IMPLEMENTATION OF A MULTIAGENT DBIST/DISTRIBUTED TESTING SOLUTION

Liviu Miclea[1], Lucian Buşoniu[2], Enyedi Szilárd[3]

Technical University of Cluj-Napoca
Department of Automation
26-28 Baritiu Str., RO-400027, Cluj-Napoca, Romania
Tel.: +40-264-594469
E-mail: [1]Liviu.Miclea@aut.utcluj.ro, [2]lucian_busoniu@yahoo.com, [3]Szilard.Enyedi@aut.utcluj.ro

Abstract: This paper aims to enhance and particularize the concepts presented in [1] for the purpose of designing and implementing a multiagent testing solution for distributed, heterogeneous and geographically scattered systems in the Java programming language. We present the essential concepts of the solution adapted to the practical problem at hand, and then detail the way the implemented agents work together towards accomplishing their common goal of diagnosing the distributed system. The presentation focuses consecutively on the utilized agent platform, on the knowledge sharing and exchange throughout the society, and on a workload balancing mechanism for the tester agents.

Keywords: system testing, distributed testing, agent, multiagent system.

1.   INTRODUCTION

Plain BIST and BISR are not suited for the testing, diagnosis and repair of heterogeneous, distributed and geographically scatterred systems, such as nationwide telecommunications or energy distribution systems. The decentralization 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 (capability named hereafter *agent mobility*).

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 (physically realized rather as communicating multiple multiagent environments). We shall call a multiagent society whose main actor is the tester agent a *testing society*.

2.   IMPLEMENTATION

The 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 inter-agent communication in ADK complies with a subset of the FIPA ACL standard (see [3], [4]).

In order to clarify how the testing society works, we shall exemplify, detailing the tasks each type of agent must perform. The society is presented in figure 1. The dashed rectangles enclose separate multiagent environments, the agents themselves being represented as labeled rectangles with differentiating patterns.
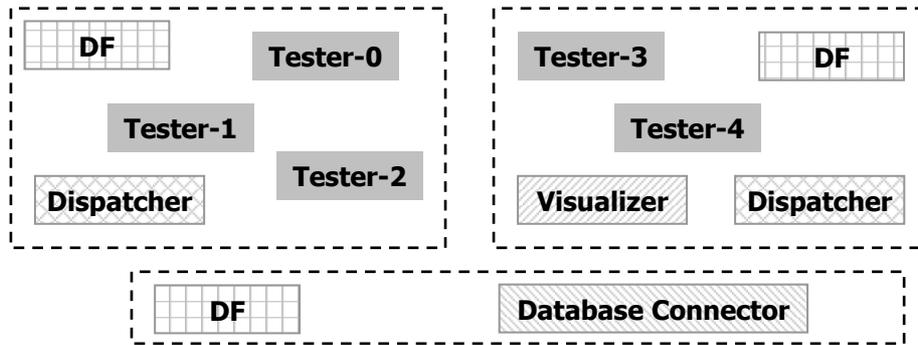
Figure 1. Example of a testing society

The *Directory Facilitator* (DF) implements the *Directory* service specified in [5]. 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 first tested device is always the one that was tested earliest in the past. 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, deregistering 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 a Graphical User Interface.

Figure 2. Device wrappers

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 sequences for all non-BISTed devices that do not store such sequences 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
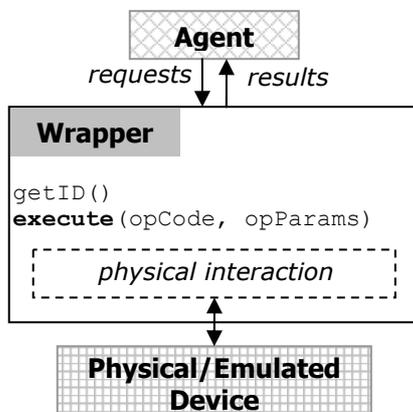
requirements stated above.

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 2). 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.

3.   EXPERIMENT

We 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. We start with a more limited scenario and detail it as we go on with the example.

3.1.   *Devices testing*

The testing society initially includes two locations. At the first location, identified `loc_0`, the devices set includes a 7404, 6 inputs inverter gate, having device ID `G7404`. This gate obviously does not have the capability to store the test sequence locally. For demonstrative reasons, we devise an extremely simple test sequence composed of only two test vectors, given in table 1.

Input vector	Expected response vector
010101	101010
101010	010101

Table 1. Test sequence

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 7404 gates. However, another tester named `tester_1`, residing on a diferrent location identified `loc_1`, "knows" about 7404s and has registered this knowledge with the local DF. A visualizer also resides on `loc_1`. Noting that another dispatcher must reside on `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 3.

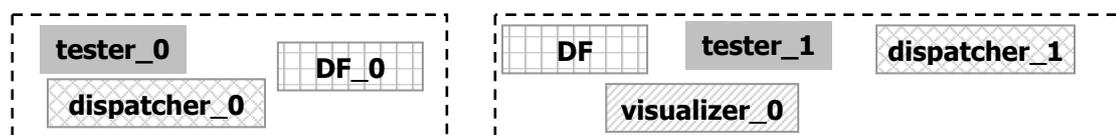**Figure 3.** Test scenario

Each ADK agent receives a "clock signal" – *heartbeat*, from the ADK ARE (**Agents Runtime Environment**), 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 7404, so it returns the 7404's wrapper to `tester_0`, also locking the 7404 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 G7404. It looks for G7404 in its devices knowledge, and does not find it. It must then ask the society about it, and does so by issuing a query for the service G7404-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 G7404. 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 7404, 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 7404 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 G7404-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 7404 – 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 7404, 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 4. You might also want to consult the full visualizer screenshot in figure 6.
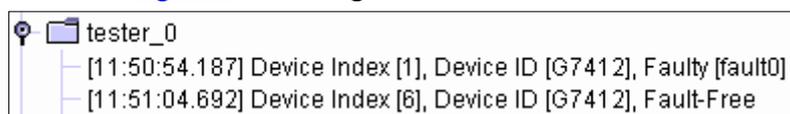
Figure 4. Visual test outcome

All interaction among agents, except the visualizer notifications, follows a relaxed version of the *FIPA Request Interaction Protocol*, described in [6]. The messages to the visualizer are simple informs (see [7]), and the testers do not expect any confirmation. 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. For a thorough description of all unclear terms and message fields, and also of all the interaction patterns followed within the testing society, see [7] and [8].

3.2. Agents 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 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, and 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. The dispatchers avoid testers' oscillation between locations 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 2.

	loc_0	loc_1	loc_2
Testers	7	2	4
Devices	10	10	7

Table 2. 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 eventually reaches a steady state in which the load factors are as follows: 2.5, 2 and 1.75.

The visual appearance of the agent transfer is presented in figure 5, both at the originating and the destination locations.

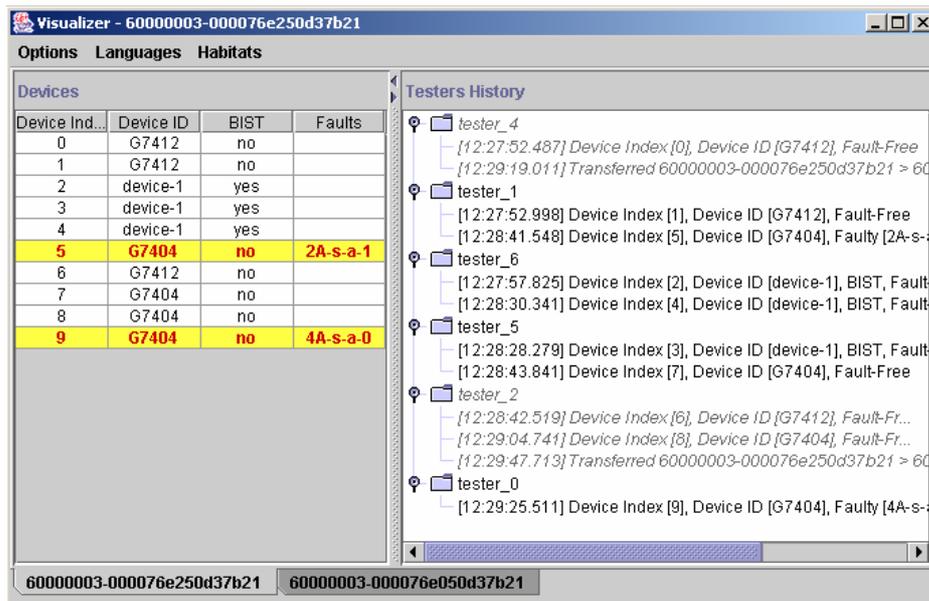**Figure 5.** Visual transfer outcome

Figure 6. Visualizer agent screenshot

4. 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.

Further work needs to be done on adapting the agent behavior to use an inference engine, advances in the test algorithms, and the design and implementation of an efficient physical level of the wrappers for various types of devices. An eventual migration of the application to Java MicroEdition is also considered, for the purpose of enlarging the range of machines on which the it can run, thus extending the area over which the testing can occur.

5. REFERENCES

[1] L. Miclea, Sz. Enyedi, G. Toderean, A. Benso, P. Prinetto (2003), *Agent Based BIST/DBISR and its Web/Wireless Management*, International Test Conference, Charlotte, NC, USA, September 30th - October 2nd 2003.

[2] L. Busoniu (2003), *Multiagent Systems in DBIST and Distributed Testing*, eng. Diploma Thesis, Technical University of Cluj-Napoca, Department of Automation

[3] *** *Fipa Agent Communication Specifications*, http://www.fipa.org/repository/aclspecs.html

[4] *** *ADK Developer's Guide* Release 2.1 (2002), Tryllian Holding BV

[5] *** *FIPA SC00001, Abstract Architecture Specification,* http://www.fipa.org/specs/fipa00001/SC00001L.html

[6] *** *FIPA SC00026, Request Interaction Protocol Specification,* http://www.fipa.org/specs/fipa00026/SC00026H.html

[7] *** *FIPA SC00037, Communicative Act Library Specification,* http://www.fipa.org/specs/fipa00037/SC00037J.html

[8] *** *IDBIST Application JavaDoc,* http://lbusoniu.inginer.com/idbist-apidocs/index.html