Increasing Systems' Availability through Agents and Reconfigurable Systems

Sz. Enyedi, L. Miclea, I. Ştefan
Technical University of Cluj-Napoca
{Szilard.Enyedi/Liviu.Miclea/Iulia.Stefan}@aut.utcluj.ro

Abstract

In this article, we propose a distributed use of software-based self-testing, where intelligent agents are responsible for the transfer of software routines to the distributed processors, which in turn will be able to execute the routines and test/repair the corresponding subsystem. This distributed strategy is flexible, reusable and re-programmable.

1. Introduction

Plain BIST and BISR are not well suited for the testing, diagnosis and repair of heterogeneous, distributed and geographically scattered systems, such as nationwide telecommunications or energy distribution systems. Such a system is presented in figure 1.

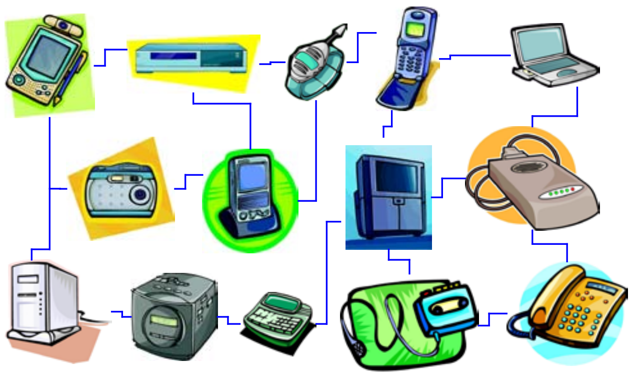

Figure 1. Heterogeneous distributed system with many subsystems.

Decentralization of test and repair greatly reduces the communicational overhead and increases the flexibility and reliability of the testing system itself. The multiagent approach is only natural to such a problem, as multiagent societies are naturally heterogeneous, decentralized and distributed.

An *agent* is, as implemented here, a piece of software capable of independent existence within an environment provided for it, which is able to communicate with entities similar to it, to unaidedly accomplish the work assigned to it and also to travel between geographically separated locations in its environment. Figure 2 presents the main characteristics of software agents.

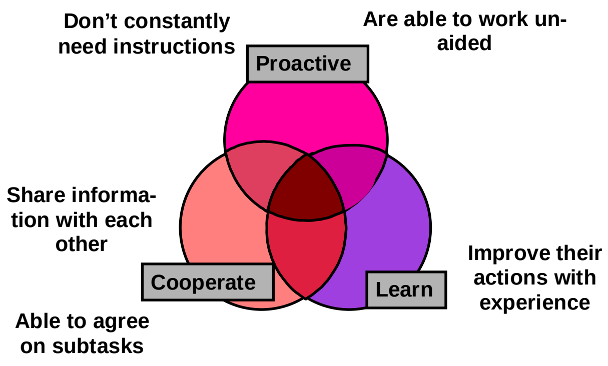
Don't constantly need instructions
Proactive
Are able to work unaided
Share information with each other
Improve their actions with experience
Cooperate
Learn
Able to agree on subtasks

Figure 2. The main agent characteristics.

The agents' communication capabilities and mobility lead to the concept of *multiagent society*, which is here a distributed collection of interacting, mobile agents, residing in different parts of the multiagent environment. We shall call a multiagent society whose main actor is the tester agent a *testing society*. Most DBIST approaches [1]-[5] use a central control authority to start/stop the remote BIST tests, to generally organize the DBIST process and gather together the results. There are also distributed, decentralized testing techniques, some involving agents [6], [7].

We present an agent society whose agents test the components (processors, memories, etc) of subsystems in a distributed system. The agents are used for the transfer of embedded software portions to the subsystems for the effective execution of BIST sessions. Agents enable the BIST functions of these subsystems, therefore the distributed BIST nature of the solution. The agents may also repair the subsystem, for example if there is a backup processor installed.

2. Agent-based DBIST and DBISR of processors and their peripherals

2.1. Generalities

The IEEE 1232 family of standards, analyzed in [8], describe common exchange formats and software services for reasoning systems used in system test and diagnosis. The goal is to make the data exchange between two different diagnostic reasoners easy. The standard also defines software interfaces, for the use of external tools that can access the diagnostic data in a consistent manner. It allows exchanging diagnostic information and embedding diagnostic reasoners in any test environment.

Intelligent agents are software modules able to make decisions on their own, communicate with each other, learn new things and even “travel” from system to system (see also [9]).

Most of the large systems we talk about are heterogeneous, comprising a large number of devices of different types. All these devices have different hardware and/or software, tasks, dependability requirements, but all are capable of running software (in order to be able to run the agent code).

A multi-agent approach and diagnosis ontology for diagnosis of spatially distributed technical systems is presented in [10]; however, in that approach, each subsystem has its own agent monitoring and diagnosing it, which can be costly in some cases. The memory holding the agent could be used for system purposes.

In this paper, we propose an innovative solution based on multi-agent approach for testing and diagnosing distributed systems. It offers many advantages like flexibility, easy maintenance, diagnosis tool for parts of the overall system, and fault tolerance due to the Built-in Self-Repair. Some modern complex devices have also *BIST-ed components*, so we can decompose the diagnosis of the whole system to the diagnosis of components. Our approach differs from other multi-agent approaches, because the agents are portable, highly platform-independent, they can deal with many types of devices and the system administrator can use various, inexpensive and friendly tools to supervise the devices, tests, agents and the agent society in general.

Programmable processors are widely used in complex systems to perform critical system functions. In many cases, the system has a distributed nature where several processors are used in different locations of the system. It is well-known that apart from the functional usage of processors they can be a very powerful means of performing other non-functional operations in the system, such as testing, diagnosis, repair, etc.

Recently, a new self-testing strategy known as software-based self-testing (SBST) emerged [11]-[17]. According to SBST an (embedded) processor is used to execute software routines previously transferred to its memory and performs testing of itself and the surrounding components in a complex system or System-on-Chip (SoC). This new self-testing paradigm offers significant flexibility over hardware-based self-testing techniques that do not allow re-programmability and revisions. In software-based self-testing, new self-test routines can be uploaded at any time, new fault models can be targeted and new components can be tested.

2.2. Agent society

The agent society is able to share resources and repair the faults whenever possible. One or more agents diagnose each subsystem.

The agents travel from device to device, try to detect and repair errors, either by themselves or with the help of other agents or a central database. They can also gather “experience” through their work.

A proposed structure of the testing and repairing agent society is presented in figure 3.

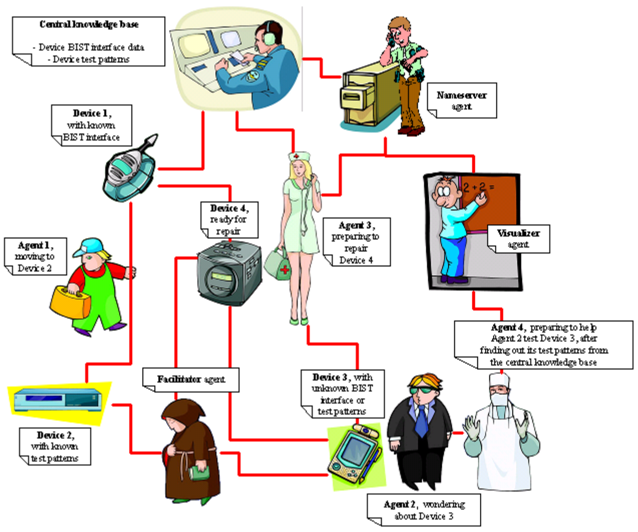
Central knowledge base
- Device BIST interface data
- Device test patterns
Nameserver agent
Device 1, with known BIST interface
Device 4, ready for repair
Agent 3, preparing to repair Device 4
Visualizer agent
Agent 1, moving to Device 2
Agent 4, preparing to help Agent 2 test Device 3, after finding out its test patterns from the central knowledge base
Facilitator agent
Device 3, with unknown BIST interface or test patterns
Device 2, with known test patterns
Agent 2, wondering about Device 3

Figure 3. Agents in action.

When an agent cannot detect a cause of an observed fault or cannot repair it, it appeals to other agents to start cooperation. Due to the diversity of devices in modern complex systems, heterogeneous agents can be implemented that take care of device(s) in their responsibility area.

Different agents have different repair capabilities and they have to ask their colleagues if they cannot repair the fault by themselves.

When it has to test a subsystem, an agent moves in, or “downloads” to the subsystem, into the memory. Then, the agent code is executed by the processor. The agent tests the processor, memory and other peripherals, using test patterns or test code downloaded and run by the processor, like shown in figure 4.

The analysis of a subsystem comprises three major steps:

- detection
- diagnosis
- repair

For each step, the agent has to:

- make a plan
- get the necessary information to execute the plan
- execute the plan
- analyze the results (not compulsory)
- decide (not compulsory)

The first step is to see if there is a fault or not. This may or may not be possible, depending on the agent’s capability in finding a way to check that specific device.

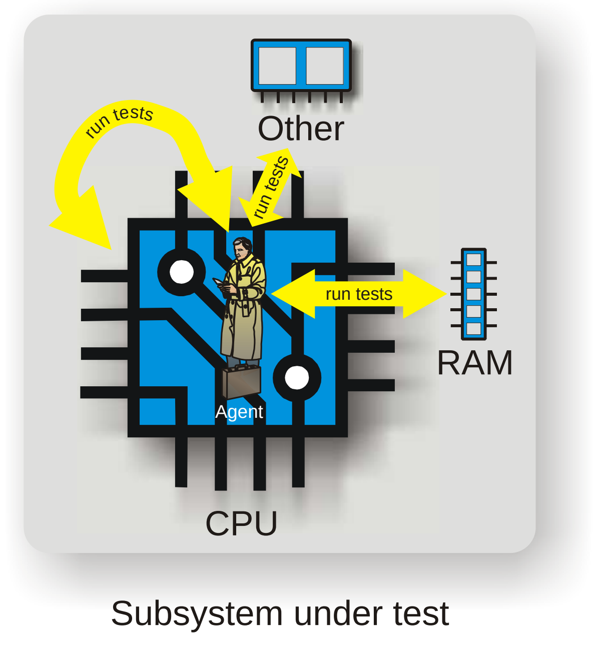

Figure 4. The agent code is executed by the processor, and it runs the tests.

When the fault has been correctly diagnosed, the agent tries to repair it. Of course, being software by nature, the agent is limited mainly to software repairs.

There are four basic types of agents in the society:

- Tester agents
- Nameserver agents
- Facilitator agents
- Visualizer agents

Tester agents are the ones "working", i.e. effectively testing the devices.

Nameservers are like phone books, they make easier for the agents to find each other.

Facilitators are like the Yellow Pages, they know who has what and who knows how to detect or fix what problem.

Visualizers are the interfaces between the agent society and other systems, for example accepting commands from the system administrator and supplying information about tested devices and society status.

More about agent management can be found in [18].

2.3. Experimental agent platforms and resource needs

2.3.1. Java Micro Edition. Sun's Java 2 Micro Edition is standardized, portable, has a small footprint (Sun's KVM reference implementation has about 128 kiloytes), optimized for networking and very flexible.

To ensure portability among different manufacturers' devices, the MIDP 1.0 (Mobile Information Device Profile) and specification establishes some basic functionality for the first generation Java enabled mobile devices. This guarantees that the programs – "midlets" – will run on any MIDP 1.0 certified hardware.

MIDP 1.0 offers only HTTP type connections by default, but there are a few workarounds to have always on, flexible, raw socket connections – proprietary network connections – between the server and the mobile device. MIDP 2.0 is more flexible in this respect, but few mobile devices comply with it.

On need, the j2me agents can be easily extended with additional functions, enabling a device's additional testing abilities.

The drawback of the j2me solution is that from its conception, Java (Enterprise, Standard or Micro) has been designed for portability. This means that it does not allow native access to the hardware, only through the functions of the virtual machine. On the other hand, special, device-specific classes can be developed, which bypass the virtual machine and access the hardware directly.

Another drawback is that the "midlets" – j2me programs – can be installed and run only on the user's request. This is a security measure, aiming at protecting the user's handheld – the original target of j2me – from unwanted programs. However, if there is already a midlet running on the device, with an active network connection, it can send and receive data, including microagents.

2.3.2. BREW. Qualcomm's BREW platform is similar to Java Micro Edition, but the programs can be developed in C++, as well. There is a Micro Java virtual machine for BREW, so that even the j2me programs are able to run. The main advantage of BREW over Java Micro Edition is that it can run native applications that access the hardware. Its main disadvantage is that its use is not widespread, but the number of BREW enabled devices is increasing.

BREW is mainly embedded into CDMA communication devices.

2.3.3. Symbian. Symbian is actually a low scale operating system, supported by Ericsson, Panasonic, Nokia, Psion, Samsung, Siemens and Sony Ericsson. It is mainly for, but not limited to, enhanced mobile phones. It can even run a Java Micro Edition virtual machine, allowing the j2me solution, presented above, to run. Still, the main advantage of Symbian is that it accepts programs that access the underlying hardware directly, circumventing the problems of the aforementioned Java Micro Edition.

Unfortunately, Symbian also requires more resources than the j2me virtual machine, making it more expensive as embedded agent platform.

2.3.4. PalmOS. PalmOS was originally an operating system for Personal Digital Assistants. Later, some PalmOS PDAs became smartphones, and PalmOS got wireless.

The main advantage of PalmOS, like Symbian's, is that its programs can access the hardware directly. The disadvantage is that it was not designed for background applications, but for programs that interact a lot with the user. However, the latest versions (PalmOS 5 and 6) are promising.

2.3.5. Embedded Linux. Linux, the most acclaimed open operating system, also has many downscaled embedded versions. µCLinux, for example, runs on microcontrollers.

Linux, in its embedded versions, is the most powerful and resource efficient platform for embedded computational tasks. The downside is that since the native programs contain native machine instructions, they are not portable to other processors.

2.3.6. Single Board Computers. An SBC is, in fact, a hardware platform. It is a powerful computer, usually with network access, audio and video capabilities, lots of processing power, but all crammed on one small printed circuit board. There are even 45x45mm SBC boards.

Most of them use x86 compatible processors, thus are able to run MS Windows. Nevertheless, the majority uses Linux, for its flexibility.

2.4. Agent communication

At software level, the agents communicate with each other through the FIPA (Foundation for Intelligent Physical Agents) ACL (Agent Communication Language) [18]. For now, our agents have a reduced language set, mainly allowing sharing test sets, device test/repair data and system coverage plans.

The FIPA MTP (Agent Message Transport Protocol) specifications [18] present different ways of communication for the agents to exchange data. IIOP (Internet Inter-ORB Protocol), WAP (Wireless Application Protocol) and HTTP (HyperText Transfer Protocol), TCP/IP over wireline are described, as well as generic wireless solutions. They also deal with bit-oriented, string-oriented and XML-oriented message representations.

3. Conclusions and future work

We presented here a few ideas regarding DBIST and DBISR with intelligent agents. The agents are able to work together in order to find and possibly solve device problems.

The agents travel from device to device, try to detect and repair errors, and learn new solutions. They can “live” on their own, or work together with other agents and/or a database.

When an agent cannot detect a cause of an observed fault or cannot repair it, it appeals to other agents to start cooperation. We use a decentralized diagnosis model, which reduces the complexity and communication overhead of centralized solutions. Due to the diversity of devices in modern complex systems, heterogeneous agents can be implemented that take care of device(s) in their responsibility area.

Different agents have different repair capabilities and they have to ask their colleagues if they cannot repair the fault by themselves.

Tester agents do the testing and repair what is repairable. Visualizers supply the interface between the agent society and the outer world. Nameservers and Facilitators provide lookup services for the agents, so they find each other and also offer their services and knowledge.

The agent management and communication follow FIPA specifications, which describe the management services and communication protocols and formats.

The utilization of intelligent agents for the detection, diagnosis and repair of faults in distributed systems is the focus of the proposed architecture. Significant part of a subsystem can be self-tested using the processing power of the processors used in the various sites of a distributed system. The self-testing is executed using embedded software routines which are able to detect faults in the processors themselves as well as in other subsystem’s parts such as the memory system and input/output system.

The proposed architecture is flexible and reprogrammable. It can be used to perform distributed self-testing in systems with different processor architectures and with different components in each subsystem. The architecture is also scalable and extensible since every time a new component (new memory, new I/O device) is added to a subsystem, a new embedded software module can be transferred by an agent to perform self-testing to the new component.

4. References

[1] L.Miclea, Enyedi Sz., R. Orghidan, *On line BIST Experiments for Distributed Systems*, IEEE European Test Workshop ETW’2001, Stockholm, Sweden, May 29th – June 1st, 2001, pp 37-39

[2] L. Miclea, D. Cimpoca, M. Gordan, *An On-Line BIST Structure for Distributed Control Systems,* Digest of IEEE European Test Workshop ETW’2000, Cascais, Portugal May 23rd – 26th 2000, pp. 283-284

[3] A. Benso, S. Chiusano, S. Di Carlo, *HD2BIST: a Hierarchical Framework for BIST Scheduling, Data patterns delivering and diagnosis in SoCs,* ITC International Test Conference, pp. 899-901, 10 - 2000.

[4] Monica Lobetti Bodoni, A. Benso, S. Chiusano, G. di Natale, P. Prinetto, *An Effective Distributed BIST Architecture for RAMs* , Informal Digest of IEEE European Test Workshop ETW 2000, pp. 201-206

[5] R. Pendurkar, A. Chatterjee, Y. Zorian, *A Distributed BIST Technique for Diagnosis of MCM Interconnections,* International Test Conference 1996 Proceedings, pp. 214-221

[6] L. Miclea, Enyedi Sz., A. Benso, *Intelligent Agents and BIST/BISR - Working Together in Distributed Systems,* Proceedings of International Test Conference, Baltimore, USA, 8th–10th October, 2002, pp. 940-946.

[7] L.Miclea, Enyedi Sz., *Distributed Built-In Self-Test using Intelligent Agents,* IEEE European Test Workshop ETW'2002, Corfu, Greece, May 26th–May 29th, 2002.

[8] J. Sheppard, M. Kaufman, *IEEE 1232 and p1522 standards*, AUTOTESTCON Proceedings, 2000 IEEE, 2000, pp. 388-397

[9] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, 1999

[10] I. A. Letia, F. Craciun, Z Köpe, A Netin, *Distributed diagnosis by BDI agents*, In M H Hamza (ed), IASTED International Conference "Applied Informatics", Innsbruck, Austria, 2000, 862-867, ACTA Press

[11] L. Chen, S.Dey, *Software-Based Self-Testing Methodology for Processor Cores*, IEEE Transactions on CAD of Integrated Circuits and Systems, vo.20, no.3, pp. 369-380, March 2001.

[12] F.Corno, M.Sonza Reorda, G.Squillero, M.Violante, *On the Test of Microprocessor IP Cores*, in Proceedings of the Design Automation & Test in Europe 2001, pp.209-213.

[13] L. Chen, S. Ravi, A.Raghunathan, S. Dey, *A Scalable Software-Based Self-Testing Methodology for Programmable Processors*, in Proceedings of the Design Automation Conference 2003, pp. 548-553.

[14] N.Kranitis, A.Paschalis, D.Gizopoulos, Y.Zorian, *Instruction-Based Self-Testing of Processor Cores*, in Journal of Electronic Testing: Theory and Applications, no 19, pp.103-112, 2003 (Special Issue on 20th IEEE VLSI Test Symposium 2002)

[15] A.Krstic, L.Chen, W.C.Lai, K.T.Cheng, S.Dey, *Embedded Software-Based Self-Test for Programmable Core-Based Designs*, IEEE Design & Test of Computers, July-August 2002, pp. 18-26.

[16] N.Kranitis, G.Xenoulis, A.Paschalis, D.Gizopoulos, Y. Zorian, *Application and Analysis of RT-Level Software-Based Self-Testing for Embedded Processor Cores*, in Proceedings of the IEEE International Test Conference (ITC) 2003, Charlotte, NC, USA, September 30 – October 2, 2003.

[17] N.Kranitis, Y.Xenoulis, D.Gizopoulos, A.Paschalis, Y.Zorian, *Low-Cost Software-Based Self-Testing of RISC Processor Cores*, IEEE Design Automation and Test in Europe Conference (DATE'2003), Munich, Germany, March 2003.

[18] ***, FIPA standards and specifications, http://www.fipa.org