INTERNATIONAL BLACK SEA UNIVERSITY

FACULTY of COMPUTER TECHNOLOGIES AND ENGINEERING

DEVELOPMENT OF NEW TECHNIQUES FOR MODELING AND ANALYSIS OF CLOUD COMPUTING SECURITY

Medhat Abouelyazid Madany Mousa

An extended Abstract of Ph.D dissertation in Engineering in Informatics

Tbilisi / 2014

Scientific Supervisor: Prof. Dr. Irakli Rodonaia ------------------------------------

Experts:

1. Dr. Giorgi Ghlonti --

2. Assoc. Prof. Dr. Giorgi Mandaria ---

Opponents:

1. Assoc. Prof. Dr. Nikoloz Abzianidze ---

2. Prof. Dr. Medea Tevdoradze --

Introduction

Cloud computing is attracting great attention nowadays. The elastic nature of cloud makes it suitable for almost any type of organization. The major challenge faced by cloud users and providers are security concerns towards cloud services. The ability of a system to react consistently and correctly to situations ranging from benign but unusual events to outright attacks is key to the achievement of the goals of self-protection, self-healing, and self-optimization.

From the security standpoint there are many serious threats against the objects of cloud computing. Malware attacks may be initiated against any hosts, as a result the performance-based Quality of Service (**QoS**) will be degraded accordingly, especially in case of the attack mechanism that are based on **D**istributed **D**enial **o**f **S**ervice (**D**)**DoS**. The latest achievement in combating these phenomena is implementation of **autonomic** computing paradigm. Autonomic systems exhibit the ability of self-monitoring, self-repairing, and self-optimizing by constantly sensing themselves therefore tuning their performance. The notions of autonomic components and autonomic-component ensembles are considered in this thesis. We also presented the coordinating language for ensemble components, which is used to represent specificity of security issues in autonomic computing environment. To reveal abnormal behavior of autonomic-component ensembles the information theoretical metrics are proposed to use in the approach described in the paper. To realize this approach new classes and methods have been developed within an environment for establishing the proper way of component ensembles interactions. Additionally this research presents a comprehensive study and a new approach for optimization of the hardware resource, which are demonstrated in multiple scenarios and case studies. The suggested framework allows us to express **formally** the desired behavior (**what** is to be implemented, by opposition to **how**) of a given cloud computing system based on rigorous mathematical description. Using formal methods allow us to guarantee the desired behavior or point to conditions that may violate it.

Content of the thesis.

In the first chapter (literature overview), we started with an overview around the main concept of Cloud Computing, by explaining the conceptual reference model. In addition, we presented importance of the formal languages and rigorous explanation within the dynamic networks.

In the second chapter, we addressed the formal languages in terms of their application to cloud autonomic systems. By showing the concepts of SCEL (**S**oftware **C**omponent **E**nsemble **L**anguage -

a kernel language for programming autonomic computing systems) and its run-time implementation jResp, we present also different scenarios that explain and clarify the main contributions of proposed framework. We present the theoretical ground of the approach which is based on the Kolmogorov Complexity metrics, with complementary use of the Kullback-Leibler metrics. How it could be implemented in the actual environment of SCEL with respect to the enforced SLA is also demonstrated in the chapter.

In chapter three, which essentially is based on results obtained in chapter two, formal verification (using Kripke structure and SPIN as our proposed model checking and formal verification tool) in the autonomic cloud computing environment has been precisely reported. In the last section of the research, we ended up with the conclusion and future work that could be an interest of any researcher whoever has an interest in the same direction (modeling of autonomic cloud computing, formal verification and model checking of policy conflict in the autonomic distributed software systems).

Methodology

To achieve objectives set in the thesis, the following techniques and research methods have been used:

- Formal methods approach is widely used in the thesis as the main methodology. The central problem of formal methods is to be able to guarantee the behavior of a given computing system following some rigorous approach. At the heart of formal methods, one finds the notion of specification. A specification is a model of a system that contains a description of its desired behavior—what is to be implemented, by opposition to how.
- Autonomic cloud systems were described by using the specialized formal language SCEL (Software Component Ensemble Language) that enables users to model and describe behavior of service components plus their ensembles, their interactions, their sensitivity and adaptivity to the environment they are working in.
- Kernel notions and elements of SCEL - set of programming abstractions that permit to represent behaviors, knowledge, aggregations according to specific customized policies, and to support programming context-awareness, self-awareness and adaptation - are used in the thesis.
- Autonomic Components Ensembles are based on the idea of virtualization. The term virtualization broadly describes the separation of a resource or request for a service from the underlying physical

delivery of that service. With virtual memory, for example, Computer software gains access to more memory than its physically installed, via the Background swapping of data to disk storage.

- Abstract constructions of SCEL are programmatically implemented with the jResp-Java based implementation of main concepts of SCEL.
- Security issues related to Autonomic Components Ensembles are realized with the informational-theoretical methodology of Kolmogorov complexity metrics and Kullback- Leibler divergence metrics.
- Formal Verification of Autonomic Components ensembles provision of required (SLA) is carried out by using model-checking methodology based on Linear Temporal Logic and software tool SPIN.

Purpose of the Study

Generally, the proposed research is to identify the main threats, which the IaaS and PaaS layers, respectively, are exposed by means of exploring what is behind the cloud computing, its core services and components. The next area of the research is to develop and explore reliable and comprehensive methods of defense from the indicated threats. In this context, main objectives of the research are determined below:

- To explore thoroughly the area via studying related works.
- To understand and clarify the specific purposes and ideas of the distributed and dynamic software systems in terms of violation of certain security policies that affect the overall SLA and the Quality of Service accordingly
- To explore and describe the current challenges of potential threats that may exists against the traditional systems and its transition from the physical generally environment to IaaS & PaaS.
- To identify and investigate the potential attack mechanisms within hypervisors and between different hypervisors
- To provide some recommendations regarding the risks mitigation and ways of building a highly secured infrastructure
- To analyze the associated issues in terms of the self-managing components within the distributed software systems and give some optimized solutions for improving the whole infrastructure.
- To develop and implement formal methods and techniques for modeling and analysis security issues in autonomic cloud computing systems.

- To develop algorithms that capable to handle the computing processes across the large distributed and dynamic computing grids, which are Autonomic Computing (AC)-based systems, as well as interaction between component ensembles with their relations to Distributed Resource Algorithm (DRS)
- To develop solutions for AC infrastructure enabling applications, running on components of particular ensemble, to move under some circumstances to other healthy hosts within the datacenter
- To develop formal verification methods termed *model checking* that can be used to detect conflicts in autonomic computing policies.

The Main Contributions and the Scientific Novelty:

- A new approach to the analysis and evaluation of rapidly growing field of IT-Autonomic Cloud Computation (**ACC**) - is offered in the thesis. ACC represents a promising approach to achieving high level of adaptiveness, and expressiveness by adding self-management capabilities to the components of IT systems. Rigorous mathematical techniques termed formal methods to improve the predictability and dependability of AC is proposed and employed in the thesis. Formal methods offer appropriate abstractions to deal with the large dimension of autonomic cloud systems, and with their need to adapt to the changes of the working environment and to the evolving requirements.
- A set of programming abstractions that permit to represent behavior, knowledge, and aggregations according to specific policies, and to support programming context-awareness, self-awareness and adaptation have presented in the thesis. Based on these abstractions, the Software Component Ensemble Language (**SCEL**), as well as a kernel language for formal reasoning on autonomic systems behavior is used in a new context – provision of security in terms of revealing malware threats such as **DDOS**.
- The formal constructions of SCEL are updated and complemented by newly developed objects, which allow researchers to create and formally investigate scenarios of various malware threats in rigorous mathematical manner.
- A new approach (and corresponding constructions) for interactions between AC and their behavior is proposed in the thesis. It allows autonomic components numerically estimate level of security threats on regular basis and undertake the relevant actions. Namely, web services running on AC-

VMs in case of increased threats can operatively move from current (insecure) components to secure ones. Moreover, this is very important to maintain the required Service Level Agreement (**SLA**).

- The informational- theoretical metrics –Kolmogorov complexity (**KS**) –levels of security threats in the new, original context have measured in numerically estimate using KS. The KS is used by component's knowledge interfaces and it is computed by component's Autonomic Managers AMs. Unlike known ways of using KS in IT systems, autonomic independent and parallel computation of KS for all existing components, based on so-called numerical estimation of packet flows.
- Although KS is not computable, the approximation of KS by using string compression mechanisms is proposed. New programmatic objects of jResp (Java implementation of the SCEL) are developed and added to implement this approximation of KS. Namely, new classes and methods "*complexitySensors*" are developed.
- A new interpretation of KS. It allows users to interpret KS in probabilistic manner (as known, generally KS is informational-theoretical notion, not probabilistic) and use it for formal verification of autonomic-component ensembles.
- A new structural model of formal technique termed *model checking* is proposed to detect conflicts in AC policies and SLA breaches. This technique essentially uses the proposed interpretation of KS. Namely, the probabilistic interpretation of KS is used in the thesis to numerically estimate distribution of response time including (downtime time) of VMs and essentially refine the migration time of VMs, which is important for detection of SLA violation.

Practical Implications and Importance

The main purpose of the research is to produce a relevant framework that could be adopted in modern IT infrastructure (see section 2.16). The contributions of the research is intended to be the main stream in research of autonomic cloud computing. The presented work can be further developed for security and policy conflict verifications within the autonomic computing environment

Structure and volume of the work

The thesis study is 178 pages and consists of 3 chapters, a list of references and list of figures and list of tables.

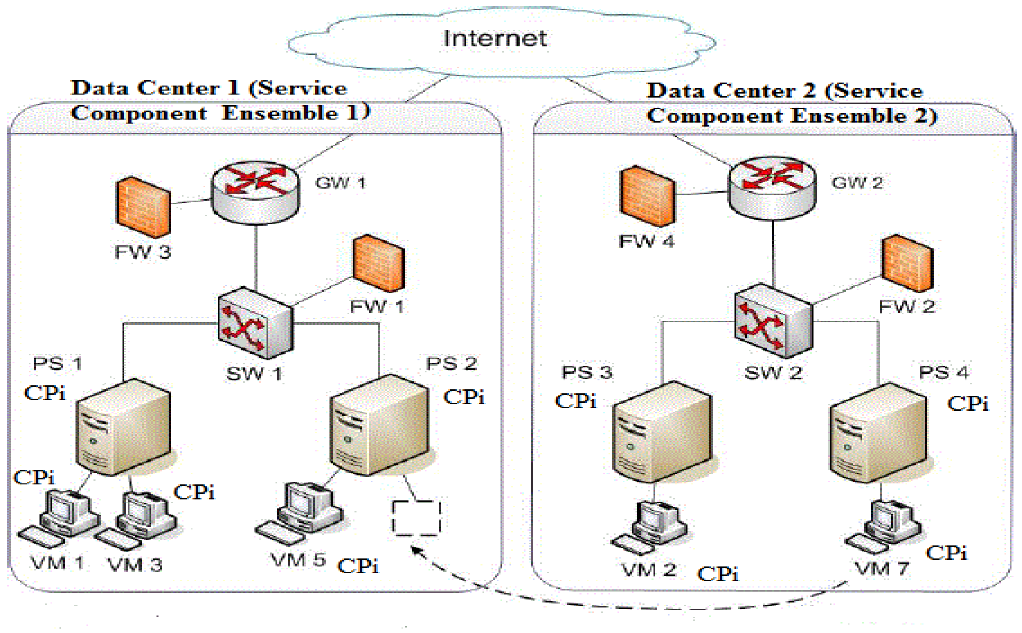
Definition of the Problem

The notions of autonomic components (ACs) and autonomic-component ensembles (ACEs) have been put forward as a means to structure a system into well-understood, independent, and distributed building blocks that interact in specified ways. ACs are entities with dedicated knowledge units and resources that can cooperate while playing different roles. Awareness is made possible by providing ACs with information about their own state and behavior that can be stored in their knowledge repositories. These repositories also enable ACs to store and retrieve information about their working environment, and to use it for redirecting and adapting their behavior. Each AC is equipped with an interface, consisting of a collection of attributes, such as provided functionalities, spatial coordinates, group memberships, trust level, response time, etc.

Attributes are used by the ACs to dynamically organize themselves into ACEs. Individual ACs not only can single out communication partners by using their identities, but they can also select partners by exploiting the attributes in the interfaces of the individual ACs. Predicates over such attributes are used to specify the targets of communication actions, thus providing a sort of attribute-based communication. In this way, the formation rule of ACEs is endogenous to ACs: members of an ensemble are connected by the interdependency relations defined through predicates. An ACE is therefore not a rigid fixed network but rather a highly dynamic structure where ACs' linkages are dynamically established. The proposed abstractions are the basis of SCEL (Software Component Ensemble Language), a kernel language for programming autonomic computing systems.

The following scenario is considered. A singleton application currently runs on one of the VMs at Data Center 2 (VM7 in Service Component Ensemble 2).. This application runs alone on its node and,since the application is a singleton, no additional instances can be spawned. During the sessionthe application experiences consistently high CPU load. This increase may be caused either by legitimate traffic overload or by coordinated attacks (DDOS) launched against the PaaS provider. The latter might be wrongly assumed legitimate requests and resources would be scaled up to handle them. This would result in an increase in the cost of running the application (because provider will be charged by these extra resources) as well as a waste of energy.

Figure 1: Cloud Platform and CPi Migration



Source: Pandey, Voorsluys, Niu, Khandoker, & Buyya, 2012, Page- 48

Hence, it is necessary to distinguish between these two cases, the earlier this distinction is made, the higher is the degree of protection of the application from failure and poor performance.To provide this protection, the following security measures are suggested. The traffic flows through the node (CPi) has to be analyzed using Kolmogorov complexity metrics (see later in the text). During the session the constant monitoring of the metric (by the special probe implemented in the separate module), along with measure of CPU load, is being executed. If the simultaneous increase of these two metrics is registered at least in 3 successive time units, the conclusion about the real treat of the DDOS attack must be drawn. As a result, the application has to migrate from the CPi where it was running to another CPi (which may belong to the same ensemble or other ensemble). A new CPi must be found according to some requirements: complexity level and CPU load must be rather low, integrated hardware index (which includes such indicators as processor speed, available memory, available disk space, number of cores, etc) must correspond to the application resource requirements (they are published in the interface of the CPi where the application is running). If the required CPi is found, the application has to migrate there as soon as possible and stop its running on the "old" CPi. The process formally can be described in SCEL statements. We assume that, other than id, the interfaces and provide the attributes "ComplexityLevel", "CPULoad" and "Memory" stores a context information, updated by the underlying infrastructure (usually, from the firewalls, gateways or special probes) and are `sensed' by the managed element. The CPi where the application is running is the SCEL ($\mathcal{I}[\mathcal{K}, \Pi, AM[ME]]$

The autonomic manager AM is defined as follows: AM $\triangleq$ $P_{ComplexityMonitor}$ [$P_{CPULoad}$ [$P_{migrateCP}$]]

P$_{ComplexityMonitor}$ $\triangleq$ **qry**("*ComplexityLevel*", "high") @ self.

get("*ComplexityHigh*", false) @self.

put(("*ComplexityHigh*", true) @self. **qru**("*ComplexityLevel*","low")@self.

get("*ComplexityHigh*",true)@self.

put("*ComplexityHigh*",false)@self.$P_{ComplexityMonitor}$

P$_{CPULoad}$ $\triangleq$ **qry**("*CPUloadLevel*","low")@ self. **get**("*CPULow*", false) @self.

put(("*CPULow*", true) @self. **qru**("*CPUloadLevel*","high")@self.

get("*CPULow*",true)@self.

put("*CPULow*",false)@self. $P_{CPULoad}$

P$_{MigrateCP\ i}$ $\triangleq$**qry**("*Cloud service*", ?X)@ self

get("*Cloud service_args*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*) @self.

/* retrieving from the knowledge repository the process implementing a required functionality id and bounding it to a process variable X */

/* searching an item *c* among components belonging to the ensemble identified by predicate Ω */

qry("*CPiId*", ?c) @Ω.

/* storing actual parameters of the process to be executed in the found component c : moving from VM7 to /*MV5 on fig.2 */

put("*Cloud service*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*)@c

get("*Cloud service*", "*sessionId*", "*terminated*") @self.

/* removing the process from the knowledge repository of 'old' CPi */

get("*Cloud service*", "*sessionId*", X) @self.nil

/* eliminating the process in 'old' CPi */

Here the predicate Ω is determined as follows:

Ω ($\mathcal{I}$) =($\mathcal{I}$.ComplexityLevel="low") $\wedge$ ($\mathcal{I}$ .CPULoad <75) $\wedge$ ($\mathcal{I}$ .Memory>=500))

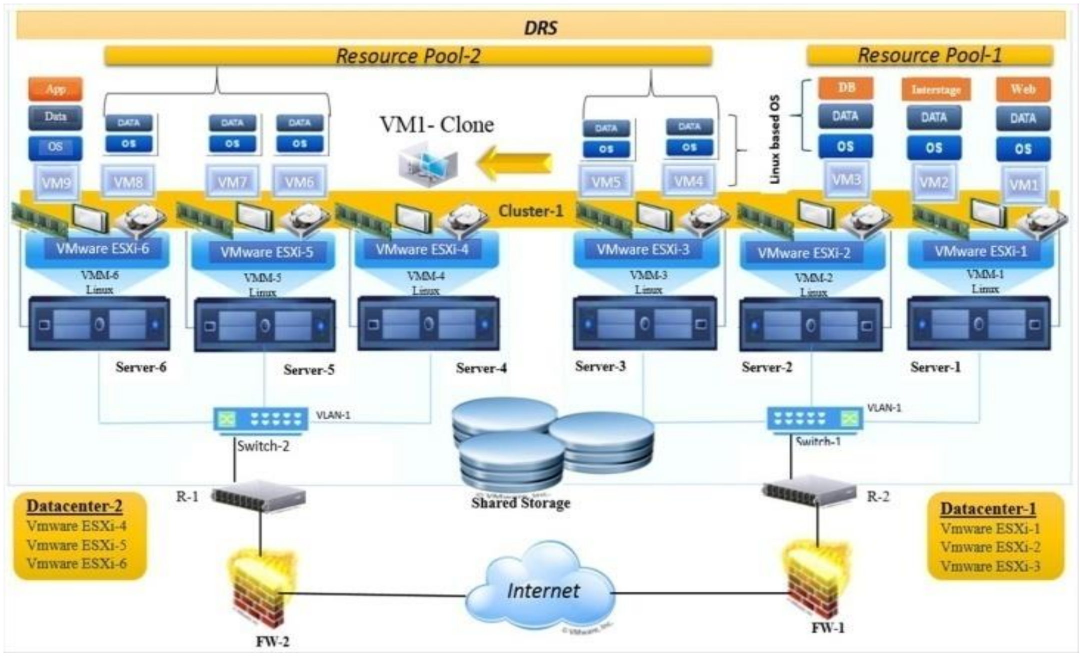
and is used for group-oriented communication in the action qry("CPiId", ?c) @. This predicate defines the ensemble of components which publish in their interfaces attributes "ComplexityLevel", "CPULoad" and" Memory"along with relevant values. We assume that these attributes are provided by the interface of each component and obtain dynamically updated values from corresponding probes (sensors) as a result of constant monitoring (sensing) of the computing environment. We assume also that the attribute "ComplexityLevel" gives an indication in the range [0:1] of the complexity level (see explanation below in the text) of data flow through the ensemble, the attribute "CPULoad" – in the range [0:100], the attribute "Memory" – in the range [0:1000]. In this context the meaning of the predicate is as follows: finΩ a component CPi (or components) where the "ComplexityLevel" is low (i.e. less than 0.15), "CPULoad" is less than 75 and available memory index "Memory" is more than 500.

Independently of the service component on which the cloud service is being executed ("old ‘ CPi or newly found "receiver of migrated service" CPi) the SCEL statements which describe the process P_s executed by the managed element ME are as follows:

> $\mathbf{P}_s$ ≜ **get** ("*Cloud service*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*)@self.
> **get**("*CPUload*", ?L) @self.
> /* L is a current CPUload of the component
> **get**("*memory*", ?M) @self.
> /* M is a current allocated memory
> **put**("*CPUload*", (L+ *CPUValue*))@self.
> **put**("*memory*", (M- *memoryValue*))@self.
> $\mathbf{P}_s$ [X(*sessionId, memoryValue, CPUValue*)]
> /* the new process (additionally to the already running process Ps), having actual parameters sessionId, memoryValue, CPUValue, starts */

In the thesis the run-time Java implementation of the SCEL formal code (expressed in jResp environment) has been developed. The main classes of jResp, corresponding to the above SCEL code, are as follows: **ServiceComponent, CloudService, ServiceCaller, RequestHandler, OfferAgent**. The scenario, described above, is realized by means of jResp classes **Scenario** and **Main** (the structure of datacenter):

Figure.2 Structure of the datacenter.



```
public class Main {
        private Scenario scenario;
        private Random r = new Random();
        private Target id;
          public Main(int size){
                this( new Scenario(size, new Random() , 900, 1000, 99, 100) );
        }
/* above the minimum and maximum values of components' memory (900 and 1000,
/* respectively) and minimum and maximum values of components' CPU rates (99 and
/* 100, respectively) are defined


          public Main(Scenario scenario) {
                this.scenario = scenario;
                instantiateNet();
         }
```

```
private void instantiateNet() {
          Random r = new Random();
          SimulationScheduler sim = new SimulationScheduler();
          SimulationEnvironment env = new SimulationEnvironment(sim, new
               RandomSelector(r), new DeterministicDelayFactory(1.0) );
          sim.schedulePeriodicAction(new SimulationAction() {

         VirtualPort vp = new VirtualPort(10);
         Hashtable<String, Node> nodes = new Hashtable<String, Node>();
         for(int i=0; i<scenario.getSize(); i++){
                         Node n = new Node("ServiceComponent"+i, new TupleSpace());
                    n.addPort(vp);
                          n.addSensor(scenario.getComplexitySensor(i));
                          n.addSensor(scenario.getCpuSensor(i));
                          n.addSensor(scenario.getMemorySensor(i));
                          n.addActuator(scenario.getServiceInvocationActuator(i, n));
                          n.addAttributeCollector(scenario. getComplexityAttributeCollector  ());
                          n.addAttributeCollector(scenario.getCpuLoadAttributeCollector());
                          n.addAttributeCollector(scenario.getCpuRateAttributeCollector(i));
                          n.addAttributeCollector(scenario.getMemoryAttributeCollector())
                          n.put(new Tuple("REQUEST", 1, new CloudService("1", 10, 2.0),
                          n.getLocalAddress() ));
                          n.put( new Tuple( "LOCATION" , n.getLocalAddress() ) );
                          Agent a= new RequestHandler();
                          n.addAgent(a);
                          a=new OfferAgent();
                         n.addAgent(a);
                        nodes.put(n.getName(), n);
       }
       for (Node n: nodes.values()) {
              n.start();
               /* for simulation option:
               /* env.simulate(10000);

            public static void main(String[] args) {
          int size=8; /* Here 8 components (virtual machines) start
             /* high complexity level (=1, no security threats) is assigned to  all virtual   machines
          ServiceComponent c0 = new ServiceComponent ( 0 ,1, 0 , 1 );
          ServiceComponent c1 = new ServiceComponent ( 1 , 1, 100 , 1 );
          ServiceComponent c2 = new ServiceComponent ( 2 , 1, 100 , 1 );
             ……………………………………………………………………………
             ServiceComponent  c7= new ServiceComponent (7, 1, 100,1);

          new Main( new Scenario( c0 , c1 , c2, c3, c4, c5, c6, c7 ));
        }
      }
```

In order to include into autonomic components ensembles the process of regular monitoring and updating of Kolmogorov complexity level, we need to make significant changes in jResp sensors implementation. Regularly recalculated and obtained Complexity metrics attributes have to be incorporated in knowledge space of components. For this reason, the necessary changes are shown and described. First, all attributes sensors are extensions of base class **AbstractSensor** (which is one of the basic classes of jResp):

```
public abstract class AbstractSensor extends Observable {
            protected String name;
            protected Tuple value;
            public AbstractSensor(String name) {
            this.name = name;
      }
            public String getName() {
            return name;
      }
```

```
		public final Tuple getValue() {
		return value;
	}
		public final void setValue( Tuple t ) {
		this.value = t;
		this.setChanged();
		this.notifyObservers(t);
	}
}
```

In order to regularly update complexity level attribute, components' autonomic managers have to connect to the netflow enabled router (see later) as sensor client using socket technology. On the other hands, the router has to get this connection (through socket technology).

public class ComplexitySensorClient extends AbstractSensor {

```
	  private String serverAddress;
	private int serverPort;
	private Gson gson = RESPFactory.getGSon();
	private long refreshTime

    public ComplexitySensorClient t(String name , String serverAddress , int serverPort , long
refreshTime )
	  throws IOException {
		super( name );
		this.serverAddress = "148.169.2.45"; /* this is a network address of the router
		this.serverPort = 1024;
		this.refreshTime = refreshTime;
		new Thread( new SensorThread() ).start();
	}

    public class SensorThread implements Runnable {
		@Override
```

```
public void run() {
        while (true) {
                Socket s;
                try {
                        System.out.println(getName()+" requests a complexity
                         value...");
                        s = new Socket(serverAddress,serverPort);
                        BufferedReader reader = new BufferedReader(new
                        InputStreamReader(s.getInputStream()));
                        Tuple t = gson.fromJson(reader, Tuple.class);
                       /* Json for serialization of tuples and messages
                        reader.close();
                        s.close();
                        System.out.println(getName()+" delivers a complexity
                         value...");
                        setValue(t);
                        } catch (UnknownHostException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                setValue(null);
                try {
                        Thread.sleep(refreshTime);
                } catch (InterruptedException e) {
                        e.printStackTrace();
                        return ;
                }
```

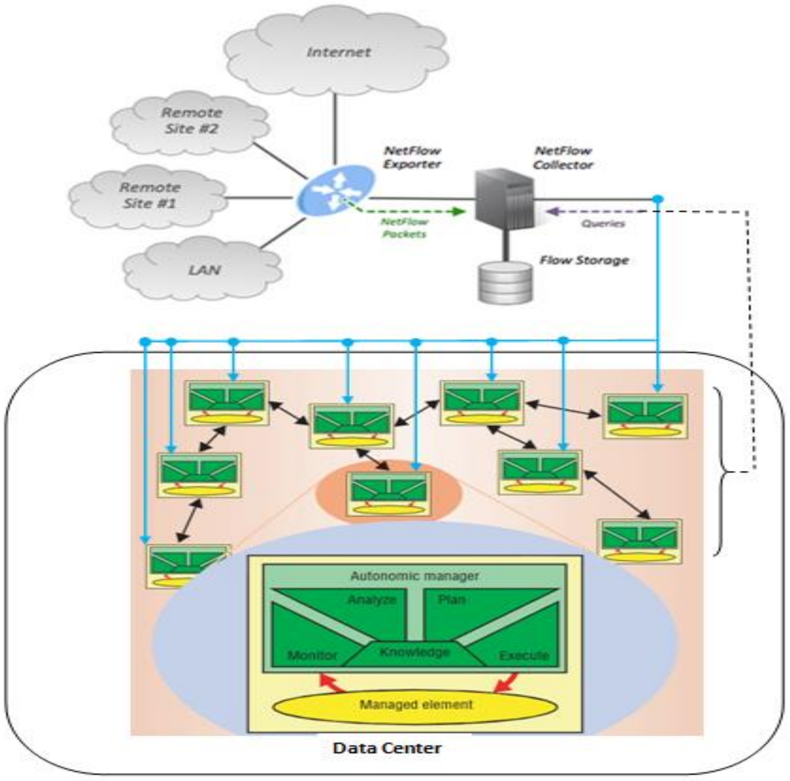
```
                    }
                }
            }
        }
```

The class **ComplexitySensorServer** allows remote components to access to a sensor via a network connection and retrieve complexity level. Implementation of the class is similar to the **ComplexitySensorClient** (again through socket technology). As a result, updating of the complexity level attribute which occurs in the method **updateSensorsValue()** of the **ServiceComponent** class is performed regularly (after specific time period).

In the approach to autonomous computing security and anomaly detection, developed in the thesis, the notions of **netflows**, their **informational-theoretical metrics** and components' **autonomic manager** are essentially leveraged. A network **flow** can be defined in many ways. In a general sense, a flow is a series of packets with some attribute(s) in common. Each packet that is forwarded within a router or switch is examined for a set of IP packet attributes. These attributes are the IP packet identity or fingerprint of the packet and determine if the packet is unique or similar to other packets. All packets with the same source/destination IP address, source/destination ports, protocol interface and class of service are grouped into a flow and then packets and bytes are labeled. This methodology of fingerprinting or determining a flow is scalable because a large amount of network information is condensed into a database of netflow information called the netflow cache.

A ***netflow-enabled device (netflow exporter***: router or switch) sends to the ***netflow collector*** (fig.3)single flow as soon as the relative connection expires. Packets captured by the netflow collector are stored to a ***flow storage***. Port and address IP distributions are highly correlated in network traffic. For this reason, we only considered source and destination IP.

Figure 3: Interaction between NetFlow devices and autonomic components



Source: Haag, User Documentation nfdump & NfSen , 2014

Flows accumulated at the flow storage, are then subdivided into ***component flows***. That is, flows, which have the component's IP address as a destination address, are grouped and sent to the corresponding component (more exactly, to the ***autonomic manager*** of a component - these flows are marked with blue arrows on the fig.3). After receiving their destined flows, the component's autonomic manager can start the processing in order to reveal the abnormal behavior of flows in accordance with the technique, which will be described further. In the thesis the different files with the particular titles (relevant to the concrete SCP_i's IP addresses) to store component flows are used.

Stated as simply as possible, it is hypothesized that information, comprising observations of actions with a single root cause, whether they are faults or attacks, is highly correlated. Highly correlated data has a high compression ratio. The **Kolmogorov Complexity**, K(x), of a string of data measures the size of the smallest program capable of representing the given piece of data. It measures the degree of randomness for the given data. The length of the shortest program to generate a completely random string is equal to the size of the string itself. For all other cases, it is smaller than the size of the string and the program size becomes smaller as more regularity or pattern is discernible from the string. A side effect of this measure is its ability to represent the correlation between disparate pieces of data. This side effect is exploited to design an effective method for detecting DDoS attacks. The DDoS attack detection algorithm makes use of a fundamental theorem of Kolmogorov Complexity that states for any two random strings X and Y,

$$K(XY) \leq K(X)+K(Y)+c,$$

Where K(X) and K(Y) are the complexities of the respective strings, c is a constant and K(XY) is the joint complexity of the concatenation of the strings. Simply put, the joint Kolmogorov complexity of two strings is less than or equal to the sum of the complexities of the individual strings. The equivalence holds when the two strings X and Y are totally random, i.e. they are completely unrelated to each other. Another effect of this relationship is that the joint complexity of the strings decreases as the correlation between the strings increases. Intuitively, if two strings are related, they share common characteristics and thus common patterns. In terms of detection of DDoS attacks, the property given by inequality (1) is exploited to distinguish between concerted denial-of-service attacks and cases of traffic overload. The assumption is that an attacker performs an attack using large numbers of similar packets (in terms of their type, destination address, execution pattern etc.) sourced from different locations but intended for the same destination. Thus, there is a lot of similarity in the traffic pattern. A Kolmogorov complexity based detection algorithm can quickly identify such a pattern. On the other hand, a case of legitimate traffic overload in the network tends to have many different traffic types. The traffic flows are not highly correlated and appear to be random. Therefore, the algorithm samples every distinct flow of packets (distinguished by their source and destination addresses) to determine if there is a large amount of correlation between the packets in a flow. If it is determined to be so, then all suspicious flows at the node are again correlated with each other to determine that it is indeed an attack and not a case of a traffic overload.

While it is known that, in general, Kolmogorov complexity is not computable, various methods exist to compute estimates of the complexity. It is possible to obtain an upper bound of it (which in practice is a very good approximation) by using a universal compression algorithm, like Lempel and Ziv The component's autonomic manager is used for monitoring the corresponding (its own) flows and calculating estimates of the complexity. A component's session trace consists of all packets of the flows during some specified time (session). These traces are concatenated and converted into ASCII string (using tools *nfdump*, *tcpdump* or *Win_RK*). Then this string *s* is compressed using the Lempel-Ziv algorithm (LZW.java), thus obtaining a new shorter string s^1. Now the Kolmogorov complexity (KC) as the ratio between the length of s and the length of s^1 can be calculated. Given that the length of s^1 can never be longer that the length of *s*, the KC is always between 0 and 1.

To estimate the possible range of KS metrics' value that indicates a lot of similarity in the traffic pattern (thus, large numbers of similar packets (in terms of their type, destination address, execution pattern, etc., which is suspicious from the standpoint of DOS or DDOS attack), numerous simulation experiments were carried out in the thesis.. The well-known simulation tool **CloudSim** - a framework for modeling and simulation of cloud computing infrastructures and services – has been used. The goal of the simulation was the determination of affects that had different options on autonomic packet flows between components SCP_i ensembles. As a result of simulation experiments we determined that the range of Kolmogorov Complexity metrics' values that indicates a rather high level of DDOS threats is in the range 0÷0.15. This value is accepted in the thesis as a serious security threat.

To reduce the probability of so-called "false positive alarm" (that is, when alarm of DDOS attack treat arises in conditions of regular "healthy" traffic) another information theory metrics, namely, Kullback-Leibler divergence metric is used in the thesis. In case when both metrics (Kolmogorov Complexity level and Kullback-Leibler divergence metric) have relevant low values, then we assume that the probability of DDOS attack is extremely high. In jResp the computation of the Kullback-Leibler divergence metric is executed again in the class **ComplexitySensorServer** (similarly as it is computed for Kolmogorov Complexity metrics).

The IT infrastructure provided by the datacenter owners/operators must meet various SLAs established with the clients. The SLAs may be resource related (e.g., amount of computing power, memory/storage space, network bandwidth), performance related (e.g., service time or throughput), or even quality of service related (e.g., 24-7 availability, data security, percentage of dropped requests). A datacenter comprises a large number of potentially heterogeneous servers chosen from a set of known and well-characterized server types. In particular, servers of a given type are modeled by their processing capacity and (C_*^p)n memory size a(C_*^m)l as their operational expense (energy cost), which is proportional to their average power consumption. Each client produces one or more VMs, which are executed on some servers in the datacenter. Each client has also established an SLA contract with the datacenter owner. Performance of each client in the cloud computing system should be monitored and necessary decisions should be taken to satisfy the SLA requirements. A client in the system is application software that can produce a number of requests in each time unit. To model the response time of clients, we assume that the inter-arrival times of the requests for each client follow an exponential distribution function similar to the inter-arrival times of the requests in the e-commerce applications. The minimum allowed inter-arrival time of the requests is specified in the SLA contract. The exponential distribution function is used to model the service time of the clients in this system. Based on this model, the response time distribution of a VM (placed on server j) is an exponential distribution with mean:

$$\bar{R}_{ij} = \frac{1}{C_j^p \phi_{ij}\mu_{ij} - \alpha_{ij}\lambda_i} \tag{1}$$

Where µij denotes the service rate of the ith client on the jth server when a unit of processing capacity is allocated to the VM of this client. The VM unit is defined as the basic unit of virtual resource, which is associated with a set of physical resources such as CPU time, main memory, storage space, electricity etc. In real cloud systems, any virtual resource a customer can apply should be a multiple of the VM unit.

As it was stated above, in case of high probability of the DDOS attack the immediate migration of the component from the VM (where the component is being run currently) to another VM (which is to be selected by using the ensemble's components autonomic managers' knowledge base and issuing the special SCEL statement **qry**) is required. The time of migration must be taken into account when determining the response time based on (1). Migrating a VM between servers causes a downtime in the client's application. Duration of the downtime is related to the migration technique used in the datacenter. The downtime also is the function of the link speed and VM memory size.

In the thesis the following approach to update response time distribution of a VM (1) is implemented. Although the Kolmogorov complexity (KS) by its original definition is not the probabilistic notion (it is an informational-theoretical one), we use the KS in the form obtained by using Lempel-Ziv compression algorithm (which converts the original KS indicator into the values in range 0 ÷ 1). Now we can consider the indicator as some approximation of probability of existence of malware (DDOS) threats. Namely, we estimate the probability of the fact that a *j*th server allocated to the *i*th client's VM is exposed to the malware threat of attack (and needs to migrate) as the value : Prob($MigrVM_{ij}$) =1-KS_{ij} . Then the formula (1) must be updated by adding the term representing the expected downtime of the VM_{ij}:

$$\overline{R}_{ij} = \frac{1}{C_j^p \varphi_{ij} \mu_{ij} - \alpha_{ij} \lambda_i} + (1 - KS_{ij}) * DT_{ij}(LinkSpeed) \text{..........}(2)$$

Here we assume that VM memory size is constant and is equal to 1,024 MB. This value of downtime will be essentially used in the approach of **formal verification** of autonomic computing system developed in the thesis.

Autonomic systems are defined as systems that “manage themselves according to an administrator's goals". The effectiveness of the self-management depends on the quality of the autonomic computing policies used to express these goals. Poorly defined policies lead to ineffective self-management; conflicting policies can be downright damaging to the autonomic system. A formal technique termed **model checking** can be used to detect conflicts in autonomic computing policies. Model checking represents a formal technique for verifying whether a system satisfies its specification. The technique involves building a mathematically based model of the system behavior, and checking that system properties specified formally in a **temporal logic** hold within this model. For each refuted property, the technique yields a counterexample consisting of an execution path for which the property does not hold. The result is based on an exhaustive analysis of the state space of the considered model - a characteristic that sets model checking apart from complementary techniques such as **testing** and **simulation**.

The system model most commonly used in model checking is termed **a *Kripke structure***. It consists of a state transition graph $M = (S, S_0, R, L)$, where S represents the finite set of states in which the system can exist, $S_0 \subseteq S$: set of the initial states, is a relation that defines all possible transitions between states, and $L : S \rightarrow 2^{AP}$ ing function that labels each state with the set of atomic propositions that are true in that state.

Commonly used temporal logics include *linear temporal logic* (LTL). The approach to verifying autonomic computing policies described in the thesis uses LTL, which is a logic that adds the temporal operators in Table 2 and calculation for them to first-order logic

$\bigcirc\varphi$	φ is true in the *next* moment in time
$\square\varphi$	φ is true in *all* future moments
$\diamondsuit\varphi$	φ is true in *some* future moment
$\varphi\,\mathcal{U}\,\psi$	φ is true *until* ψ is true

Table 2. Basic rules of Linear Temporal Logic (LTL)

An LTL formula such as φ and ψ in this table is a combination of atomic propositions, logical operators and the LTL temporal operators ○ , □ , ◊ and U . Given a Kripke structure $M \models [](\Diamond a)$ $M=(S,S_0,R,L)$ and an LTL formula ϕ , the notation $M \models \phi$ is used to state that the system model M satisfies the LTL formula ϕ . For example, if $s \in S$ is an atomic proposition, $a \in AP$ states that a is eventually true on the every path from each state .

The approach used in the thesis to verifying autonomic computing policies requires that two types of information are available for the autonomic system:

- A *structural model* that specifies the system parameters that need to be monitored or controlled for the considered application, and their value domains
- A *performance model* that defines the relationships between the system parameters defined in the structural model, and between these parameters and any internal parameters that the system may have.

The steps involved in building the Kripke structure and the LTL formulas are detailed below:

1. The set of system states S and the set of initial system states $S_0 \subseteq S$ are derived from the structural model of the autonomic system.
2. The labelling function $L: S \rightarrow 2^{AP}$ extracted from the performance model for the system.
3. The state transition relation $R \subseteq S \times S$ is defined by the operation rules (i.e., by the action policies) for the system.
4. Finally, system constraints specified by a goal policy correspond to LTL formulas φ that model M must always satisfy: M ⫤ □ . Likewise, the final-state conditions expressed by goal policies correspond to LTL formulas y that model M must “eventually” satisfy: M ⫤ ψ

The ◊ssertions M ⫤ □ φ and M ⫤ ◊ ψ obtained in step 4 are verified using a standard LTL model checker. If these assertions are true, then the policy set is conflict free, i.e., its action policies take the system from any initial state to a valid final state transitioning only through intermediate states that satisfy the invariants specified by the constraint goal policies. If one or more assertions are not true, the policy set contains conflicts. The counterexample generated by the model checker for each such assertion can be used to identify reachable states that do not comply with the system invariants and/or unreachable final states, and thus represent a starting point for resolving the policy conflict. To illustrate the application of model checking to conflict detection in autonomic computing policies, we consider a case study that presents VMs migration described above.

Structural model. The monitored and controlled data-centre parameters relevant for this case study are:

Number of homogeneous servers within cluster = 6;

Capacity of a single server in cluster : $C_j^p = 3\,\text{GHz}$ (assumed as a unit of server capacity) $C_j^m = 8$ GB of RAM; j=1÷6

Number of clients' classes k=2;

Number of clients assigned to each server is randomly generated between 1 and 3, each client is randomly picked from one of two classes;

Portion of the ith client's request served by the jth server (host of a VM) α_{ij} is uniformly selected between 0 and 1;

Average request rates of the ith client λ_i are chosen uniformly between 0.1 and 1 request per second (the minimum allowed inter-arrival time of the requests is specified in the SLA contract).

Processing capacity of the jth server allocated to the ith client's VM is calculated as $C_j^p \phi_{ij}$, where ϕ_{ij} is

the portion of processing resources of the jth server that is allocated to the ith client (assumed to be equal to 1/(number of clients) assigned to the server jth);

Service rate μ_{ij} of the ith client on the jth server (of capacity 1), that is, μ_{ij} is the service rate of the ith client on the jth server when a unit of processing capacity is allocated to the VM of this client; μ_{ij} are set based on the highest clock frequency for the servers.

To demonstrate the principal capabilities of model checking we use a simplified version of the above-mentioned model: just one client is assigned to each server (single VM runs on each server, namely, VM7 in Service Component Ensemble 2 of datacenter 1 [2]), α_{ij}=1, ϕ_{ij}=1, $\lambda=\lambda_{ij}$ is uniformly distributed between 0.1 and 1 request per second, $\mu = C_j^p \phi_{ij} \mu_{ij}$ is uniformly distributed between 1 and 3 request per second.

So, the response time (1) for the simplified case looks like $R_1=1/(\mu-\lambda)$ and the total response time will be equal to :

$R=1/(\mu-\lambda) + (1 - KS)*DT(LinkSpeed)$ (3)

For the structural model monitored and controlled datacenter parameters are:

- The request rate $x > 0$ (i.e $x = \lambda$)
- The service rate $y > 0$ (i.e $y = \mu$)
- The Kolmogorov complexity metric $z > 0$

We will assume that $0.1 \le x \le 1$, $1 \le y \le 3$, $0 \le z \le 1$

Performance model. There are "internal" parameters (i.e. parameters that are neither monitored nor controlled, but are calculated based on the parameters defined in the structural model):

Total response time

$$T = 1/(y-x) + (1-z)*DT$$

Goal policies. The average total processing time should eventually be in accordance with SLA requirements: $T \le T_{SLA}$=4sec.

Policy verification. To verify the correctness of this policy set, we need to construct the Kripke structure and to derive the LTL formulas associated with the structural and performance system models,

1) Constructing Kripke structure $M = (S, S_0, R, L)$ the autonomic data centre. Each combination of values that can be taken by the monitored and controlled parameters of the system corresponds to a different state . The mc $s \in S$ d parameters are x , y and z , so the set of states is

$$S \equiv \{ (x,y,z) \mid 0.1 \le x \le 1, 1 \le y \le 3, 0 \le z \le 1\}$$

The set of initial states $S_0 \equiv \{(x, y, z \mid 0.1 \le x \le 1, 1 \le y \le 2, 0 \le z \le 0.5\}$

Now we define the set of atomic propositions AP for the Kripke structure by including in AP atomic propositions representing the values of the configuration parameters for the system. We use the notation [X = a] as an atomic proposition stating that the value of the parameter X is a . In addition, we need to determine the truth values for the conditions representing the invariants $T \le T_{SLA}$ defined by the goal policies for the system. Therefore, we also include within AP atomic propositions of the form $[Y \le a]$, where Y is variable and a is a constant . As a result, the complete set of atomic propositions AP used to define the labelling function $L : S \rightarrow 2^{AP}$ consists of all atomic propositions:

$AP=\{[X=a] \mid X \in \{x, y, z, \}, a \ge 0\}, \cup \{ [Y \le a] \mid Y \in \{ T \}, a \ge 0\}$

For example, given the state s =(0.1, 1.2 , 0.15), in which the server (host) of VM7 is receiving user requests at 0.1 transactions /second, service rate of the server is 1.2 services/second and Kolmogorov complexity metric (currently calculated for the server) is 0.15 , several atomic propositions that hold in state s are [x=0.1], [y=1.2] and [z=0.15].

The variation of metrics *x*, *y* and *z* must be encoded by transition relations

$R_1 \equiv \{(s_j, s_k) \mid x(s_j) \neq x(s_k), y(s_j) \neq y(s_k), z(s_j) \neq z(s_k)\}$

2) Derivi[illegible] the LTL properties to be verified. The invariant "always $T<4$" maps to formula: $[T \leq 4]$

The LTL properties derived so far were verified using the SPIN model checker. To improve the efficiency of the SPIN, the advantages of the SPIN's implementation of the state compression algorithm and of the partial-order reduction algorithm were used. The verification of the LTL properties was carried out for three system models characterized by different datacenter performance parameters:

a) Link Speed = 100Mbps
b) Link Speed = 1 Gbps
c) Link Speed = 10 Gbps

When the verification of the LTL properties was performed for scenario a), SPIN detected several possible constraint violations ($T>4$ sec). For each violation case, the corresponding counterexample traces were generated by SPIN. For scenarios b) and c) SPIN found no invalid states, thus confirming that the policy set is conflict free in these scenarios.

This case study demonstrates that a set of autonomic computing policies that is valid for one scenario can exhibit conflicts when the autonomic system operates in a different scenario. Model checking provides the unique capability to verify autonomic computing policies exhaustively and for the precise scenario in which the self-managing system operates.

Conclusion and future work

As we have clarified in the introduction section, the cloud computing is a new paradigm, and complexity of its design and security management structure has been revealed. Therefore, the security development process is still an opening case yet. However, the research has been conducted according to reality of the current demands; naturally, the well-designed security policy leads to a stable performance of the whole Autonomic Components Ensembles behavior, as well as the action that should be taken against any security violation.

The research presented in the thesis investigates some main challenges in the modern autonomic cloud computing, i.e. the unpredictability of performance, possible violations of SLA, the complexity of the security design, and formal verification via model checking. Therefore, new structural model of formal technique, developed in the thesis, allows us to detect the policy conflict in AC's environment by using a new probabilistic interpretation of KS. This interpretation is used to calculate the required response time including the downtime in case of migration the AC from one host to another. Hence, the SLA violations will be also accordingly determined

In addition, it is shown in the thesis that the KS can be computed using string compression mechanism, and corresponding methods and classes in jResp are developed. . Furthermore, we extended our standard topology to tackle modern infrastructure's demands, e.g. fault tolerance technique, resource pool approach, and live migration of the VMs among the cluster members. The prototyping of the reported work is given to address some specific points that will assist in our research goal, i.e. knowledge repository that is essential part in any autonomic component, reflects the natural behavior of the deployed VM and its interaction with the rest of the other VMS. Therefore, it is shown that the autonomic component presents a promising approach to achieve high-level of expressiveness and security by adding self-awareness capabilities to new IT infrastructure.

Finally, the future work will be connected to biological approach "Artificial Immune Systems" and its instantiation based on our work standpoint. That is, to simulate the way that how the SCEL and its environment could be rigorously developed and reconstructed for finding the threats in early stages, regardless of the threat initiation "internally or externally," and the corresponding adequate response.

List of Publications

Rodonaia, I., Rodonaia, V., & Mousa, M. (2013). Using of information theory metrics in security Modeling of Autonomic Cloud Computing. Transactions Automated Control Systems, Georgian Technical University No 1(14). 240-247.

Milnikov, A., Mousa, M., Rodonaia, I., & Rodonaia, V. (2013). A technique of formal security modeling in automatic cloud computing environment. Proceeding of the 11 International Conference on applied Electromagnetics, Wireless and Optical Communications (ELECTROSIENCE’ 13)”, June 25-27, Dubrovnik, Croatia. . 69-74.

Mousa, M. (2013). Modeling of cloud computing security using Kolmogorov Complexity,. AICT2013 Conference Organizing Committee, Application of Information and Communication Technologies,, (pp. 200-205). BAKU, AZARBIJAN, 24-28.

Mousa, M. (2013). Optimization of Autonomic Component Ensembles Resources. Journal of Technical Science & Technologies International Black Sea University, Volume 2, 2nd-issue., 41-44.

Mousa, M., Rodonaia, I., & Rodonaia, V. (2013). Security modeling of autonomic -component ensembles. Journal of Technical Scienc & Technologies International Black Sea University, 2(1), 15-20.

Mousa, M., Rodonaia, I., Rodonaia, V., Shonia, O., & Prangishvili, A. (2014). Formal verification in autonomic-component ensembles,. 5th International Conference on Circuits, Systems, Control, and Signals. Salerno, Italy. 57-61.