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

Andrea Baldini, Alfredo Benso, Paolo Prinetto

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24
I-10129, Torino, Italy
{baldini, benso, prinetto}@polito.it

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, namely self-testing, self-diagnosis and self-repairing. 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, whose implementation is presented as significant case study and demonstrator.

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, and in this context it means 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]. Digital systems become more interconnected and diverse, and not only computing systems' 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 of autonomic systems in accordance with high-level policies; self-optimization, i.e., proactive behavior of each component and the system itself, 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 this idea of dealing with dependability issues during the mission operation of a system, our key point is the basic decision of extensively using mobile
agents, i.e., autonomous and identifiable computer program that can move from host to host in a heterogeneous 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] by an agent is a software module that enjoys the following properties: autonomy, i.e., state encapsulation and independent decision-making capabilities; reactivity, i.e., ability to perceive external environment and respond to changes that occur in it; pro-activeness, i.e., goal-directed behavior exhibited by taking the initiative; 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 offered by the mobile agent paradigm are: autonomy, even more strongly than non-mobile agents; better support for mobile hosts; reduction of network traffic, since flows of messages can be reduced sending only the agent code and related state, so that an agent can simply work on site; facilitation for software deployment, i.e., software installation and updating process. 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/upgrading capabilities, i.e., test, diagnosis, and repairing procedures are always up-to-date in a natural way; policy-based administrative control, i.e., the administrators have complete control of the environment stating the general but potentially very precise dependability policies, such as testing frequency for a particular component of a system. 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
A Dependable Autonomic Computing Environment for Self-Testing of Complex Heterogeneous Systems.

from advancements in the use of automatic test equipments (ATE) such as automatic test pattern generation (ATPG) [8], from the idea of using hardware Built-In Self Test (BIST) [9] and more recently 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, to the National Academy of Engineers at Harvard University in a March 2001 keynote address, we have been interested in using the concept of autonomic computing to leverage on dependability issues.

Many examples are available in the context of self-healing in particular.

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.

In the context of autonomic computing, we can also mention RTP4 (Resilient Telco Platform 4) Continuous Services, [16] a software solution that addresses users who need to achieve a true, permanent availability of their applications. The RTP4 incorporates numerous self-configuring, -healing, -optimizing and -protecting abilities, but the main issue is that an application based on RTP4 Continuous Services gains these abilities without having to consider the current system configuration.

Another example of autonomous system architecture is described in the Rainbow project (part of Carnegie MellonABLE 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 Fig. 1. General architecture).

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 the functional units of the processors, the memory (RAM or mass storage), and the communication interfaces (network adapters, modems).
The idea of using mobile agents for implementing the self-testing capabilities, idea at the basis of the project, implies the instantiation of a fleet of mobile agents responsible for specific test actions, and management. The agents use the Communication Infrastructure to reach each single target system.

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.

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, hiding the details, 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”. The policy is specified by an administrator of the environment through a Graphical User Interface or using a fixed text notation.

Goals are a translation 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 identification code of the target system;
A Dependable Autonomic Computing Environment for Self-Testing of Complex Heterogeneous Systems.

- Component Identification: unique identification 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 by administrators to monitor the hardware configuration of the whole system and by the environment to plan testing;
- On-action-failure behavior: the system allows administrators to decide the consequence of an action failure, as specified in the related policy;
- Frequency: test frequency to be respected by the environment;
- 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 more than n minutes;
- Quality: quality of the test or of the operations to perform on the system component, e.g., for a test action quality is the fault coverage of a test procedure.
- Constraints: constraints related to time of activation of the goal, bandwidth occupation and resource occupation.

3.2 Java-based Infrastructure and JADE Framework

An important choice is the use of a Java-based infrastructure.

Java is an ideal platform for agents and mobile agents in particular: it contains Java object-oriented language; standard virtual machines allow code portability; support for multithreading is included. Moreover access to low-level functions and system calls is guaranteed by Java Native Interface (JNI).

As a consequence of Java choice, each target system needs to run a Java Virtual Machine (JVM); on top of a Java Virtual Machine, the agents need a framework in which they can live and work. Many agent platforms exist, and they provide the same basic services to their fleet of agents: they create the agents; they take care of moving the agents; they restart the agents once they moved to another system; they allow the agents to exchange messages; they kill the agents.

In our case we chose JADE (Java Agent DEvelopment Framework) [19], an agent middleware that implements an agent infrastructure. The implementation of multi-agent systems is supported through a middleware, which complies with the FIPA specifications [20], and through a set of tools that supports the debugging and deployment phase.

4 Detailed Architecture

The self-healing environment is based on a common JAVA platform populated with autonomous agents (see Fig. 2. Detailed architectural view).

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 Fig. 1. General architecture.
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

Referring again to the previous figure (see Fig. 2. Detailed architectural view), we can describe shortly the responsibilities of each agent.

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

1. Policy and Sys Info Manager: a module responsible for the reception and elaboration of the information about each target system and for policy management;  
2. Goal Generator: a module responsible for the translation of the information and settings in policies to be passed to the mobile Manager Agent;  
3. Knowledge Base, a database which stores all the target systems information;  
4. Goal Interpreter: a module responsible for the activation of the mobile agents triggered by the request of the mobile Manager Agent;  
5. 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 capabilities 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 Fig. 2. Detailed architectural view).  

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

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 activates a series of goals: 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, and it stores the result (Knowledge Base).

Fig. 3. Main flow of execution (basic scenario)
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 misbehaviors, accessing the hardware registers.

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. If we perform online tests we must consider that some other programs could request, or worse need, an access to the device during the test. That means that even if we can obtain an exclusive access to a device, we should retain it only for a short time. Moreover, the tests must be designed in such a manner not to rely on the device state; if we release a resource and then reacquire it, we cannot expect to find it in the same state.

Another big problem with online test is the operating system itself. It is always present and it needs some resources; or better it reserves some for itself. Since it grants us access to devices, it will never allow us to take and test the ones it uses.

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. If the Local agent returns positive answers, the Test agent asks for a final acknowledgement and communicates the name of the test it is going to perform.

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. The Local Agent prototype integrates the module for 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, providing the basic services of storing only simplified information on systems and few agents.

At last the development of Mobile Agents and native libraries is under way: we have a limited set of working agents, the majority involving testing of target systems. 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. It is obvious that we cannot claim to have verified the complete functioning of the entire environment, but we have achieved some significant practical result.

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

Target systems have been analyzed and tested through the local agent using specific native libraries and ad-hoc tests.

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. Test cases). 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.

Table 1. Test cases

<table>
<thead>
<tr>
<th>Component</th>
<th>Type / Model of component</th>
<th>Type of Test</th>
<th>Lib</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Mobile AMD Athlon XP-M 2000+</td>
<td>Functional ALU</td>
<td>✓</td>
</tr>
<tr>
<td>RAM</td>
<td>DDR SDRAM 1 SODIMM 512MB</td>
<td>Partial March Test</td>
<td>✓</td>
</tr>
<tr>
<td>Modem</td>
<td>[SiS] 56k Winmodem HAMR5600</td>
<td>AT Query</td>
<td>✓</td>
</tr>
<tr>
<td>LAN</td>
<td>SiS 900 PCI Fast Ethernet Adapter</td>
<td>TCPLoss</td>
<td></td>
</tr>
</tbody>
</table>

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 screenshot taken from the JADE tool sniffer during a running session of the environment (see Fig. 4. Screenshot of JADE Sniffer).
Fig. 4. Screenshot of JADE Sniffer

In the figure the GUI of JADE main container (middleware) shows two panes: on the left there is the list of the currently active agents, we can distinguish the main components of our environment (MainContainer represents the running place of the Agency, other Containers represent target systems); on the right there is 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), even if this is not directly visible in the sniffer pane.

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, including JADE-S, is 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 in the current implementation and for current experiments. Mobile agents’ dimensions can vary a lot in case of complex payloads.

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. It is obvious that 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, in order to evaluate the actual validity of the self-healing capabilities with respect to known dependability measures.

References

13. IBM iSeries servers. iSeries V5R2 and IBM’s autonomic computing initiative. Self-Managing IT Infrastructure Technologies for e-business, IBM Corporation, 2002
14. EBiquity, www.ebiquity.org, The EBiquity Research Group at the University of Maryland