Journal of Computer Science 3 (4): 249-258, 2007
ISSN 1549-3636

MAEST :Multi-Agent Environment for Software Testing

Ramdane Maamri and Zaidi Sahnoun
Lire Laboratory, Computer Science Department, University of Mentouri, Algeria

Abstract: The test is a significant aspect of software development and plays a considerable role in detecting errors in the implementation phase. As the software becomes more pervasive and more often employed to achieve critical tasks, it will be increasingly required to be of high quality. Significant reductions in the cost of software development and software maintenance could be achieved if we can find efficient ways to perform effective and rapid testing process.

Key words: Testing agent, multi-agent, test cases, testing process

INTRODUCTION

Software testing is a very labour-intensive and very expensive process. Studies indicate that more than fifty percent of the total cost of software development with the percentage for testing critical software being even higher. Software testing is also a very costly part of software maintenance in terms of contribution in the total time of the process of maintenance. Significant reductions in the cost of software development and software maintenance could be achieved if we can find efficient ways to perform effective and rapid testing process.

Although many computer aided software test tools are available today, most are limited to automating only one part of the test effort. A testing tools that minimize tester interference, cuts down on testing time and carries out the tests independently needs to be developed in the software development scene, we therefore propose a new testing environment that minimize the tester interference and provide an open test multi-agent environment which can help during the whole testing process.

In this article, we propose a multi-agent system using agents to provide assistance during the whole testing process. This system has several characteristics. Firstly, it minimizes the interference of the tester by automating the process of test. Secondly, by intellectually selecting redundant free and consistent and effective test cases, the testing time is reduced while the fault detection ability increases. Thirdly, architecture suggested is open and extensible. It supports dynamic addition, suppression of the agents and the services. Lastly, the agents can be located in only one computer or a network.

Test levels: The testing view has evolved during the last few years and the testing is no longer considered as the last activity which begins after the coding phase is completed. The testing activity is seen now as a process that begins from the development phase to the maintenance phase. Each activity in the development or the maintenance process should have a corresponding part in the testing activity

Each testing phase in Fig. 1, is dedicated to particular class of errors[1], the aim of acceptance test is to verify that user requirements are respected, the goal of system test is to verify the system specification are respected, the integration test is to verify that the interfaces between software units respect their specifications, while the unit test is to verify tat each software unit operate as defined in its specification.

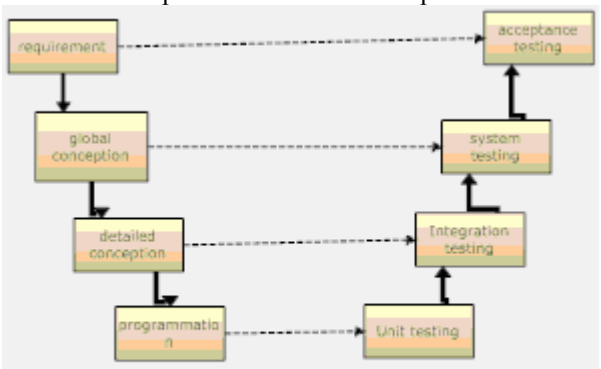

Fig1. Development and testing processes

Corresponding Author: Ramdane Maamri, Lire laboratory, Computer Science Department, University of Mentouri Algeria

The testing process: The testing process can be divided mostly in three different phases: test generation, test execution and Test result analysis.

a) Test generation phase: Involves analysis of the specification and determination of which functionalities will be tested, determining how these can be tested and developing and specifying test scripts.

The test cases are mostly generated by using two strategies**:** White box and black box strategy.. In black box testing (functional testing)[2], (e.g. partition, flow testing, syntax testing, domain testing, logic-based technique and state testing) only the outside of the system under test is known to the testers. They generate test cases based on requirement and design information. While the white box testing (Structural techniques)[3], (e.g. control flow graph path testing) are based on internal code. There are mainly three types of structural testing techniques: control flow based testing, data flow based testing[4] and mutation testing[5]. Control flow based testing coverage criteria expresses testing requirement in terms of nodes, edges, or path in the program control flow graph. Data flow based testing coverage criteria expresses testing requirements in terms of the definition-use association present in the program. Mutation testing begins by creating mutants of the original program. The changes made in the original program correspond to most likely errors that could be present. The goal of the testing is to execute the original program and its mutants on test cases that distinguish them from each other.

Naturally, the distinction between black and white box testing leads to many gradations of grey box testing, e.g., when the module structure of a system is known, but not the code of each module.

b) Test execution phase: Involves the development of a test environment in which the test scripts can be executed.

c) Test result analysis: When all test events have been carefully registered, they can be analyzed for compliance with the expected results, so that a verdict about the system's well-functioning can be assigned.

Multi agent environment for software testing (MAEST):

Multi-Agent systems[6] have been identified as essential to the successful engineering of complex or large systems. A multi-agent system is composed of a group of agents that are autonomous or semiautonomous and which interact or work together, to perform some tasks or achieve some goals. The agents in such systems may either be homogeneous or heterogeneous and they may have common goals or goals that are distinct [7]. Several methodologies have been proposed for the development of multi-agent systems. Most of them are either an extension of object-oriented methodologies: GAIA[8], multi-Agent Software Engineering MaSE[9-11], or an extension of knowledge-based methodologies: COMOMAS[12], MAS-Common KADS[13].

In this paper, we propose a multi-agent system, named MAEST, which purpose is to provide assistance to testers in the test process. This section presents the beginning of its specification. When designing and specifying MAEST, we used Multi agent Systems Engineering (MaSE), a methodology for developing heterogeneous multi agent systems. MaSE uses a number of graphically based models to describe system goals, behaviors, agent types and agent communication interfaces. MaSE is also associated with a tool, agentTool, which supports the methodology.

The first task when designing agent and multi-agent systems is to identify goals and sub-goals. In MaSE, this is made during the Capturing Goals step. This step consists of two sub-steps: identifying goals and structuring them in a Goal Hierarchy Diagram.

The system under test is composed out of subsystems communicating with and affecting each other. Each of the components may be used in completely different configuration. In general each subsystem is tested by team of testers using both static and dynamic approaches in order to increase the quality of the system. The main task of the supervisor is to coordinate control and inspection activities of integrated test. However, when testing complex systems it is not sufficient to support the aspect of coordinating test tools only, but also the whole process, from the test specification to the analysis of test results. Therefore, the following aspects of the test process have to be supported by any integrated test environment:

- Organisation of test relevant data,
- Design of test cases and composition of test suite,
- Coordination of test execution and Analysis of test execution results.

The Goal Hierarchy Diagram of MAEST is shown in Fig. 2.

MAEST architecture: The basic components of our multi-agent testing environment are shown in Fig. 3. It consists of four types of agents:

* Administrator agent
* Testing agent
* Interface agent and
* helping agents (TGC agent, oracle agent, execution agent and verdict agent).

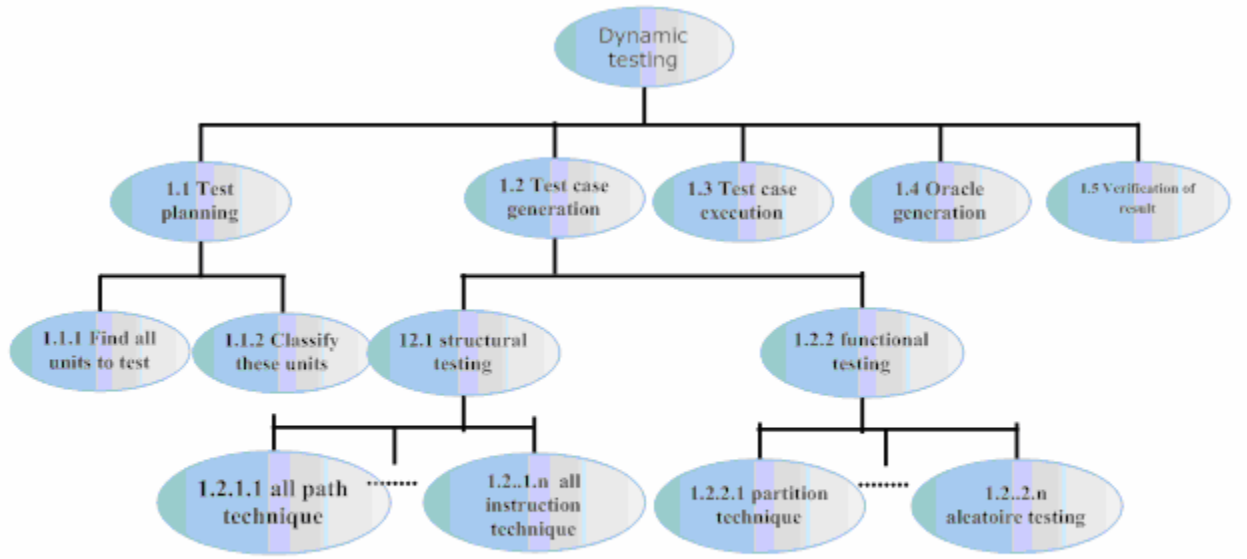

Fig 2. Goal Hierarchy diagram of MAEST

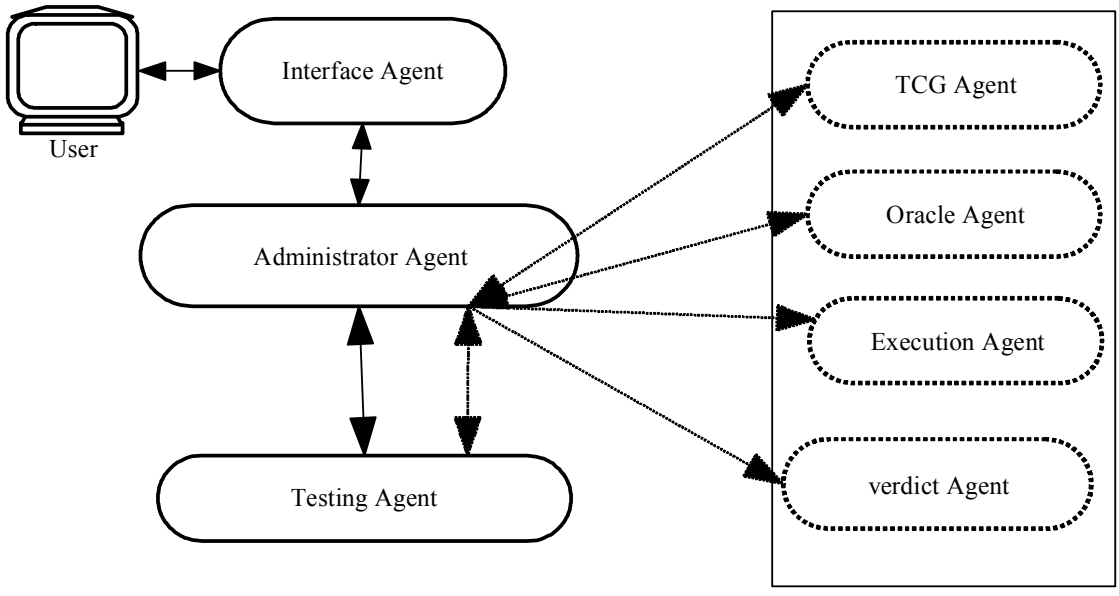

Fig. 3: Proposed MA system architecture

The proposed MAS is completely decentralized. Each agent runs locally or on different machines in a network. Further, Agents can dynamically join and leave the system to achieve the maximum flexibility and extendibility. The only links between the agents represents a service level dependency. Such a

dependency can represents a fact that one agent depends on another for a goal to be fulfilled, task to be performed, or a resource to be made available. A test task can be decomposed into many small tasks until it can be carried out directly by an agent. The decomposition of testing tasks is also performed by agents. More than one agent may have the same functionality, but they may be specialized to deal with different information formats, executing on different platforms, using different testing methods or testing criteria, etc. The overall architecture of the MAEST environment is shown in Fig. 3.

Administrator agent: Administrator Agent manages the whole system, processing the complex communication between the inside of the system, coordinating all the Agents in the system and distributes controls in the system. It is unique in the system. The main functions of administrator Agent include:

* The interface of the system with other systems or Agent systems;
* Administrating Agent registers the table of the system;
* Coordinating the interaction of Agents in the same system;
* Creating and administrating all the active Agent instances, including the status and life cycles of active Agents;
* Processing the communication of Agents.

The Agent register table includes all information that identifies the Agent, besides its ID, address, name, etc. It is an important part that records the specific method and the service function of the cooperation of Agents. According to this register table, administrator Agent creates Agent instances and uses them to create an instance, it is necessary to analyze the testing request, then try to find out what type of Agent need to be created and how to create it by accessing the register information of the administrator Agent. Finally, create it.

Plan construction algorithm
input: the product to test (a set of units)
output: sets of units to be tested in parallel and their orders.

Algorithm
1- Construct calling tree
2- i ←0
3- if there is no leaf in the tree
  S_i one node of the high level
  Replace this node by a stub
 else
  S_i ← all the leaves of the call's tree
4- Eliminate all nodes included in S_i
5- i← i+1

6- repeat (3-5) until root of the tree is included in S_i

Plan execution algorithm
input: set S={ Si /(Si is a set of units which can be tested at in the same time i}
output: global rapport of the testing process
Algorithm:
1 i← 0
2 for each unit in the set Si
  -Send a message to testing agent
  - Receive a verdict message from the testing agent
  - If unit has been detected faulty, replace it by driver
3 i← i+1
4 repeat (2) et (3) until (i=j / the root of the calling tree is in Sj)

Example of the messages treated by administrator agent:
* Administrator Agent receive an assertive message from a new agent ; after registering all the information about this agent (capability, address , etc..) in its data base and creating a new mailing box ,it sends an assertive message containing the address of the mailing box to this new agent.
* Administrator Agent receives an expressive message from an interface agent; it constructs a testing plan and sends a sequence of expressive messages to the testing agent in order to execute the plan.
* Administrator Agent receive an expressive message from all other agents (except the interface agent), it forwards it the appropriate agent.
* Administrator Agent receive an assertive message from a (clone of) testing agent, it registers the partial results contained in the message in its data base.
* When the administrator agent receives an assertive message from the (and all clones of) testing agent or the allowed time is over, it sends an assertive message to the interface agent containing the final result of testing process.

Testing agent: The main objective of the testing agent is to supervise the all testing process from the test cases generation to the final verdict of a unit. It takes formally recorded specification and code information and then sends message asking for test cases. When it receives different test cases from different test cases generation agents, it tries to take optimal test suite by eliminating redundant test cases. The testing agent uses two types of rules; the first one is applied to select

redundant free test cases and the second one to select consistent test cases.

Redundant test cases rule

let x and y set of test case

$$non\ redundant\ set\ of\ test\ cases(x,y) = x \cup y - x \cap y$$

Consistent test cases rule

let x and y set of test case

Consistent test case (x,y)

$$= \begin{Bmatrix} x & if\ x\ is\ generated\ by\ more\ general\ approach\ then\ the\ one\ used\ to\ generate\ y \\ y & otherwise \end{Bmatrix}$$

For example the set of test cases generated by condition approach is more general then the one generated by instruction approach.

Example of the messages treated by testing Agent: *

* When the testing agent receive expressive message from the administrator agent; it treat the information contained in the message and then sends an expressing message to each test cases generating agent(TGC) asking them to generate test cases for the given unit.

* When the testing agent receives an assertive message from a TGC agent, it stores the set of test cases contained in the message in its local base.

* When the testing agent receive an assertive message from each TGC agent or the time is over, it takes all sets of test cases stored in its local data base and creates one optimized set of test case and then sends an expressive message to program execution agent and the oracle agent.

* After testing agent receives an assertive message from both program execution agent and oracle agent, it sends an expressive message to the verdict agent.

* Once an expressive message is received from the verdict agent, the testing agent forwards it to the administrator agent.

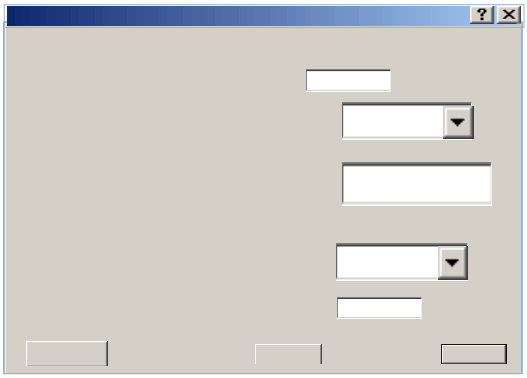

Fig. 4: Input interface

Interface agent: In our environment, interface Agent is designed to achieve the interaction between system and users. It realized the generality of user interface. After requesting testing command, the only thing we need to do is just to wait easily but need not know how or where test is performed. Agent will do everything for us. And the system will return the final result to users. One advantage of interface Agent is that it is a friendly interface. The main functions of interface Agent include:

* Describing user's testing request in some kind of visual format;
* Communicating with other agents of the system, submitting the testing request and returning the resulting rapport to user;
* When user gives an incorrect request description (such as file not found error) interface Agent will display error information and prompt user to correct it.

The main role of this agent is receiving, from the user, program information such as (Fig. 4):

* Program file
* Programming language
* Specification file
* Specification language
* time allowed

Transforms these information in KCML message and sends it to administrator agent. When it receives a message from the latter agent at the end of the testing process, it treats the message and shows the final reports about the results of the testing process such as (Fig. 5):

* The number of the tested units
* The passed units
* The failed units

For each failed unit, which test cases caused failure?

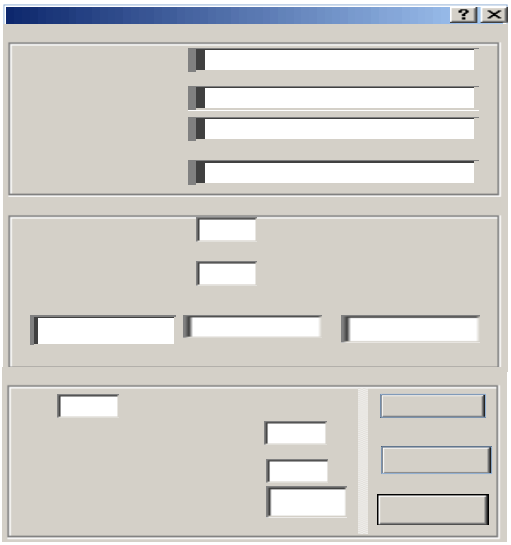

Fig. 5: Result interface

Helping agent: The software architecture of these agents reflects our desire to reuse the existing tools for test case generation, oracle agent and verdict agent. Each one of these agents consists of a core implementing the functionality of the tool and the local user interface and a wrapper which gives the tool the behaviour and the appearance of an agent.

Program execution agents: To enable concurrent execution of multiples test cases, The administrator agent may invokes one program execution agent for each test cases. So we can initiate execution of individual test case or a test suite (i.e., the test cases in a test series) with various parameters choosing which test cases to execute and when to stop.

Program execution agent simply invokes an executable component; so Components to be tested may be developed in any language. The tested component may be a single procedure during unit testing, a set of integrated components during integration testing, or the entire system during software system testing. This agent develops the driver and stubs for unit and integration testing as necessary.

Oracle agent: A test oracle is mechanism for specifying correct and /or expected behaviour and verifying that test execution meet that specification. Testing process is of little importance if we can not verify the behavioural correctness. Most of testing research has neglected the issue of oracles. They focus only on defining what to test without checking the behavioural results, thereby ignoring the test oracle and requiring manual checking of test results. In that most test criteria require high number of test cases, manual checking make the testing process insure - the test executions may be run, yet the goals of testing are not achieved since results may be checked only manually.

Test generation cases (T.G.C) agent: The main objective of each T.G.C agent is to assist a tester in the generation of test cases for software using one testing approach. It takes formally recorded needed information (specification and or code information) of the unit under test and then uses one testing technique to generate test cases suite.

Our system is designed as an open system, it is relatively easy to add new T.G.C agent with different core to the set of the testing agents.

Verdict agent: The main work of the verdict agent is to analyze the correctness of the test run. It compares the expected output and real output and gives its verdict.

Cloning of agents: When a particular agent is need but not free, the administrator agent creates a clone of this agent, which is identical in that it has exactly the same behaviour. The clone agent can manage and control itself on a local dimension and interact directly with its originator to exchange, provide and receive services, data. The life time of the clone is limited to end of action work.

The Agents Intercommunication: In our system, agents communicate with the administrator by messages sending. The information contained in the messages can be divided into two types:

* Testing tasks descriptions, which include requests of testing tasks to be performed and reports of the results of testing activities;
* Agents description, such as the capability of an agent to perform certain types of testing tasks and its resource requirements such as hardware and software platform and the format of inputs and outputs. Such information are represented in an ontology[14] about software testing.

A message goes through several stages before being processed. Once a message is received by an agent, it is passed on to its communication component which takes as input the string KQML message and converts it into a KQML message object and then test the validity of the message, as well as the value of various fields of the message examples of which are the sender, the content, etc. and then checks if the message is invalid, the appropriate error message is dispatched to the sender. If it is valid, the KQML message object is send to the planning processor.

Message communication mechanism: The message mechanism consists of a set of communication primitives for message passing between agents[14]. Its design objectives are generally applicable, flexible, lightweight, scaleable and simple.

The communication mechanism used in our system is based on the concept of message box (an unbounded buffer of messages). All messages are sent to mboxes and stay there until they are retrieved by agents.

Each mbox is identified in the system with a different id. However, its location is transparent to the agents. Given an mbox id, the agents can operate the mbox without knowing its physical location. The mbox can be opened by more than two agents at the same. For example, a administrator agent has a mbox to receive task requests. Multiple agents can send message to this mbox.

Ontology of software testing: Ontology defines the basic terms and relations comprising the vocabulary of a topic area, as well as the rules for combining terms and relations to define extensions to the vocabulary[15]. It can be used as a means for agents to share knowledge, to transfer information and to negotiate their actions[16]. For this reason, we designed ontology for software testing which takes in consideration the following aspects:

Software testing activities occur in various software development stages and have different testing purposes. For example, unit testing is to test the correctness of software units at implementation stage. The context of testing in the development process determines the appropriate testing methods as well as the input and output of the testing activity. Typical testing contexts include unit testing, integration testing, system testing and regression testing and so on.

There are various kinds of testing activities, including test planning, test case generation, test execution, test result verification, test coverage measurement, test report generation and so on.

For each testing activity, there may be a number of testing methods applicable. For instance, there are structural testing, fault-based testing and error-based testing for unit testing. Each test method can be further divided into program-based and specification-based. There are two main groups of program-based structural test: control-flow methods and data-flow methods. The control flow methods include statement coverage, branch coverage and path coverage, etc.

Each testing activity may involve a number of software artefacts as the objects under test, intermediate data, testing result, test plans, test suites and test scripts and so on. Testing results include error reports, test coverage measurements, etc. Each artefact may also be associated with a history of creation and revision.

Information about the environment in which testing is performed includes hardware and software configurations. For each hardware device, there are three essential fields: the device category, the manufacturer and the model. For software components, there are also three essential fields: the type, product and version.

The capability of a tester is determined by the activities that a tester can perform together with the context for the agent to perform the activity, the testing method used, the environment to perform the testing, the required resources (i.e. the input) and the output that the tester can generate.

Consists of a testing activity and related information about how the activity is required to be performed, such as the context, the testing method to use, the environment in which to carried out the activity, the available resources and the requirements on the test results.

Figure 5 shows a message send by a new agent to the administrator agent expressing its capability. The agent is capable of doing path coverage test case generation in the context of unit testing of a program written in C language.

Communication protocol: In our system, agents of similar functionalities may have different capabilities and are implemented with different algorithms, may be executed on different platforms and specialized in dealing with different formats of information. The agent society is dynamically changing; new agents can be added into the system and old agents can be replaced by a newer version. This makes task scheduling and assignment more important and more difficult as well. Therefore, the administrator agent manages a register of agents and keeps a record of their capabilities and performances. Each agent registers its capability to the administrator when joining the system. Tests tasks are also submitted to the administrator. For each task, the administrator will send it to the most suitable agent. When an agent sends a message to the administrator, its intension must be made clear if it is to register their capabilities or to submit a test job quests, or to report the test result, etc. Such intensions are represented as 1 of the 7 illocutionary forces, which can be assertive, directive, commissive, prohibitive, declarative, or expressive.

The MAEST analysis

The system autonomy: Software testing consists of formatting the test plan, selecting the test items,

producing the test cases, executing the test and finally analyzing the test result. For the case of regression testing, regression test cases are selection for test execution. As we can see in Fig. 1, when a test tool is used, it is necessary for the tester to interfere in the test process. The sections (1) - (7) of Fig. 6a, which are carried out by the tester, are automated within tour system as we can see in Fig. 6b. Therefore by using our system, we can minimize the tester's interference and autonomously carry out the testing process. Whereby, the general testing tool passively executes testing. Our system actively executes testing through its autonomy. For this purpose, out system has control over the execution actions and the internal status transformations.

Time estimation: Our system reduces also the test time by intellectually selecting redundant free and consistent test cases from the massive amount of test cases generated from test case generation agents. In order to assess the effectiveness of our environment, we carried out two experiments to test a small example but have complex decision logic, the triangle program which accepts the lengths of three sides of a triangle and classifies it as scalene, isosceles and equilateral or not a triangle at all.

In the first one, we used three independents tools written (TCG1, TCG2 and TCG3) by us which respectively automate the random testing approach, all paths approach and equivalence partitioning approach

```
(ASSERTIVE
Receiver Administrator
Ontology testing ontology
(Content (CAPABILITY
(CONTEXT type ”unit_test”> )
(ACTIVITY type ”test_case_generation”> )
(METHOD type =”path_coverage”>)
( CAPABILITY _DATA type =”input”>
(ARTEFACT type =”object_under_test” FORMAT=”c”>) )
( CAPABILITY _DATA type =”OUTPUT”>
(ARTEFACT type =” test_suite” FORMAT=”list”) ) ) )
```

Fig. 5: KQML message

(The equivalents classes were manually calculated) to generate test cases. And in the second, we used our environment with three testing agents (using the same testing approaches as the three tolls in the first experiment). Table 1-4 indicate respectively the number of generated test cases and the necessary time to generate in the first experiment, the number of identical test cases generated between the various approaches, the number of test cases and the time estimation to execute them in the second experiment and the .time comparison in the two experiments.

Table 1: Time

USED APPROACH	NUMBERS OF TEST CASES GENERATED	UNITS OF TIME TO GENERATE THESE TEST CASES	UNITS OF TIME TO EXECUTE THESE TEST CASES
TCG 1	50	40sec	5 sec
TCG2	25	22 sec	2.5 sec
TCG3	10	2sec	1 sec

Table 2: common test cases

COMMON TEST CASES	NUMBERS OF TEST CASES
TCG 1 and TCG 2	20
TCG 1 and TCG 3	8
TCG 2 and TCG 3	7

Journal of Computer Science 3 (4): 249-258, 2007
ISSN 1549-3636

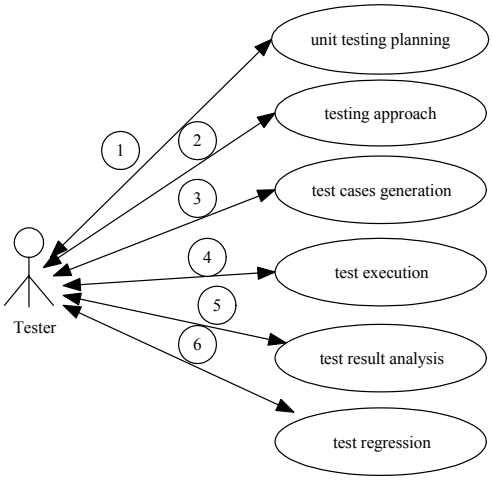

Fig. 6a: Classical testing process

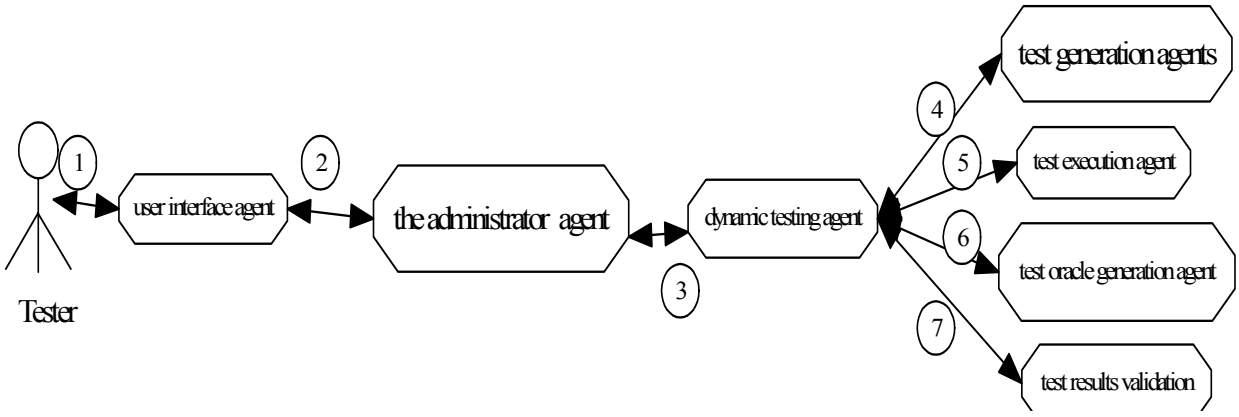

Fig. 6b: Our system process

Table 3: Number of test case and time estimation in our environment

Number of test cases	12
Time estimation	10, 2 sec

Table 4: Time

	NECESSARY TEST CASES TO TEST PROGRAM USING THE THREE APPROACHES	TIME TO GENERATE AND EXECUTE THE NECESSARY TEST CASE
Classical way	85	72.5 sec
Our system	12	10.2 sec

CONCLUSION

In this study, we proposed a multi-agent testing system where the all testing process can be executed on its own without the interference of tester. It also supports a test integration environment where testing can be executed gradually from unit test to system test.

In this system, the tester only has to concentrate on the high level goal, which is overseeing the test result, the detailed test procedures are carried out by our system's agents. In order words, these agents do all steps from the beginning to the end of the testing process (selecting test cases, executing testing…).

Our system has advantages in 4 aspects; first, it minimizes tester interference by executing the tests autonomously. Second, by intellectually selecting redundant free and consistent and effective test cases, the testing time is reduced while the fault detection ability increases. Third, the described architecture is open and extensible. It supports the dynamic addition

Corresponding Author: Ramdane Maamri, Lire laboratory, Computer Science Department, University of Mentouri Algeria

and retraction of agents and services. And finally, the agents can be in the same or different computers.

REFERENCES

1. Myers, G..J., 2004. The art of Software Testing. Willy 2nb.
2. Beizer, B., 1995. Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley: New York.
3. Ntafos, S., 1988. A comparison of some structural testing strategies. IEEE Trans. onSoft. Eng., 14: 868-874.
4. Rapps, S. and E.J. Weyuker, 1985. Selecting software test data using data flow information. IEEE Trans. Software Engg., 11: 367-375.
5. William, E.H., 1982. Weak mutation testing and completeness of test sets. IEEE Trans. on Software Eng.g, 8: 371-379.
6. Lesser, V., 1995. Multiagent Systems: An Emerging Subdiscipline of AI. ACM Computing Surveys, 27: 340-342.
7. Wooldridge, M. and N.R. Jennings, 1994. Agent Theories, Architectures and Languages: A Survey". Proceedings of the ECAI-94 Workshop on Agent Theories, Architectures and Languages, Amsterdam, The Netherlands, pp: 1-39.
8. Wooldridge, M., N. Jennings and D. Kinny, 2000. The GAIA methodology for agent oriented analysis and design. J. Autonomous Agents and Multi-Agent Systems, 3: 285-312.
9. Deloach, S.A., 1999. Multi-agent system engineering: a methodology and knowledge for designing agent systems. Proc. AOIS-1999.
10. Deloach, S.A., 2001. Analysis and Design using MaSE and agentTool. The 12th Midwest Artificial Intelligence and Cognitive Science Conference
11. Deloach, S.A., M.F. Wood and C.H. Sparkman, 2001. Multiagent Systems Engineering. Intl. J. Software Engineering and Knowledge Engineering. World Scientific Publishers, 11: 231-258.
12. Glaser, N., 1999. Contribution to knowledge modeling in a multi-agent framework (the COMOMAS approach), Phd thesis, Université Henry Poincaré, Nancy 1, France.
13. Iglesias Carlos A., Garijo Mercedes, C. Gonzalez José and R. Velasco Juan, 1998. Analysis and Design of Multiagent Systems using MAS-CommonKADS, Intelligent Agents IV (ATAL97), LNAI 1365, pp: 313-326, Springer-Verlag.
14. Neches, R. *et al*., 1991. Enabling Technology for Knowledge Sharing. AI Magazine, pp: 36-56.
15. Cao, J., X. Feng, J. Lu and S.K. Das, , Mailbox-based scheme for mobile agent communication. Computer.
16. Staab, S. and A. Maedche, 2001. Knowledge portals --- Ontology at work, AI Magazine, 21: 2.