

Agent Based Test and Repair of Distributed Systems

*Original*

Agent Based Test and Repair of Distributed Systems / Miclea, L.; Szilárd, E.; Benso, Alfredo; Prinetto, Paolo Ernesto. -  
In: JOURNAL OF EMBEDDED COMPUTING. - ISSN 1740-4460. - STAMPA. - 1:3(2005), pp. 405-414.

*Availability:*

This version is available at: 11583/1404475 since:

*Publisher:*

IOS Press

*Published*

DOI:

*Terms of use:*

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

*Publisher copyright*

(Article begins on next page)



Politecnico di Torino

# Agent Based Test and Repair of Distributed Systems

Authors: Miclea L., Szilárd E., Benso A., Prinetto P.,

Author's version of the manuscript published in the JOURNAL OF EMBEDDED COMPUTING Vol. 1, No. 3, 2005, pp. 405-414.

**The final PUBLISHED manuscript is available at:**

**URL:** <http://iospress.metapress.com/content/8cuwwa5vh6gy1dw0/fulltext.pdf>

# Agent Based Test and Repair of Distributed Systems

Liviu Miclea, Szilárd Enyedi, Paolo Prinetto, Alfredo Benso, *Members, IEEE*

**Abstract**—This article demonstrates how to use intelligent agents for testing and repairing a distributed system, whose elements may or may not have embedded BIST (Built-In Self-Test) and BISR (Built-In Self-Repair) facilities.

Agents are software modules that perform monitoring, diagnosis and repair of the faults. They form together a society whose members communicate, set goals and solve tasks.

An experimental solution is presented, and future developments of the proposed approach are explored.

**Index Terms**—Intelligent agent, distributed BIST, BISR, self-repair, Java.

## I. INTRODUCTION

### A. Built-In Self-Test

Any system needs to be tested, even the simplest ones, at least once after production but often also during all its mission life. Testing [1,2] however takes a lot of time and hassle. Traditional off-line testing requires the system to be turned off and testers connected to it. The external tester devices and the time lost with preparing and actually testing the components of the system can be expensive and in many cases it may be around the 50% of the overall development cost of the device. If the tests are run during the normal functioning of the system, it is *on-line testing*. Otherwise, it is *off-line testing*. The on-line test can be concurrent, where the test mode is normal mode, and not concurrent, where the idle time is test mode. The on-line test is mandatory when very high functional security and reliability is required, the target faults are transient faults or we need a low latency. In this case no ATE (Automated Test Equipment) is required, but we pay through high hardware overhead.

A solution that fully or partially eliminates external testers involves building the testing capabilities right into the device or system during the design stage. In this way the device is able to test itself without the need of expensive and time con-

suming external test equipments. This solution is usually known as *Built-In Self-Test* [3].

### B. Distributed Built-In Self-Test

As a method for enhancing the availability, stability and security in functioning, built-in self-test has been around for quite a while. However, large systems, with many subsystems – like nationwide telecommunication infrastructures, major computer networks and huge manufacturing plants – need a slightly different approach. Their subsystems may even be scattered over large geographical areas. Another problem is that the subsystems may be of different types, requiring different testing methods. These systems need distributed monitoring, diagnosis and repairing, since it is more expensive to go there and verify the subsystem periodically and, eventually, fix it, than monitoring, diagnosing and repairing it in a distributed manner. Even the communication among different BIST modules of different subsystems and with central management becomes an issue. If the communication is expensive, a decentralized test management can be more efficient.

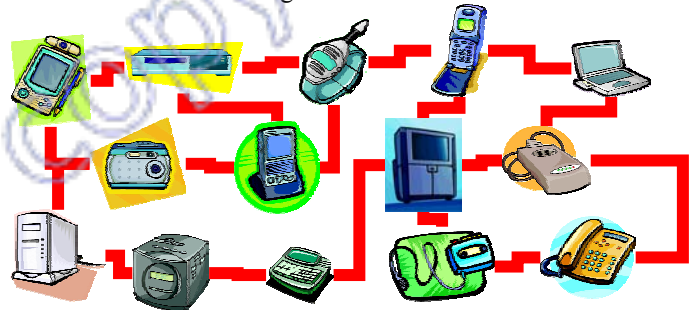


Fig. 1. Heterogeneous network of devices.

The distributed nature of DBIST (Distributed Built-In Self-Test) [4-8] means that each of the modules in the DUT has its own BIST routine, which runs the test more or less independently from the other modules. This way, the actual BIST of the whole device is decomposed into smaller, dedicated BISTs, which should be simpler and easier to develop and maintain. The testing is not done centrally, but locally, in a distributed manner. The system may or may not have a central DBIST management module.

### C. Software Agents

In general, an agent is a software module which is designed to assist an individual user, and to act on that user's behalf. An agent should be able to assist the user in the performance of routine or tedious tasks, to learn the patterns or quirks of a

Manuscript received November 9, 2003.

Liviu Miclea is with the Automation Department, Technical University of Cluj-Napoca, 26-28 Barițiu str., 400027 Cluj-Napoca, Romania (phone/fax: +40-264-594469, e-mail: Liviu.Miclea@aut.utcluj.ro).

Szilárd Enyedi is with the Automation Department, Technical University of Cluj-Napoca, 26-28 Barițiu str., 400027 Cluj-Napoca, Romania (phone/fax: +40-264-594469, e-mail: Szilard.Enyedi@aut.utcluj.ro).

Paolo Prinetto is with the Automation and Informatics Department of Politecnico di Torino, 24 Duca degli Abruzzi, 10129 Torino, Italy (phone: +39- 0115647163/7163, e-mail: Paolo.Prinetto@polito.it).

given situation, to examine and learn from its environment, and to determine the best method in which to carry out its tasks. An agent is delegated to execute a given task, under the constraints it has been given in which to operate.

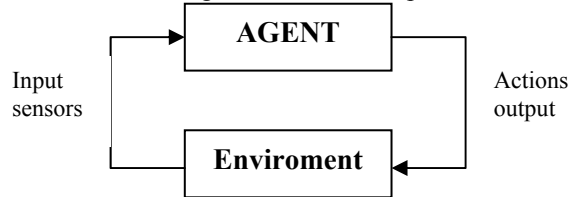


Fig. 2. An agent in its environment.

This automation and learning should be ongoing acts of the agent in order to make it effective. By automating these tasks, an agent frees the user to accomplish other, more productive work.

Agents should be able to resolve ambiguity and make decisions to complete their tasks. They also should be able to learn from other agents the best manner in which to complete a given task. For example, in a test environment a software agent should be able to identify the test requirements of a device, find out how to test the device, and then enable or directly perform the test of the device.

Agents are intended to reduce waste of resources (not reduce *use* of resources, *per se*, but rather the *waste* of resources). A task that can be automated can usually be done more efficiently, especially for frequently-executed, similar tasks. Not only can an agent help avoid the waste of time, but also other resources such as bandwidth, because it can make decisions by itself, without connecting to a central server over and over again. It needs to connect to the server only when it does not find a solution on its own.

For more about agents, see [9].

## II. AGENT-BASED DBIST AND DBISR

### A. Generalities

The IEEE 1232 family of standards, analyzed in [10], describes 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.

It is important to have a communication layer, because the test and repair knowledge is distributed between the agent of the society and the central knowledge base. The agents do not communicate much, but when they need to, it is critical they have a reliable connection.

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.

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.

Our distributed testing methodology deals with environ-

ments whose subsystems are all able to run agent code or able to be controlled by other subsystems that can run code. If a subsystem cannot run the agent code, an agent from a nearby subsystem – one that is able to run agent code – will test/repair the first subsystem. This idea was previously discussed in [11,12].

A multi-agent approach and diagnosis ontology for diagnosis of spatially distributed heterogeneous systems is presented in [13]; however, in that approach, each subsystem has its own agent monitoring and diagnosing it, which can be costly in some cases. Another problem of this approach is that each system has an ad-hoc designed agent, so it lacks generality and flexibility.

Moreover, 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, diagnosing and repairing distributed systems. It offers many advantages like flexibility, easy maintenance, diagnosis tool for parts of the overall system. Monitoring and diagnosing faults is one of the application areas for agent-based systems. 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.

### B. The Agent Society

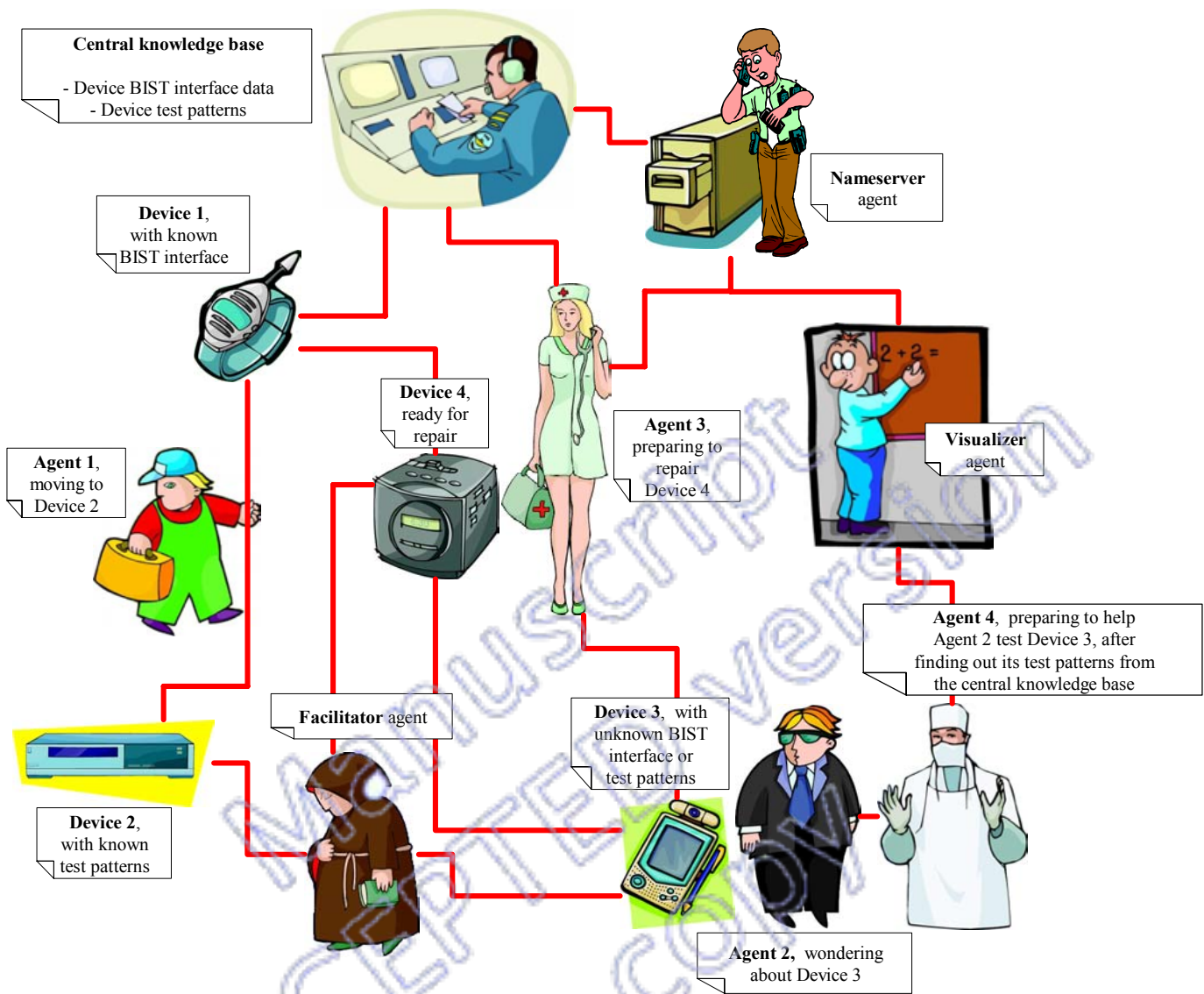
The agent society is able to share resources and repair the faults whenever possible. One or more agents diagnose each subsystem.

The agents travel from device to device, try to activate or to directly perform the test of a device and, if possible, to repair detected faults. Agents can perform these tasks either by themselves or with the help of other agents and a central database. They can also gather “experience” through their work.

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.

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.

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



When an agent has to analyze a specific subsystem (device) its main goal is to devise a way to test the device, to diagnose errors, and if possible to repair faults. These tasks require the agents to work in a very organized and coordinated way. In particular we can highlight the following macro-steps:

- 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.

The simplest case is when the device has BIST capabilities, and the agent knows how to access it. One of the most useful aspects of Built-In Self-Test is that the user (in our case, the agent) does not need to know the actual details of the testing process. The only thing the agent needs to know is how to activate the device's BIST functions, and, eventually, what parameters to pass to these built-in testing functions. If the

agent does not know how to access the BIST module of the device, it can ask other agents or the central database about it.

Another case is when the device does not have BIST, but has some previously generated and stored test sequences in its memory. This is not Built-In Self-Test, only some input values (test patterns or test vectors) for which the output values of the fault-free device are known. Usually, these test patterns are stored together with the corresponding expected outputs. These test patterns can be stored in the device itself, or in an agent, or a central database – knowledge base.

This approach is close to Software Implemented Hardware Fault Tolerance [14], where the system level fault tolerance is improved resorting to software, only.

In this situation, if the agent knows how to access them, it can extract and apply these test patterns. If not, the agent can ask other agents or a database about how to access and apply these test patterns in the device.

Of course, there may be cases when the device does not contain the test patterns in its memory, thus the agent has to request them from other agents or databases.



After detecting the fault, the agent starts a diagnosis (although most fault detection methods include diagnosis as well). In order to do this, the agent uses the same sources of information as for detection.

When the fault has been correctly diagnosed, the agent tries to repair it. It uses the same sources of information as for the detection and diagnosis. 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/repair agents (Agents 1-4 in fig. 3)
- 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.

If you look at Figure 3, you may notice that there are four Devices to be tested, four tester Agents, a central knowledge base, a Nameserver agent, a Visualizer agent, and a Facilitator agent.

Device 1 is a watch with radio capabilities. One or more of the agents, or the knowledge base, know how to activate the built-in self-test functions of the watch.

Device 2, the personal video recorder and DVD combo, does not have built-in self-test functions, but the agent society – knowledge base and the agents – has some input-output value pairs, or test patterns, for this device. Agent 1 is moving “inside” Device 2, to apply the known inputs, and then measure the outputs and compare them to the expected values.

Agent 2 does not know how to test Device 3, a PDA with wireless capabilities and attached camera – the agent does not have information about the device’s internal test functions, not even known outputs for given input values. Therefore, Agent 2 queries the Facilitator about someone with the skills to test a device of type Device 3. The Facilitator, who knows the abilities of all four tester agents, tells him that Agent 4 is familiar with Device 3 type devices. Happy that someone can help him, Agent 2 contacts the Nameserver, to communicate with Agent 4. The Nameserver introduces Agent 2 and Agent 4 to each other, and the two agree that Agent 4 will go and test Device 3. However, Agent 4 previously asked the central knowledge base about more efficient or new test methods for Device 3.

Device 4 is a storage server. Agent 3 just finished testing it, and found some unstable storage areas in the server. Fortunately, Device 4 is repairable on the field, because the data from unstable areas can be relocated, and the unstable zones can be marked bad. This is exactly what agent 3 is about to do.

Of course, all this exciting action cannot be seen from the outside, without the Visualizer agent. The Visualizer agent reports to the administrator of the system, in real-time or by keeping a log of events.

More about agent management can be found in [15].

### C. Agent communication

At software level, the agents communicate with each other through the FIPA (Foundation for Intelligent Physical Agents) ACL (Agent Communication Language) [15]. FIPA ACL specifications describe aspects of the structure of messages and the ontology service. 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 [15] 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. Our agents, in their current development status, use TCP/IP over wireline and wireless connections, with the messages in ASCII string format. They ask information from the central database through HTTP. Another variant uses XML to simplify inter-agent, agent-to-database communication and use of protocols like HTTP and WAP.

At hardware level, the agents use whatever communication layer is available for the device (serial, I2C, Ethernet or other). We have also considered embedded TCP/IP solutions.

For a system with mobile subsystems to be tested, short range, standardized radio-based Bluetooth chips can be used. For large scattered systems, radio-based Wi-Fi solutions or GPRS boards are available. Wi-Fi works even with public Access Points, while GPRS boards are adequate for low-cost, always-on sporadic communication over large distances.

### D. Implementation

The programming language of choice was Java, due mainly to its platform independence and strong network facilities, which make it ideal for distributed applications running on heterogeneous systems. From the large spectrum of available multiagent platforms, we selected the **Agents Development Kit (ADK)** from **Tryllian BV**, The Netherlands. This platform is built upon Java Standard Edition and offers a flexible, scalable and consistent task model for the agents’ behavior, a natively distributed multiagent environment, strong mobility for agents (i.e. both data and execution state are transferred along with the agent code), and last but not least, powerful and standards-compliant communication facilities. The interagent communication in ADK complies with a subset of the FIPA ACL standard.

In order to clarify how the testing society works, we shall give an example of such a society, detailing the tasks each type of agent must perform. The society is presented in figure 4. The ellipses enclose separate multiagent environments, the

agents themselves being represented as coloured labeled rectangles.

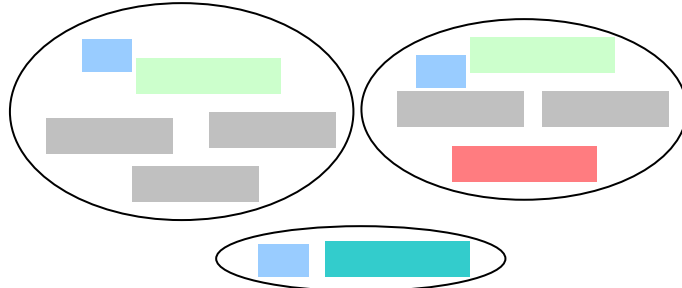


Fig. 4. Example of a testing society.

The *Directory Facilitator* (DF) implements the *Directory* service. Exactly one DF must be present at each location and each DF holds a map of the services (“things” that agents know how to do) supplied by all agents within its location, that can be updated and queried via messages. The DFs are capable of federating over the network – grouping in a cluster within which all services are accessible. The Dispatcher is responsible with supplying testers with device wrappers (see below), when they wish to initiate tests, and with balancing the testers load by transferring them between locations. The dispatcher holds wrappers for all devices to be tested present at its location, and maintains a ratings system that ensures that the first tested device is always the one that was tested earliest in the past. Exactly one dispatcher must be present at each location that can host tests (testers can travel and devices are “seen” by the society only through the dispatcher). DFs and dispatchers will hereafter be called service agents.

The Tester carries out the effective testing. Currently, the testers are able to perform vector testing and BIST. When they do not know how to test an encountered device, they ask among the other agents about it. Testers are also able to transfer between locations at the dispatchers’ requests, unregistering from the originating habitat and registering in the new one with all the services that they supply. The testers notify the *Visualizers* of each test outcome and of transfers. The visualizers are then responsible of informing the human supervisor of these events via Graphical User Interface.

The database *Connector* maintains the link between the agent society and the *database*, which stores information about all types of devices present in the system and test se-

quences on local memory. Tester agents use the database via the connector as a last resort when they cannot learn how to test a device from anywhere else. Typically, the connector resides either on the database machine or on a closely situated one, separated from the rest of the testing society.

The society is fully scalable, new locations can be added dynamically, testers and visualizers can be spawned at any location in the testing society at any moment, as long as the service agents conform to the requirements stated above.

A requirement imposed on the application was uniform handling of the devices by testers. This is accomplished by separating the devices’ physical and logical levels by means of an object called a *device wrapper* (see figure 5).

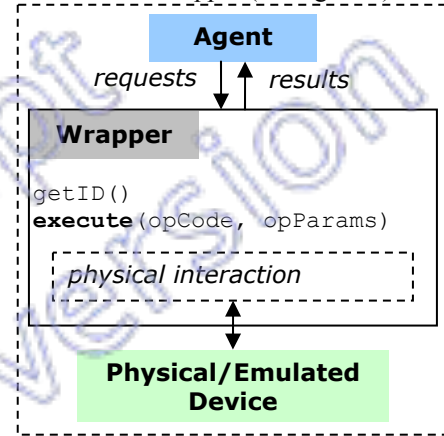


Fig. 5. Device wrappers.

All device functionality is accessed via the `execute()` method of the corresponding wrapper, which knows how to transmit signals to the hardware of the device and collect the results. Operations are identified by *operation codes*, such as `SUSPEND`, `RESUME`, `APPLY_VECTORS` etc. This model also facilitates the usage of emulated devices. Note also that a `getID()` operation is supplied by the wrapper. Device IDs uniquely identify device types.

## E. Experiments

### 1) Local testing

The test scenario includes two National Semiconductor *SCAN928028* 8 channel, 10:1 serializers with at-speed BIST capabilities, an Epson Electronics *S1L35043* LSI CMOS gate array, and a National Semiconductor *DP83840A* VLSI physi-

Options Languages

Devices

Device Ind...	Device ID	BIST	Faults
0	SCAN928028	yes	
1	SCAN928028	yes	
2	S1L35043	no	A20-s-a-0
3	DP83840A	no	

Testers History

[15:21:03.617] Device Index [2], Device ID [S1L35043], F: [0]  
[15:21:05.580] Device Index [1], Device ID [SCAN928028], F: [0]  
tester\_3  
[15:20:34.675] Device Index [1], Device ID [SCAN928028], F: [0]  
tester\_4  
[15:20:35.406] Device Index [3], Device ID [DP83840A], F: [0]  
[15:20:51.459] Device Index [3], Device ID [DP83840A], F: [0]

Fig. 6. A screenshot of the Visualizer agent

quences for all non-BISTed devices that do not store such cal layer device for 10-Base-T and 10-Base-X Ethernet. The

visualizer runs in the current stage of the application under Java2SE, and communicates with the tester microagents by FIPA ACL. A screenshot of the GUI of this agent is shown in figure 6. To the left the visualizer outputs a table of known devices at the given location. The devices discovered faulty are visually differentiated. To the right, a history of all operations performed by the testers present at the given location is maintained.

We shall explain in detail how a test takes place and how a tester is required to transfer and accomplishes this task, by setting first a test scenario.

The testing society initially includes two locations. At the first location, identified *loc\_0*, the device set includes a S1L35043. This IC does not have the capability to store the test sequence locally.

Obviously, a DF must reside on *loc\_0*, and, as devices to test are present, a dispatcher is also mandatory. A tester agent named *tester\_0* also resides here, and up to the present moment did not test any S1L35043. However, another tester named *tester\_1*, residing on a different location identified *loc\_1*, “knows” about S1L35043s and has registered this knowledge with the local DF. A visualizer also resides in *loc\_1*. Noting that another dispatcher must reside in *loc\_1*, and that a connector agent and a database are present in the society, but the example does not interact with them, we can summarize the situation in figure 7.

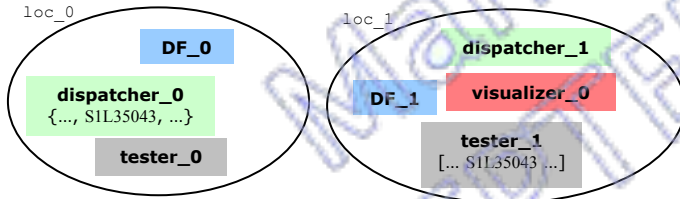


Fig. 7. Test scenario.

Each ADK agent receives a “clock signal” – *heartbeat*, from the ADK ARE (**A**gents **R**untime **E**nvironment), and each tester has a *test.rate* property, that gives the rate at which heartbeats initiate tests. The tester uses a random numbers generator to uniformly initiate tests over time. Let’s say that at heartbeat *k*, *tester\_0* decides to initiate a test. Therefore it asks *dispatcher\_0* for a device to test – the testers maintain the address of the local dispatcher internally, so the DF needs not be queried each time the dispatcher is needed. The dispatcher looks up its served devices table and sees that the device which hasn’t been tested for the longest time (or perhaps not at all) is the S1L35043, so it returns the S1L35043’s wrapper to *tester\_0*, also locking the S1L35043 in the table, so that another tester cannot gain access to it while it is being tested.

*tester\_0* queries the wrapper for the device ID, and is answered with S1L35043. It looks for S1L35043 in its devices “knowledge base” (we use quotation marks as this currently is just a hash table), and does not find it. It must then ask the society about it, and does so by issuing a query for the service S1L35043-info-supplier with DF\_0. DF\_0 does not find any provider locally, so it propagates the search in the DFs federation, which includes DF\_1. DF\_1 knows that *tester\_1* supplies

the requested service, so *tester\_1* is returned to DF\_0 in DF\_1’s subresults set. DF\_0 then forms the result set by joining all the results from the federation (which may or may not include testers other than *tester\_1*), and returns this result set to *tester\_0*. *tester\_0* then chooses at random an agent from the result set and asks it about S1L35043. If all works fine, the agent replies with the information. If not, after a timeout elapses, another agent from the result set is queried, and so on. If the results are drained, the tester resorts to the database. Let us assume that in this particular case, *tester\_0* has chosen to ask *tester\_1* about the S1L35043, and that the answer has been sent.

*tester\_0* saves the received information into its knowledge base and queries it to see whether the device supports BIST, and if not, whether a test sequence is locally stored. The S1L35043 does neither. So, the tester searches its knowledge base for a test sequence, does not find it, and the whole interaction pattern described above repeats until *tester\_0* gains possession of the test sequence. Note that the searched service name is this time S1L35043-test-sequence-supplier.

*tester\_0* can now perform the test. All interaction with the device is done via *execute()* calls. It first asks the wrapper to SUSPEND the S1L35043 – take it from the normal circuit flow and prepare it for testing. Then it issues two APPLY\_VECTOR operations, with the two input vectors from the test sequence, and reads the actual responses, comparing them to the expected ones. If they match, the device is RESUMED and the dispatcher is notified that the test has been completed, so it can unlock the device in its served devices table. If the results do not match, and if it can be done, the tester DISABLES the S1L35043, and then notifies the dispatcher of the test completion.

All testers maintain an internal periodically updated list of currently active visualizers, and they send each test outcome to all the agents in that list. *visualizer\_0* will therefore be informed of the test outcome and will reflect it in its GUI. The visual aspects of both faulty and fault-free test outcomes, together with their reflection in the devices GUI table, are shown in figure 6.

All interactions among agents, except the visualizer notifications, follow a relaxed version of the FIPA Request Interaction Protocol. The messages to the visualizer are simple informs, and the testers do not expect any confirmation. We exemplify below with the search query issued to DF\_0 by *tester\_0* for the providers of service S1L35043-info-supplier.

```
(request
:sender (tester_0)
:receiver (DF_0)
:subject (search)
:conversation-id (<automatically_generated>)
:content (
  search-id=<automatically_generated>
  search-key= S1L35043-info-supplier
  search-depth=2
  search-timeout=20
```



)
)
(agree
:sender (DF_0)
:receiver (tester_0)
:subject (search)
:conversation-id (<same_as_above>)
:content (
results-count=<results_count>
result-0=...
...
result-i=tester_1
...
result-<results_count>-1=...
)
)
(refuse /* sent if, for example, the search key was omitted */
:sender (DF_0)
:receiver (tester_0)
:subject (search)
:conversation-id (<same_as_above>)
:reason (missing-argument)
)

The search-depth content field specifies over how many DFs in the federation the search may propagate. Since all DFs know about each other, a depth of 2 suffices. The search-timeout field is self-explanatory. If the search is successful, the DF replies with an agree holding the results count (which may be 0) and the 0-based indexed list of results (agent addresses). If the search request is invalid, the DF replies with a refuse holding the refusal reason.

## 2) Agent Migration

Testers can move between locations at the request of the local dispatcher. The reason for which this agent issues move requests is *load balancing*. We define the *load factor* of a location as being the ratio of the number of served devices to the number of testers present at that location. Periodically, each dispatcher recomputes its own load factor and queries all other dispatchers about theirs. If the maximum remote load factor exceeds its own by at least the value of the *threshold* (given in percents and customizable via the threshold dispatcher property), the dispatcher randomly chooses a tester resident at its location and requests it to move to the heavier loaded location.

The tester first completes any test it was running, then tries deregistering all its services from the local DF. The deregistration is *atomic*, i.e. if any individual service deregistration fails, the process fails completely, the services are re-registered and the agent cancels the transfer. If the deregistration succeeds (and in a normal society state it always does), the agent moves, updates its internal references towards the local service agents, registers its services at the new location, and resumes normal operation. Whether it succeeds or not, the tester always informs the requesting dispatcher of the attempt outcome. Also, the visualizers are notified of a completed trans-

fer. Note that the dispatcher does not request another tester to move if the first one failed, rather any action is delayed until the next load balancing tick. The dispatchers avoid testers oscillation between location by not moving testers to the location from where they received the last tester.

We deepen the experiment scenario by adding a new location, identified `loc_2`, and by specifying the number of devices and testers at each location, as in table 1.

	loc_0	loc_1	loc_2
Testers	7	2	4
Devices	10	10	7

Table 1. Location structures.

The initial load factors of the locations are, respectively, 1.43, 5 and 1.75. Assuming that the thresholds are all 30%, at the first load balancing tick, `dispatcher_0` will see that the load factor of `loc_1` exceeds its own by 149% and will send a tester there. Load factors change to, respectively, 1.66, 3.33 and 1.75. Assuming `loc_2`'s first balancing tick occurs a bit later, the dispatcher there will determine that `loc_1`'s load factor exceeds its own by 90%, and will request a tester to move there. The process continues in a same manner and eventually reaches a steady state in which the load factors are as follows: 2.5, 2, 1.75.

## III. CONCLUSIONS AND FUTURE WORK

The multiagent solution, being a natural approach to the DBIST problem, and to distributed testing in general, offers significant advantages over traditional solutions, among which the most important are:

- a great increase in the flexibility and scalability of both the system under test and of the testing system itself;
- greater speed due to parallelism;
- reduction of the communicational overhead due to decentralized management;
- the high level of application modularity eases maintenance and further development.

The areas in which further work needs to be done include redesigning the agent behaviour to include "real" artificial intelligence, more attention over the testing algorithms themselves, which at this point are rudimentary, designing and implementing an efficient physical level of the wrappers for various types of devices, and an eventual migration of the application to Java Micro Edition [16], in order to enlarge the range of machines on which the application can run, thus extending the area over which the testing can occur.

## ACKNOWLEDGEMENT

The authors would like to thank dipl. eng. Lucian Buşoniuc for his help on agents.

## REFERENCES

- [1] Abramovici, M., Breuer, M.A., Friedman, A.D., *Digital systems testing and testable design*, New York, Computer Science Press, 1990.

- [2] Rochit Rajsuman, *Digital Hardware Testing: Transistor-Level Fault Modelling and Testing*, Artech House, Boston, London, 1992.
- [3] Janusz Rajski, Jerzy Tyszer, *Arithmetic Built-In Self-Test for Embedded Systems*, Prentice Hall PTR, 1997.
- [4] L. Miclea, Enyedi Sz., R. Orghidan, "On line BIST Experiments for Distributed Systems", in *Proc. IEEE European Test Workshop ETW 2001*, Stockholm, Sweden, 2001, pp. 37-39.
- [5] L. Miclea, D. Cimpoca, M. Gordan, "An On-Line BIST Structure for Distributed Control Systems", in *Digest of IEEE European Test Workshop ETW 2000*, Cascais, Portugal, 2000, pp. 283-284.
- [6] A. Benso, S. Chiusano, S. Di Carlo, "HD2BIST: a Hierarchical Framework for BIST Scheduling, Data Patterns Delivering and Diagnosis in SoCs", in *Proc. International Test Conference ITC 2000*, Atlantic City, NJ, USA, 2000, pp. 899-901.
- [7] Monica Lobetti Bodoni, A. Benso, S. Chiusano, G. di Natale, P. Prinetto, "An Effective Distributed BIST Architecture for RAMs", in *Informal Digest of IEEE European Test Workshop ETW 2000*, Cascais, Portugal, 2000, pp. 201-206.
- [8] R. Pendurkar, A. Chatterjee, Y. Zorian, "A Distributed BIST Technique for Diagnosis of MCM Interconnections", in *Proc. International Test Conference ITC 1996*, Washington, DC, USA, 1996, pp. 214-221.
- [9] J. Ferber, *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, 1999.
- [10] J. Sheppard, M. Kaufman, "IEEE 1232 and p1522 standards", in *Proc. AUTOTESTCON*, Anaheim, CA, USA, 2000, pp. 388-397.
- [11] L. Miclea, Enyedi Sz., A. Benso, "Intelligent Agents and BIST/BISR - Working Together in Distributed Systems", in *Proc. International Test Conference ITC 2002*, Baltimore, USA, 2002, pp. 940-946.
- [12] L. Miclea, Enyedi Sz., "Distributed Built-In Self-Test using Intelligent Agents", in *Proc. IEEE European Test Workshop ETW 2002*, Corfu, Greece, 2002, pp. 17-19.
- [13] I. A. Letia, F. Craciun, Z. Köpe, A. Netin, "Distributed diagnosis by BDI agents", in *Proc. International Conference on Applied Informatics IASTED*, Innsbruck, Austria, 2000, pp. 862-867.
- [14] Alfredo Benso, Silvia Chiusano, Paolo Prinetto, *A COTS Wrapping Toolkit for Fault Tolerant Applications under Windows NT*, IOLTW 2000: IEEE International On-Line Test Workshop, Majorca (ES), July 2000, pp. 9-16.
- [15] *FIPA standards and specifications*, Foundation for Intelligent Physical Agents, 2002. Available: <http://www.fipa.org>
- [16] Qusay Mahmoud, *Learning Wireless Java*, O'Reilly, 2002.

#### IV. BIOGRAPHIES



**Liviu Miclea** (M'2000) was born in Unirea, Romania, on the 11<sup>th</sup> of January, 1959. Mr. Miclea graduated the Informatics High School of Cluj-Napoca, Romania, in 1978, earned his engineering diploma at the Faculty of Automation and Computer Science of the Technical University of Cluj-Napoca in 1984, and his PhD in Automatic Systems in 1995, at the same university.

Between 1984 and 1995, he worked as Engineer, Scientific Researcher and Senior Researcher at the Institute for Automation IPA Bucharest, Cluj-Napoca subsidiary. From 1995 until 1998 he taught Reliability and Diagnosis as well as Systems Theory, as Assistant Professor, at the Automation Department of the Technical University of Cluj-Napoca. In 1998, he earned his degree of Associate Professor at the Automation Department. Currently, he is the Head of the Automation Department at the Technical University of Cluj-Napoca. He is author or co-author of 10 books, 20 research works and 50 scientific publications (of which in 23 as unique or first author). His research interests include: design for testability, automatic testing, computer aided design, distributed systems.

Dr. Miclea is member and liaison for Romania of IEEE-TTTC (IEEE Test Technology Technical Council), member of IEEE Computer Society, IEEE Communications Society, IFAC Foundation Romania and the Romanian Society of Automatics and Technical Informatics SRAIT.

His IEEE achievements in 2003 include a Certificate of Appreciation from the IEEE Computer Society, for more than five years of fructuous IEEE activity, and the Award of Excellence for an ITC paper and its presentation. He is regular co-chairman of the bi-annual IEEE AQTR conference and often reviews IEEE ITC, ETW and VLSI conference papers.



**Enyedi Szilárd** (M'2002) was born in Cluj-Napoca, Romania, on the 26<sup>th</sup> of December, 1976. Mr. Enyedi graduated Apáczai Csere János High School of Cluj-Napoca, Romania, in 1995, earned his engineering diploma at the Faculty of Automation and Computer Science of the Technical University of Cluj-Napoca in 2000, and his MSc in Automatic Systems in 2001, at the same university. He is currently preparing his PhD in Telecommunications

there.

Since 2000, he teaches Reliability and Diagnosis, CAD, Computer Programming, Process Equipments and Interfaces, Distributed Control Systems, Application-Oriented Software Environments, Operating Systems and Computer Networks and Internet Technologies at the Automation Department of the Technical University of Cluj-Napoca, and Delphi courses at the Chamber of Commerce of Cluj-Napoca. Since 2001, he is Research Assistant, at the Automation Department. He is co-author of one book, 6 research works and 17 scientific publications. His research interests include: internet applications, wired/wireless communication, design for testability, automatic testing, computer aided design, distributed systems, embedded systems, digital electronics and control.

Mr. Enyedi is member of IEEE Communications Society since 2003.

His IEEE achievements in 2003 include an ETW poster and the Award of Excellence for an ITC paper. He is also a regular co-organizer of the bi-annual IEEE AQTR conference.



**Paolo Prinetto** was born in Gassino Torinese, Italy, on March 17, 1953. He received the M.S. in Electronic Engineering in 1976 from the Politecnico di Torino, Italy. Since 1990 he is full professor of Computer Engineering at the same University, and, since 1998, joint professor at the University of Illinois at Chicago. His research interests cover testing, test generation, BIST and dependability. He is a Golden Core Member of the IEEE Computer Society and the elected chair of the IEEE Computer Society Test Technology Technical Council (TTTC).



**Alfredo Benso** was born in Torino, Italy, on November 28, 1970. He received his M.S. degree in Computer Engineering (1995) and his Ph.D. (1998), from the Politecnico di Torino, Italy. He is currently a researcher at the same university, where his research interests include Design-for-Testability techniques, BIST for complex digital systems, dependability analysis of computer-based systems, and software-implemented hardware fault tolerance (SWIHFT). He is the chair of the IEEE Computer Society Test Technology Technical Council (TTTC) Web-based Activi-

ties Group.