[] int_exc := exc
[] reaction_i ||
state:= idle||
resp_buffer:= resp_buffer^result
od]|

F::|[**do**[]$_{j:1..M}$*int_exc→internal_exc_handling||*
rec::{OK,Failed}
[] $_{j:1..M}$ *int_exc/\ rec=OK→ int_exc:= Nil*
[] $_{j:1..K}$ *int_exc/\ rec=Failed→ state:=failed*
[] **od]|**

The action system **N** defines the intended functionality of the autonomic component, i.e., executing requests that are sent to it by the autonomic manager. When a request is chosen for execution, it is deleted from the buffer of requests *req_buffer*. When the autonomic component successfully completes request execution, the result is put into the response buffer *resp_buffer*. The action system **N** models successful service provisioning

Upon detecting an error, an exception *int_exc* is raised and control is passed to the subsystem **F**. The action system **F** specifies handling of errors. If error handling succeeds then the component resumes its normal function, i.e., control is passed to the subsystem **N**. However, if error handling fails, the component state is changed to *failed*.

 ISBN: 978-1-61208-219-6

B. Specification of the overall architecture

We aim at defining the overall system architecture by composing models of components of control systems with the model of autonomic manager. To enforce separation of concerns we introduce self-awareness capabilities into the model of system components.

Assume that $\mathbf{AC} = \{\mathbf{AC_0}, \ldots, \mathbf{AC_M}\}$ is a set of autonomic components. We specify an autonomic control system as a prioritizing composition of the autonomic manager and the action systems specifying the autonomic components:

$$\mathbf{S:: AM//AC_0 \,||\, AC_1 \,||\ldots||\, AC_M}$$

Initially, the components communicate via shared variables. In the refinement process, we replace shared variables by the remote procedures. Further refinement steps (omitted here) allow us eventually decompose the system into independent components.

The proof-based verification associated with the refinement allows us to reason about preservation of important system properties, such as correctness, safety and fault tolerance in presence of self-adaptation. Moreover, by instantiating the abstract functions with their concrete counterpart, we arrive at the additional properties ensuring correct execution of self-adaptation scenarios. In the large distributed systems ensuring these properties at the architectural level brings benefits of early problem discovery. Hence, it allows us to arrive at a more robust design and speed up the development cycle.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed guidelines for structuring formal models of autonomic control systems based on feedback loops. The approach is based on the formal derivation of correct by construction system via stepwise refinement. We demonstrated how to derive system architecture in a formal way and structure specifications of self-aware components and autonomic manager.

A variety of approaches has been proposed to model autonomic control systems (e.g., see [4,5] for review). However, majority of these approaches focus on modeling details of self-adaptation mechanisms or their implementation details. In our work, we aimed at defining high-level architectural guidelines for modeling autonomic systems.

Sensoria project [8] has significantly advanced the area of formal modelling of adaptive systems. It has developed a number of modelling and programming primitives for just-in-time composition. Moreover, mathematical models for reasoning about correctness as well as quantitative analysis have been proposed as well. However, the project mainly aimed at modelling services and service-oriented systems rather than autonomic control systems – the goal that we have pursued in this paper.

As a future work, we aim at validating the proposed approach by a number of case studies. Moreover, it would be interesting to develop patterns for knowledge collection and representation in the formal model of autonomic control systems.

REFERENCES

[1] T. Anderson and P.A. Lee. Fault Tolerance: Principles and Practice. Prentice-Hall, Englewood Cliffs, 1981.

[2] R.J.R. Back, "Refinement calculus, Part II: Parallel and reactive programs". In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg (Eds.), *Stepwise Refinement of Distributed Systems*, pp. 67-93. New York, Springer-Verlag, 1990.

[3] R.J.R. Back and K. Sere, "From modular systems to action systems", *Software –Concept and Tools 17*, pp. 26-39, 1996.

[4] R.J.R. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction*. New York, Springer-Verlag, 1998.

[5] M. Butler, E. Sekerinski, and K. Sere, "An Action System Approach to the Steam Boiler Problem", In J.-R. Abrial, E. Borger and H. Langmaack (Eds.), *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, pp. 129-148, New York, Springer-Verlag, 1996.

[6] F. Cristian. "Exception Handling". In T. Anderson. *Dependability of Resilient Computers*, pp. 68–97. BSP, 1989.

[7] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice Hall, 1976.

[8] EU FP6 Project Sensoria: Software Engineering for Service-Oriented Overlay Computer. http://www.sensoria-ist.eu/

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

[10] N.G. Leveson, *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[11] E. Sekerinski and K. Sere, "A Theory of Prioritizing Composition", *The Computer Journal*, *39*(8), pp. 701-712, 1996.

[12] K. Sere and M. Waldén, "Data Refinement of Remote Procedures", *Formal Aspects of Computing*, *12*(4), pp. 278–297, 2000.

[13] K. Sere and E. Troubitsyna, "Safety Analysis in Formal Specification", *Proc. World Congress on Formal Methods in the Development of Computing Systems*, pp. 1564-1583, Springer, 1999.

[14] E. Troubitsyna, "Developing Fault-Tolerant Control Systems Composed of Self-Checking Components in the Action Systems Formalism", *Proc. The Workshop on Formal Aspects of Component Software*, pp. 167-186, 2003.

 ISBN: 978-1-61208-219-6