Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 116 (2005) 45–57

www.elsevier.com/locate/entcs

A Dependable Autonomic Computing Environment for Self-Testing of Complex Heterogeneous Systems

Andrea Baldini, Alfredo Benso, Paolo Prinetto[1] ,[2] ,[3]

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24 I-10129, Torino, Italy

Abstract

This paper is part of a R&D project aiming at the definition and implementation of an environment for dependable autonomic computing. The primary goal of the study is the increase of dependability of digital systems using self-healing techniques. Mobile agents implement self-testing policies for complex and heterogeneous systems. The aim of this paper is to present the general ideas of the project, describe the design decisions and a detailed view of the current architecture. The research includes design and development of a working prototype.

Keywords: Dependability, Autonomic Computing, Self-testing, System Level Testing, Mobile Agents.

1 Introduction

This study addresses the general necessity of increasing the dependability of complex heterogeneous systems, using self-healing techniques. Self-healing is a concept coming from the autonomic computing vision, i.e., self-test, -diagnosis, -repair, and -management of target systems.

In the last years, as digital systems have grown in complexity, their operation has become brittle and unreliable [1]. Not only computing systems'

[1] Email: andrea.baldini@polito.it
[2] Email: alfredo.benso@polito.it
[3] Email: paolo.prinetto@polito.it

doi:10.1016/j.entcs.2004.02.087

complexity appears to be approaching the limits of human capability, but also designers are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. [2]

On the other hand, the goal of the companies is to increase productivity while minimizing complexity for users and autonomic computing aspire to realize it. [3]

The main characteristic of autonomic systems is self-management, divided into four aspects: [2] self-configuration, i.e., automatic configuration in accordance with high-level policies; self-optimization, i.e., proactive behavior of the system, which continually seek ways to improve its performance and increase its efficiency; self-healing, i.e., automatic detection, diagnosis, and repairing of localized software and hardware problems; self-protection, i.e., anticipation of problems based on early reports from sensors.

Starting from the idea of dealing with dependability issues during the mission operation of a system, our key point is an extensive use of mobile agents, i.e., autonomous and identifiable computer program that can move from host to host in a network under their own control, to periodically test a set of target systems (self-testing), and in case of failure to diagnose (self-diagnosis) and possibly solve the problem (self-repairing) before a severe malfunctioning occurs (self-healing).

In the field of Computer Science, there are a number of different definitions of software agent. According to Wooldridge and Jennings [4] an agent is a software module with the following properties: autonomy, i.e., state encapsulation and independent decision-making; reactivity, i.e., ability to perceive external environment and respond to changes; pro-activeness, i.e., goal-directed behavior; social ability, i.e., interaction with other agents via a communication language with an agent-independent semantics. [5]

Mobile agents are programs that can migrate from one machine to another.

Mobile agents evolve from the existing distributed computing paradigms with several novelties. [6][7] The main advantages are: autonomy, even more strongly than non-mobile agents; better support for mobile hosts; reduction of network traffic, since an agent can simply work on site; facilitation for software deployment. They can roam around, gather information about the environment, download files.

Non-mobile agents and mobile agents are present in our environment, all of them coherently introduced in the context of a multi-agent autonomic system, i.e., a computing system that can manage itself given high-level objectives from administrators.

In this paper in particular we will focus on software implemented hardware self-testing capabilities.

1.1 Motivations and Advantages

Based on our experience in the hardware and system testing fields, we think that periodic testing of complex heterogeneous systems is the right direction in order to obtain a significant gain in dependability.

The main advantages of using an autonomic computing environment based on mobile agents are fundamentally three: pro-activeness, i.e., continuous self-testing and possibly self-healing of systems occurs actively and before a malfunctioning occurs; updating and upgrading capabilities, i.e., test, diagnosis, and repairing procedures are always up-to-date; policy-based administrative control, i.e., the administrators have complete control of the environment stating the general dependability policies, e.g., testing frequency.

The environment tries to respect the policies without human intervention, hiding the details of the single procedure.

2 Related Research and State-of-the-Art

Our experience has been historically in the field of hardware and system testing. The interest in applying software-implemented testing of systems and components comes from advancements in the use of automatic test equipments (ATE) such as automatic test pattern generation (ATPG), [8] Built-In Self Test (BIST) [9] and self-repair (BISR), [10] and from our experience in system testing design processes. [11]

Dependability concerns [12] are understandably at the center of our attention; since the idea of ”Autonomic Computing” [3] was first introduced by IBM's senior vice president of research, Paul Horn, we have been interested in using this concept to leverage on dependability issues.

Many of the founding elements for IBM's autonomic computing initiative already manifest into today's iSeries servers [13] that exploit IBM's blueprint for delivering technology and tools to ease management of systems.

Another example is eBiquity, University of Maryland: [14] the goal of this project is to create systems based on the cooperation of autonomous, dynamic and adaptive component of various systems. These systems include mobile/pervasive computing, multi-agent systems, artificial intelligence and e-services.

MCA (Machine Check Architecture), Intel [15] is an open architecture that allows systems to continue executing transactions as it recovers from error conditions.

We can also mention RTP4 (Resilient Telco Platform 4) Continuous Services, [16] a software solution that incorporates numerous self-configuring,

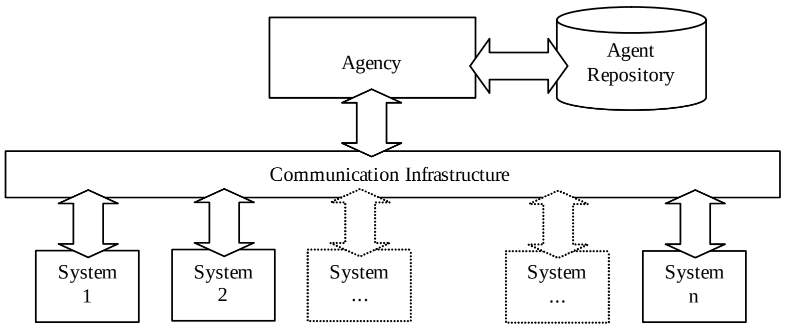

Fig. 1. General architecture

-healing, -optimizing and -protecting abilities, without having to consider the current system configuration.

Another example of autonomous system architecture is described in the Rainbow project (part of Carnegie Mellon ABLE initiative), a new technology supporting automated, dynamic system adaptation via architectural models, explicit representation of user tasks, and performance-oriented run-time gauges. [17]

One last citation is for the Recovery-Oriented Computing (ROC) project, a joint Berkeley-Stanford research project that is investigating novel techniques for building highly-dependable Internet services. [1][18]

3 General Architecture and Design Decisions

The project considers a network of systems (hereinafter called target systems) composing, together with a system acting as provider of the self-testing capabilities, i.e., the Agency, the whole environment (see Figure 1).

The periodic testing of critical component is provided by specific mobile agents. The tests we need to perform are software implemented hardware tests, such as testing the RAM of the target system or the hard disk before a failure occurs. As a matter of fact, before failing completely usually components show abnormal behavior, and testing can reveal it, in a similar way as the initial bootstrap memory testing of a PC.

Typical tests involve processor's functional units, memory (RAM or mass storage), communication interfaces (network adapters, modems), etc.

The idea of using mobile agents for implementing the self-testing capabilities, implies the instantiation of a fleet of mobile agents responsible for specific test actions, and management.

3.1 Policies and Goals

An autonomic computing system requires a uniform method for defining the policies that govern the decision-making for autonomic managers. A policy specifies the criteria that an autonomic manager uses to accomplish a definite goal or course of action.

The policy approach allows administrators to provide the general dependability directives, while the specific problem resolution and the management of the system behavior are left to the system. Our environment realizes it with policies and goals.

Policies

Each *policy* refers to a single component, e.g., video RAM, bus controller, specific piece of software, or a group of them. It expresses a general testing profile for such component or group of components. It specifies which kind of test to perform on the component, e.g., it specifies the quality of a test. It specifies also the frequency of a test, e.g., each day, each week. Moreover, it specifies what to do in case of failure, e.g., prompt a message or start diagnosis phase.

An example of policy is: "Test CPU ALU of all systems using functional test of high quality each week".

Goals

Goals are translations of policies into simple actions, and are the basis of the communications among agents.

Each Goal represents an atomic task to be performed at a time on a particular target system, so the environment can more easily plan several tasks, i.e. activate several goals, for each target system. Each goal contains:

- System Identification: unique target system id code;
- Component Identification: unique id number for each component of a system, or a single set of components, e.g., an entire board;
- Device information: hardware information collected about devices involved in the policy; hardware information are used to monitor the hardware configuration of the whole system and to plan testing;
- On-action-failure behavior: the system allows administrators to decide the consequence of an action failure;
- Frequency: test frequency to be respected;
- Inactivity time: inactivity time on the target system that can trigger an event, e.g. a test on some component when the system has been idle for

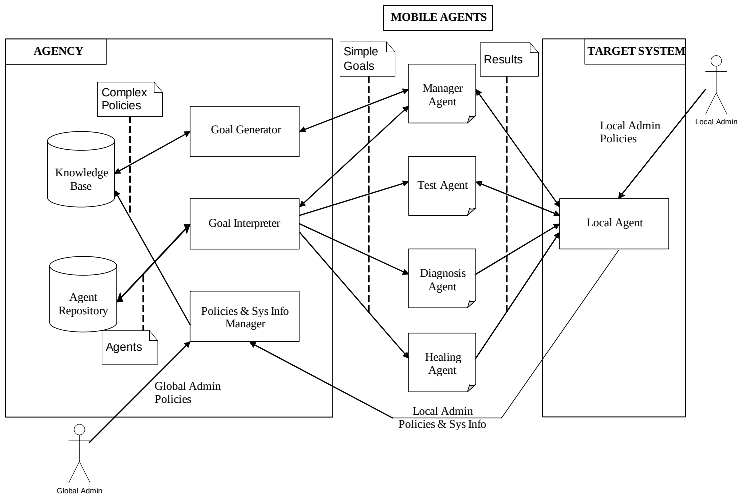

Fig. 2. Detailed architectural view

more than n minutes;

- Quality: quality of the test to perform on the system component, e.g., fault coverage;
- Constraints: constraints related to time of activation of the goal, bandwidth occupation and resource occupation.

4 Detailed Architecture

The self-healing environment is based on a common JAVA platform populated with autonomous agents. As a general framework we chose JADE (Java Agent DEvelopment Framework) [19], an agent middleware that implements an agent infrastructure compliant with the FIPA specifications [20]. The entire environment is built on top of that (see Figure 2).

The main components of this architecture are the Agency, the Target Systems, and a fleet of Mobile Agents, namely Manager, Test, Diagnosis and Healing Agents. In this scheme, the Agent Repository is included in the Agency, differently from Figure 1.

Our agents are divided into static agents and mobile agents. The former manage to retrieve information on the systems connected to the environment, along with the administrators policies (analysis), then plan the tests (planning); the latter move from system to system and start the tests (testing), retrieve the results, choose the suitable repair (diagnosis) and manage the entire testing and healing process in general (healing). In this paper we will cite only testing agents.

Mobile agents communicate through goals. Each agent maintains a list of its goals and processes them one after the other. When an agent needs a service from the outside, it sends a request, in form of a goal, to the agent able to provide that service.

While mobile agents have no stable place, static agents reside on the central Agency. Moreover, there is a static agent on each target system, the Local Agent.

4.1 Description of Agents

The *Agency* is the core of the environment. The Agency receives the information about each target system and generates a healing plan elaborating the settings coming from the Local Agents and those established by the Global Administrator. It also creates (activates) new mobile agents and send simple commands (goals) to them.

Hence the Agency is composed by five different modules:

(i) Policy and Sys Info Manager: a module responsible for the reception and elaboration of the information about each target system and for policy management;

(ii) Goal Generator: a module responsible for the translation of the information and settings in policies to be passed to the mobile Manager Agent;

(iii) Knowledge Base, a data base which stores all the target systems information;

(iv) Goal Interpreter: a module responsible for the activation of the mobile agents triggered by the request of the mobile Manager Agent;

(v) Agent Repository, containing a set of mobile agents that can be activated by the Agency in response to a Manager Agent request.

Mobile Agents are the moving executors of the environment.

- The Manager Agent is a mobile agent responsible for the management and the timely execution of the test plan generated by the Agency. The Manager Agent is in charge of a set of target systems and it moves from one to another to check the status of the testing process, retrieve test history and collect results of the tasks performed by mobile agents. It can self-clone if in charge of too many target system.
- Test (or Diagnosis or Healing) Agents are mobile agents with specific ca-

pabilities needed by the environment and activated by the Agency when necessary. An agent moves towards a target system, analyzes available resources and possibly performs its task. It can receive other tasks while it is moving or acting on a specific target system, possibly self-cloning. When it has terminated its tasks, it dies.

The *Local Agent* is in charge of the communications with the Java Virtual Machine and the Operating System of a target system, but also of the communication with the Manager Agent and mobile agents. It maintains updated information about all actions performed on the system; it maintains an updated list of available resources; it is also responsible for the management of the local policies.

4.2 Communications and Simple Behavior

We have described the roles of the agents, now we can complete the picture adding the information about communications (see Figure 2).

To understand the details of the communications, it is useful to describe a basic scenario (see Figure 3).

Each local agent alerts the environment when its system turns on or it is available; then it retrieves all useful information on the system and the local administrator policies. Such information is transmitted to the agency (and updated).

Agency agents match the local administrator policies with the global policies (Policy Manager) and, along with the software and hardware information, generate a list of tests to perform (Goal Generator). The result is a list of goals to communicate to a mobile manager agent. The mobile manager agent in charge of the addressed system moves to the target and if it is the right time (determined by policies/goals constraints) it starts the actions, i.e., it sends to the agency a list of goals each representing a single action to perform, respecting policies/goals constraints.

The agency (Goal Interpreter) selects an agent able to perform the specific action: if the agent exists in the environment, i.e., it has been already activated, then the agency assigns the action (goal) to the existing agent, else the agency creates a new agent with information coming from the agent repository and activates it with the new action (goal).

The involved mobile agent migrates, moves to the target system and, if the resources are available, executes its task (goal); results are stored in the target system (local agent). The manager agent on duty is responsible for results acquisition and analysis. Depending on results, it sends new tasks (goals) to the agency, e.g., a diagnosis action after a failure of a specific test,

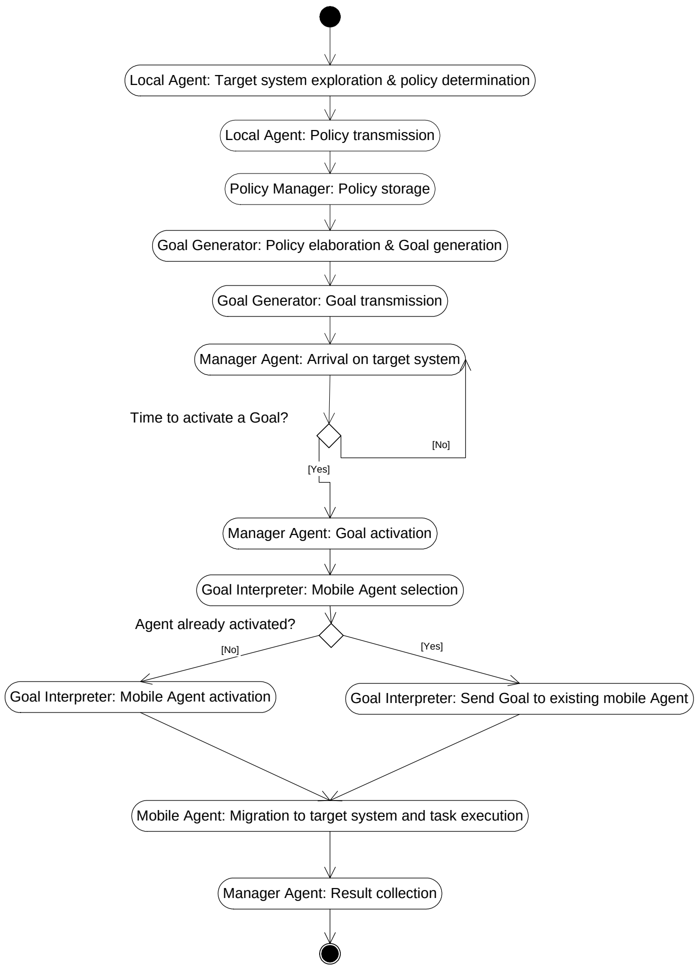
Local Agent: Target system exploration & policy determination
Local Agent: Policy transmission
Policy Manager: Policy storage
Goal Generator: Policy elaboration & Goal generation
Goal Generator: Goal transmission
Manager Agent: Arrival on target system
Time to activate a Goal?
[No]
[Yes]
Manager Agent: Goal activation
Goal Interpreter: Mobile Agent selection
Agent already activated?
[No]
[Yes]
Goal Interpreter: Mobile Agent activation
Goal Interpreter: Send Goal to existing mobile Agent
Mobile Agent: Migration to target system and task execution
Manager Agent: Result collection

Fig. 3. Main flow of execution in (basic scenario)

and it stores the result (Knowledge Base).

4.3 Payload Mechanism and Testing

The entire environment is based on the idea of using software-implemented tests, i.e., programs that are able to test the functionality of the devices and to detect misbehavior.

Moreover we try to implement tests that can run online, i.e., concurrently with the mission operation of the SUT (System Under Test), or at least without or with little user awareness, e.g., deploying idle moments.

This constraint has an impact also on software-implemented tests, in terms of access to the device during the test, i.e., we should retain a resource only for a short time, and of device state, i.e., we cannot expect to find a resource in the same state after releasing and reacquiring it.

Finally, a hardware test usually needs a native library to be loaded, specific for the target system, usually written in C or assembly code and compiled for the specific platform. These pieces of code are interpreted as additional test agents in our environment, implying that such agents contain native libraries as payload.

Given these problems, one of the main purposes of the Local Agent on a target system is to make the local tests possible.

Local tests are triggered by goals in the Test agents. As soon as a new goal is processed by the Test agent it belongs to, this agent moves to the involved SUT (System Under Test). Here it finds the Local agent and its services.

The Test agent asks for minimum resources, depending on the goal constraints.

We have already stated that a hardware test usually needs a native library to be loaded. In this case, if an up-to-date, or at least compatible, library is already locally available, it is just loaded on the local Operating system. If it is not available or too old, the Local agent requests it to the Goal Interpreter as a special Payload test agent.

The Local agent stores the library (payload) locally and finally can send a message to the Test agent with the authorization to the test. Then the Test agent performs the test, possibly after the loading of the required native library. The local agent provides all the necessary services for management, naming and versioning of the libraries, along with the Agency; moreover it provides access to low level functions and system calls though a specific Java Native Interface (JNI).

5 Working Prototype

The first thing to choose is the class of target systems, and the decision of using PCs is by far the most viable. Nowadays they are the most common digital systems of a certain complexity and they can be easily connected to other digital systems through networks.

For sake of completeness, both Windows-based systems and Linux boxes are addressed. The Agency is activated on Windows 2000 Server or Windows XP Pro and the Local Agent runs under Linux Mandrake 9.2.

Additional simplifications are included in the prototype; in particular both Knowledge Base and Agent Repository are implemented in a preliminary version.

Component	Type & Model of component	Type of Test	Lib
CPU	Mobile AMD Athlon XP-M 2000+	Functional ALU	Yes
RAM	DDR SDRAM 1 SODIMM 512MB	Partial March Test	Yes
Modem	SiS 56k Winmodem HAMR5600	AT Query	Yes
LAN	SiS 900 PCI Fast Ethernet NIC	TCPLoss	

Table 1
Test cases

At last the development of Mobile Agents and native libraries is under way: we have a limited set of working agents. We focus on functional testing of specific components, e.g., processor, RAM, modem, network adapter.

5.1 Practical Results

We have installed the environment on a very small network, involving one central system (Agency) and two target systems.

The environment is easily installed and configured on the machines, both Windows- and Linux-based.

From the testing perspective, four different test cases have been analyzed, functionally testing the arithmetic unit of the processor, a part of the memory, the internal modem and the network adapter (see Table 1). We omit the details of the specific tests performed locally. The last column represents the presence of a native library to load in order to perform the test. Test policies are simple and use periodic testing of the given components on the two target systems.

All modules correctly manage the communications, and through specific tools provided by JADE we have demonstrated correct agent mobility.

It is quite difficult to show the experiments, we try to give an idea of the exchange of messages on the following ”sniffer” screenshot (see Figure 4).

In the figure the GUI of the middleware shows two panes: on the left a list of the currently active agents (MainContainer represents the Agency, Containers represent target systems); on the right an example of message exchange among agents at a particular instant. At a certain point of the sequence of messages the Manager Agent moves from the Main Container (Agency's place) to Container3 (Target System).

A last note on the dimensions of the environment at this point, supposing a Java Virtual Machine already installed on the system: the Agency side, is

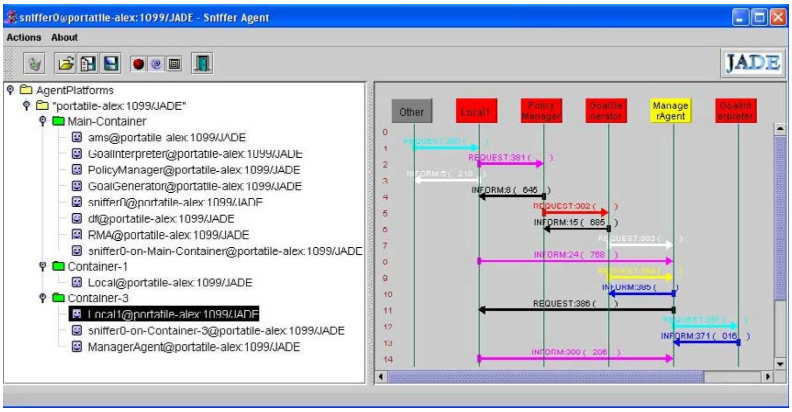
sniffer0@portatile-alex:1099/JADE - Sniffer Agent
Actions About
JADE
AgentPlatforms
"portatile-alex:1099/JADE"
Main-Container
ams@portatile-alex:1099/JADE
GoalInterpreter@portatile-alex:1099/JADE
PolicyManager@portatile-alex:1099/JADE
GoalGenerator@portatile-alex:1099/JADE
sniffer0@portatile-alex:1099/JADE
df@portatile-alex:1099/JADE
RMA@portatile-alex:1099/JADE
sniffer0-on-Main-Container@portatile-alex:1099/JADE
Container-1
Local@portatile-alex:1099/JADE
Container-3
Local1@portatile-alex:1099/JADE
sniffer0-on-Container-3@portatile-alex:1099/JADE
ManagerAgent@portatile-alex:1099/JADE
Other
Local1
PolicyManager
GoalGenerator
ManagerAgent
GoalInterpreter
REQUEST:380
REQUEST:381
INFORM:5 (218)
INFORM:8 (645)
REQUEST:302
INFORM:15 (685)
REQUEST:383
INFORM:24 (768)
REQUEST:384
INFORM:385
REQUEST:386
REQUEST:387
INFORM:371 (016)
INFORM:300 (206)

Fig. 4. Screenshot of JADE Sniffer

about 1.4 MB; the target system side, including JADE core, is about 700 KB; mobile agents, including payloads, vary from 15 KB to 30 KB.

6 Conclusions and Future Work

In the context of a R&D project aiming at the definition and implementation of an environment for dependable autonomic computing, we have presented the general ideas of the project, described the design decisions and a detailed view of the current architecture.

The work is beginning to produce results in terms of feasibility and applicability, but many parts must be addressed by research studies and fully implemented.

The prototype is limited and the application at case studies has been restricted too. We need much work to complete the case study, developing new agents (testing, diagnosis and self-repairing agents) and applying the environment to a significant number of target systems.

References

[1] A. Fox, D. Patterson; "Self-Repairing Computers", Scientific American, June 2003

[2] J.O. Kephart, D.M. Chess, "The Vision of Autonomic Computing", IEEE Computer, Jan. 2003, pp. 41-50

[3] P. Horn, "Autonomic Computing: IBM's Perspective on the State of Information Technology," IBM Corporation, October 15, 2001

[4] M. Wooldridge, N.R. Jennings, "Intelligent agents: Theory and practice." The Knowledge Engineering Review, 10(2):115-152, 1995

[5] M.R. Genesereth, S.P. Ketchpel, "Software agents." Communications of the Association for Computing Machinery, pp 48-53, July 1994

[6] A. Fuggetta, G.P. Picco, G. Vigna, "Understanding Code Mobility." IEEE Transactions on Software Engineering, Vol. 24, 1998

[7] C.G. Harrison, D.M. Chess, A. Kershenbaumn, "Mobile Agents: Are they a good idea?" Available as IBM Research Report, 1995

[8] F. Corno, U. Glaser, P. Prinetto, M. Sonza Reorda, H. Vierhaus, M. Violante; "SymFony: a Hybrid Topological-Symbolic ATPG Exploiting RT-Level Information"; Computer-Aided Design, Volume: 18, Issue: 2, February 1999, page(s) 191-201

[9] A. Benso, S. Chiusano, G. Di Natale, M. Lobetti Bodoni, P. Prinetto; "On-line & Off-line BIST in IP-Core Design"; IEEE Design and Test of Computers, Volume: 18, Issue: 5, September-October 2001, page(s) 92-99

[10] A. Benso, S. Chiusano, G. Di Natale, P. Prinetto; "An On-line BISTed RAM Architecture with Self Repair Capabilities"; IEEE Transactions on Reliability, Volume: 51, Issue: 1, March 2002, page(s) 123-128

[11] A. Baldini, A. Benso, P. Prinetto; "Efficient design of system test: a layered architecture"; ITC 2002: IEEE International Test Conference, Baltimore, MD (USA), October 2002, page(s) 930-939

[12] Benso, S. Chiusano, P. Prinetto; "A Software Development Kit for Dependable Applications in Embedded Systems"; ITC 2000: IEEE International Test Conference, Atlantic City (NJ), October 2000, page(s) 170-178

[13] IBM iSeries servers. iSeries V5R2 and IBM's autonomic computing initiative. Self-Managing IT Infrastructure Technologies for e-business, IBM, 2002

[14] EBiquity, http://www.ebiquity.org, University of Maryland

[15] "IA - 32 Intel Architecture Software Developer's Manual, Volume 3 - System Programming Guide", by Intel Corporation, Chapter 14, 2003

[16] Resilient Telco Platform. FSC-White Paper, http://www.fujitsu-siemens.com/rl/products/software/rtp4.html

[17] Rainbow project, http://www-2.cs.cmu.edu/ able/rainbow/

[18] ROC Project, http://roc.cs.berkeley.edu/

[19] JADE Framework, http://sharon.cselt.it/projects/jade/

[20] Foundation for Intelligent Physical Agents, http://www.fipa.org/