http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet

Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 6 (1): 48-56, 2007
ISSN 1812-5638
© 2007 Asian Network for Scientific Information

Multi-Agent Plateform for Software Testing

Ramdane Maamri and Zaidi Sahnoun
Lire Laboratory, University Mentouri of Constantine, 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. Unless, we can find efficient ways to perform effective testing, the percentage of development cost devoted to the test process will increase significantly. In this study, we propose a multiagent 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.

Key words: Test cases, testing process, testing tools, multi-agent system, multi-agent methodology

INTRODUCTION

Testing is an important aspect of software development and plays a major role in detecting errors in implementations. Testing activities support quality assurance by gathering information about the nature of the software being studied. These activities consist of designing test cases, executing the software with those test cases and examining the results produced by those executions. Studies indicate that more than 50% of the cost of software development is devoted to testing, with the percentage for testing critical software being even higher. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to be of higher quality. Unless we can find efficient ways to perform effective testing, the percentage of development costs devoted to testing will increase significantly. Unfortunately, most automated test systems available today are too limited to effectively achieve all of the advantages of automating testing. 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 multiagent environment which can help during the whole testing process.

The functional architecture of the testing team: Figure 1 shows the general architecture of the test laboratory. 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 tester 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:

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

Testing techniques: The fact that the system is being executed distinguishes dynamic testing from static testing (Gaudel *et al*., 1996).

Static testing: There are three static testing approaches:

- Code inspection: in which source code is read and analyzed statically (usually by developers).

Corresponding Author: Ramdane Maamri, Lire Laboratory, University Mentouri of Constantine, Algeria

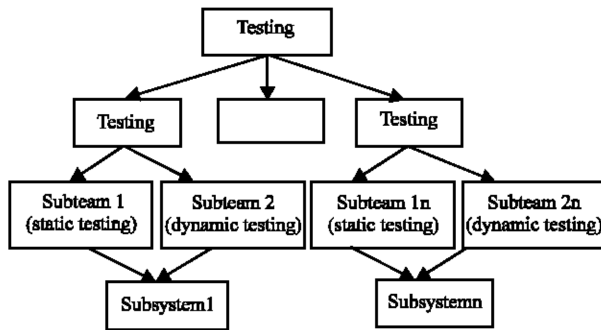

Fig. 1: The functional architecture of the testing

- Symbolic evaluation: In which a source code is used to compute symbolic expression for each output variable.
- Analysis of anomalies: in which a source code is used to find some kind of errors such as undefined variables, infinite loops, etc.

Dynamic testing: There are many different kinds of dynamic testing. In the first place, different aspects of system behaviour can be tested:

Functional tests: If the system has the intended functionality and complies with its functional specification.

Performance tests: If the system works as fast as required.

Robustness tests: Test the system reaction if its environment shows unexpected or strange behaviour.

Reliability tests: How long can we rely on the correct functioning of the system.

Moreover, testing can be applied at different levels of abstraction and for different levels of (sub-) systems: individual functions, modules, combinations of modules, subsystems and complete systems can all be tested.

A very common distinction is the one between black box and white box testing. In black box testing (functional testing) (Beizer, 1995), (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 tester. They generate test cases based on requirement and design information. While the white box testing (Structural techniques), (Ntafos, 1981), (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 (Rapps and Weyuker, 1985) and mutation testing (William, 1982). 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.

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

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.

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

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 systems: Multi-Agent systems (Lesser, 1995) have been identified as essential to the successful engineering of complex or large systems. A multiagent 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 (Wooldridge, 1994). One of their important aspects is modularity and flexibility. It is very easy to dynamically add or remove agents. Agent based architecture provides a natural method of decomposing large tasks into self contained modules, or conversely, of building a system to solve complex problems by a collection of agents, each of which is responsible for small part of the task. Agent-based systems can minimize centralized control. The design of

individual agents within a multiagent system has the advantage of being independent of the design of other agents as long as each agent follows an agreed upon protocol and ontology. This significantly contributes to the breakdown of complexity. Agents are thus viewed as black boxes whose operations are abstracted to the services they provide and which they announce to a manager agent.

Several methodologies have been proposed for the development of multi-agent systems. Most of them are either an extension of object-oriented methodologies: GAIA (Wooldridge, 2000), multi-Agent Software Engineering MaSE (Deloach, 1999; Deloach *et al*., 2001; Deloach, 2001), or an extension of knowledge-based methodologies: COMOMAS (Glaser, 1999), MAS-Common KADS (Iglesias Carlos *et al*., 1998).

OVERVIEW OF THE SYSTEM MAEST

Introduction: We used the notion of agent to specify 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 Goal Hierarchy Diagram of MAEST is shown in Fig. 2 and general architecture of the system is shown in Fig. 3.

Agents description: As shown in Fig. 3, the components in our testing environment are agents. Agents can dynamically join and leave the system to achieve the maximum flexibility and extendibility. 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 agent internal architecture: Each of the agents (Fig. 4) is implemented in Java and consists of a communication processor, planning processor, task processor and a local database. The communication processor deals with the exchange of messages with others agents. The task processor manages alls actions executed by the agent and returns the results to the

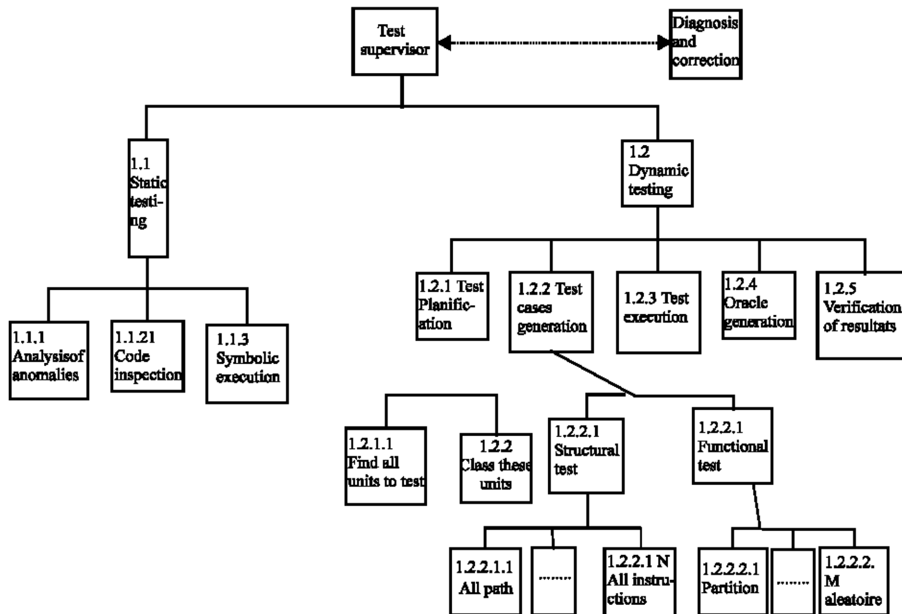

Fig. 2: Goal hierarchy diagram of MAEST

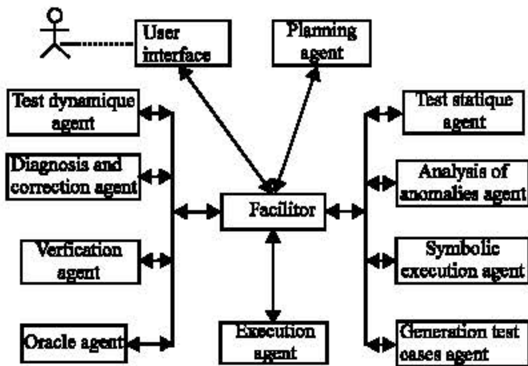

Fig. 3: System architecture

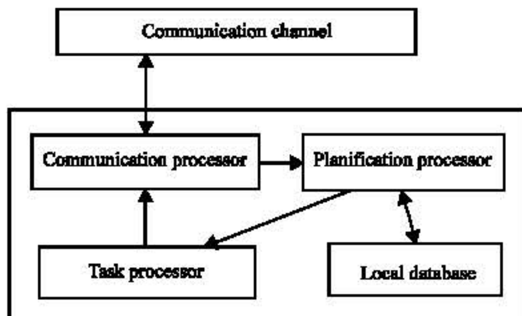

Fig. 4: Internal architecture of an agent

communication processor. The planning processor is responsible of preparing a plan of actions in order to perform the requested task.

The local database contains information about the agent's actions, its current work load, former successes/failures on tasks.

The facilitator agent: The main purpose of this agent is to coordinate the work of all agents in this environment. A facilator agent within this architecture complies with notions of facilitation as defined by Khedro and Genesereth (1995). Mainly, it supports routing of information, discovery of information, delayed and persistent notification of messages and communication management. When agents enter in the environment for the first time, they register with the facilator agent. They then advertise their services to the facilitator.

Interface agent: The interface agent serves as a link between the user and the system. It assists the users to express their requests by permitting the consultation of the base of requests as well as the list of keywords. It allows interaction between the users and results provided by different agents (i.e. pacification agent…)

Fig. 5: Input interface

Fig. 6: Result interface

The main role of this agent is to receive as shown in Fig. 5, from the user, program information such as:

- Program file
- Programming language
- Specification file
- Specification language

And at the end of the testing process, as shown in Fig. 6, this agent shows the final reports about the results of the testing process such as:

- The number of the tested units
- The passed units
- The failed units
- For each failed unit, which cases caused failure?

Program execution agents: To enable concurrent execution of multiples test cases, The planning 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.

Dynamic testing agent: The whole testing process should be well organized, well structured and well planned. The dynamic testing agent observes and controls all other dynamic testing activities. It involves developing a test strategy, the break-down into different activities and tasks, planning of all activities and tasks, allocation of time and resources, control, reporting and checking of each activity and task.

Static testing agent: The static testing agent observes and controls all other static testing activities. It involves developing a static test strategy, the break-down into different activities and tasks, planning of all activities and tasks, allocation of time and resources, control, reporting and checking of each activity and task.

Planning agent: The planning agent controls all the testing process. It involves planning the test of each unit in order to minimize drivers numbers and the testing time and also determines the units which can be tested in parallel.

To minimize the number of drivers and maximize the number of units that can be tested in parallel, the planning agent:

1 Classify the software units in classes:
   - C0 ={units that not require any calls}
   - Ci ={units that require only calls to units in prior classes Cj / j<i}
2 Construct a testing plan in which units in class Ci are tested in parallel and before those in class Cj (j>i)
3 If any units in class Ci are faulty, the will be replaced by drivers.

I.Symbolic evaluation agent: The main objective of the symbolic evaluation agent is to provide symbolic expression for each output variable.

The software architecture of the symbolic evaluation agent consists of a core implementing the functionality of the symbolic evaluation tool and the local user interface and a wrapper which gives the tool the behaviour and the appearance of an agent.

Analysis of anomalies agents: The main objective of the analysis of anomalies agents is to find out some kind of errors such as infinite loops, undefined variables, etc…

The software architecture of the analysis of anomalies agent consists of a core implementing the functionality of the analysis of anomalies tool and the local user interface and a wrapper which gives the tool the behaviour and the appearance of an agent.

Test oracle creation agents: 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 agent: The main objective of the test cases generation agent is to assist a tester in the generation of the best test cases for a software using one many testing approaches. It takes formally recorded specification and code information and then sends message asking for test cases. When it receives different test cases, it tries to take optimal test suite by eliminating redundant test cases. The test generation cases 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.

Testing agents: The main objective of each testing agent is to assist a tester in the generation of test cases for a software using one testing approach. It takes formally recorded specification and or code information, treats it as through it were a knowledge based or data base and applies test design rules to this base to automatically create test case. If requirement changes in the knowledge base, new test cases can be designed, generated, documented.

The software architecture of the testing cases generation agent reflects our desire to reuse the existing tools for test case generation. Each testing agent consists of a core implementing the functionality of test case generation tool and the local user interface and a wrapper which gives the tool the behaviour and the appearance of an agent. Our system is designed as an open system, it is relatively easy to add new testing agent with different core to the set of the testing agents.

Test results validation agent: The main work of the test results validation 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 it is not free, a new clone of this agent is created 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 AGENTS INTERCOMMUNICATION

Introduction: In our system, agents communicate with the Facilitor 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 informations are represented in an ontology (Neches *et al.*, 1991) 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 (Huo and Zhu, 2000). 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 m box is identified in the system with a different id. However, its location is transparent to the agents. Given an m box id, the agents can operate the m box without knowing its physical location. The m box can be opened by more than two agents at the same. For example, a facilator agent has an m box to receive task requests. Multiple agents can send message to this m box.

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 (Cao *et al.*, 2002]. It can be used as a means for agents to share knowledge, to transfer information and to negotiate their actions (Staab and Maedche, 2001). For this reason, we designed an ontology for software testing which takes in consideration the following concepts.

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.

In the following example shows a message send by a new agent to the facilitator 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

EXPRESSIVE

- **Receiver** *Facilitator*
- **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"))))

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 facilator agent manages a register of agents and keeps a record of their capabilities and performances. Each agent registers its capability to the facilator when joining the system. Tests tasks are also submitted to the facilator. For each task, the facilator will send it to the most suitable agent. When an agent sends a message to a facilator, its intention 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 intentions 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 shown 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. 7, which are carried out by the tester, are automated within tour system as shown in Fig. 7b.

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, w carried out two experiments to test a small program 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 (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 tools in the first experiment).

Table 1 indicates the number of generated test cases by each tool in the first experiment and the necessary time to generate and carry out these test cases.

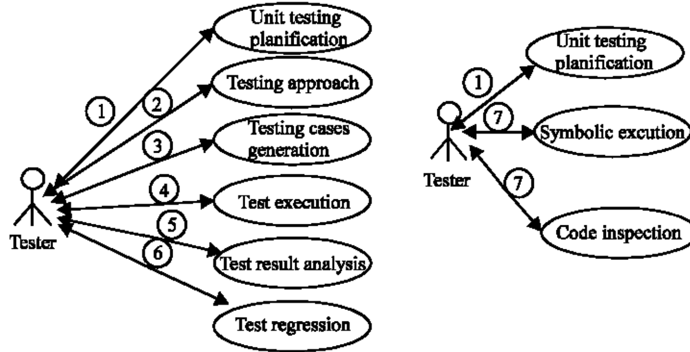

Fig. 7a: Classical testing process

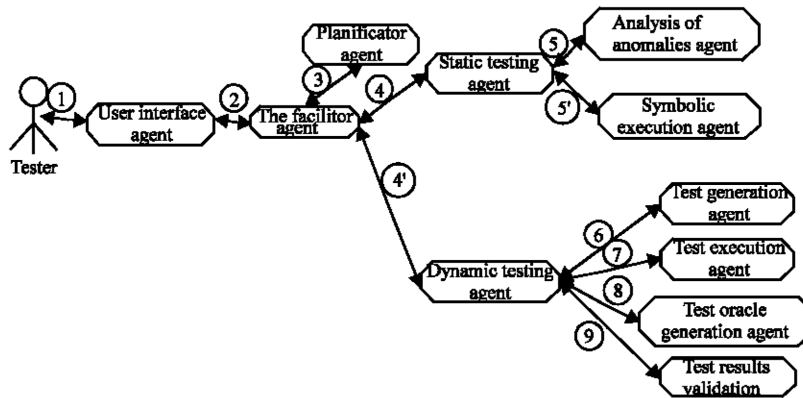

Fig. 7b: Our system process

Table 1: Time necessary to generate test cases

	Numbers of test cases generated	Units of time to generate these test cases (sec)	Units of time to execute these test cases (sec)
TCG 1	50	40	5
TCG2	25	22	2.5
TCG3	10	2	1

Table 2: Common test cases

	Numbers of identical test cases
TCG 1 and TCG 2	20
TCG 1 and TCG 3	8
TCG 2 and TGC 3	7

Table 3: Number of test cases and time estimation

Number of test cases	12.0
Time estimation	10.2 sec

Table 4: Time comparison to generate different cases

	Necessary test cases to test program using the three approaches	Time to generate and execute the necessary test case (sec)
First experience	85	72.5
Second experience	12	10.2

Table 2 indicates the number of identical test cases generated by the three tools and Table 3 indicates the number, the time estimation necessary to the generation and the execution of the test case necessary to test the program in the second experience.

Table 4 shows that the time necessary to generate and execution test cases is in general reduced in our system but in the case where the testing agents generate different test cases, the time will be worse then in the classical way.

CONCLUSIONS

In this study, we proposed a multiagent 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 and retraction of agents and services. And finally, the agents can be in the same or different computers.

REFERENCES

Beizer, B., 1995. Black-Box Testing: Techniques for Functional Testing of Software and Systems Wiley: New York,

Cao, J., X. Feng, J. Lu and S.K. Das, 2002. Mailbox-based scheme for mobile agent communication. Comp., pp: 54-60.

Deloach, S.A., 1999. Multi-agent system engineering: a methodology and knowledge for designing agent systems, Proceedings of AOIS.

Deloach, S.A., 2001. Analysis and Design using MaSE and agentTool. The 12th Midwest Artificial Intelligence and Cognitive Science Conference.

Deloach, S.A., M.F. Wood and C.H. Sparkman, 2001. Multiagent Systems Engineering. Intl. J. Software Eng. Knowledge Eng. World Scientific Publishers, 11: 231-258.

Gaudel, M.C., B. Mare, F. Schkienger and G. Bernot, 1999. Précis de génie logiciel Masson, 96 Glaser N., Contribution to knowledge modeling in a multi-agent framework (the COMOMAS approach), Ph. D Thesis, Université Henry Poincaré, Nancy 1, France.

Huo, Q. and H. Zhu, 2000. A message communication mechanism for mobile agents, Proc. of CACSCUK'2000, Loughborough, UK., Sep.

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, Springer-Verlag, pp: 313-326.

Khedro, T. and M.R. Genesereth, 1995. Facilitators: A Networked Computing Infrastructure for Distributed Software Interoperation, Workshop on AIin Distributed Information Networks, IJCAI-95, Montreal, 1995. Available from: http:/ /www. stanford.edu/group/CIFE/FCDA/ Lesser V, "Multiagent Systems: An Emerging Subdiscipline of AI". ACM Computing Surveys, 27: 340-342.

Neches, R. *et al*., 1991. Enabling technology for knowledge sharing. AI magazine. Winter issue, 1991. pp: 36-56.

Ntafos, S., 1988. A comparison of some structural testing strategies, IEEE Trans. onSoft. Eng., 14: 868-874.

Rapps, S. and E.J. Weyuker, 1985. Selecting software test data using data flow information IEEE Transactions on Software Engineering, 11: 367-375.

Staab, S. and A. Maedche, 2001. Knowledge portals-Ontology at work, AI Magazine, Vol. 21, summer.

William, E. Howden, 1982. Weak mutation testing and completeness of test sets, IEEE Transactions on Software Eng., 8: 371-379.

Wooldridge, M., 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.

Wooldridge, M., N. Jennings and D. Kinny, 2000. The GAIA methodology for agent oriented analysis and design, J. Autonomous Agents Multi-agent Systems 3: 285-312.