A Multi-Agent Software Environment for Testing Web-based Applications

Qingning Huo
Lanware Limited
68 South Lambeth Road, London SW8 1RL, UK
Email: Qingning.Huo@lanware.co.uk

Hong Zhu and Sue Greenwood
Dept of Comp., Oxford Brookes Univ.
Wheatley Campus, Oxford OX33 lHX, UK
Email: (hzhu | sgreenwood)@brookes.ac.uk

Abstract

This paper presents an agent-based software environment for testing web-based applications. The infrastructure of the system consists of a lightweight agent platform that supports agent communication, an ontology of software testing that enables flexible integration of multiple agents, and a formalism using XML to represent both the basic and compound concepts of the ontology. Relations between testing concepts are defined and their properties are analysed. A number of agents are implemented to perform various tasks in testing web-based applications. Broker agents use the ontology as a means of inferences to manage the knowledge about agents and assign each task to the most appropriate agent.

1. Introduction

The Internet and Web is becoming a distributed, hypermedia, autonomous and cooperative platform for software development, which stimulates much new progress in web-based application development [1]. A number of new features of web-based applications have been observed. For example, web-based applications often have an evolutionary lifecycle and rapidly updated. They often use a diversity of information representation formats and execution platforms. Their components can be developed using various techniques and written in different languages. They often operate in dynamic and open environments. As service based computing techniques becoming mature, they tend to be increasingly involved in collaboration with other information systems, e.g., by hyperlinks to out resource, by calls to web service providers, through software agents and so on. Moreover, it is common that they store and process such a vast volume of information that demands a network of computer systems to process and store. Finally, web-based applications usually have a large number of user in a diversity of user types.

Because of these properties, web-based applications are complex and difficult to develop and maintain. Although there is much established work in the validation and verification of traditional software [2], however, traditional testing methods and tools become inadequate for the web. First, the code and data are often mixed in a web-based application. Executable code can be embedded in data. On the other hand, information such as text, images and sounds can be presented, for example, through Java Applet programs. This requires software testing tools to bridge the gap between traditional dynamic testing and static analysis methods. Second, the diversity of information formats and execution platforms requires a software tool can support a wide range of platforms and information representation formats. It demands a flexible software environment to host and/or integrate a wide variety of tools for various platforms and languages. Moreover, the evolutionary lifecycle and incremental development of web-based applications and the ever emerge of new techniques require such an environment to be easily extended as well as to easily integrate third party tools and systems.

To meet these requirements, we proposed a multi-agent architecture of software environment [3]. Generally speaking, an agent is an active computational entity that has relatively complete functionality and cooperates with others to achieve its designed objectives. In our system, various software agents decompose testing tasks into small subtasks and carry out these tasks. They cooperate with each other to fulfil the whole testing task. This paper reports a prototype of such a system.

The paper is organised as follows. Section 2 gives the system's architecture. Section 3 presents the communication mechanism of the system. Section 4 presents an ontology of software testing and its uses in the integration of testing agents. Section 5 presents the collaboration mechanism of the system. Section 6 gives the details of the agents that perform various testing tasks for testing web-based applications. Section 7 concludes the paper with an analysis of the approach.

2. Overview of system structure

As shown in Figure 1, 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 specialised to deal with different information formats, executing on different platforms, using different testing methods or testing criteria, etc. These agents communicate to broker agents to submit and receive testing tasks. They may execute on different com-

puters and on different platforms in the system. In addition, they can be implemented in different programming languages. This makes the system flexible to integrate and extensible to improve functionality and performances.

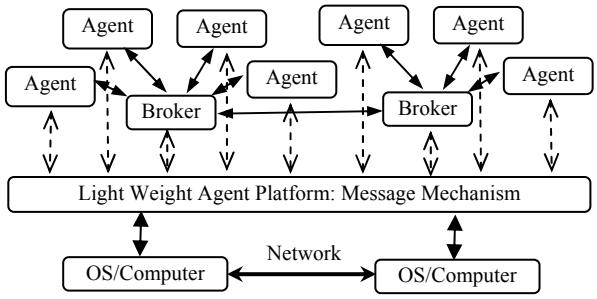

Figure 1 System structure

The key issue in the design of such a system is the co-operation between agents through an agent communication facility. In order to maximise interoperability and extendibility of the system, we divide the communication facility into the following three layers.

At the lowest level, agents communicate with each other through sending messages. Since agents may execute on different operating systems and they can join and leave the system dynamically, message passing between the agents must be supported by a platform built on top of operating systems. A message mechanism layer is implemented to support transmitting messages. This constitutes a light weight agent platform.

The middle level defines the contents of messages so that agents can communicate at an abstract and extendible language. The information contained in the messages can be classified into two types: (1) about testing tasks, which include requests of testing tasks to be performed, and reports of the results of testing activities; (b) about agents, 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 [4, 5, 6] about software testing.

The top level of the infrastructure is the communication and collaboration protocols, which defines the message formats and sequences for the collaborations between agents. We adopted the theory of speech-act [7, 8] to define the communication protocols. The following 3 sections discusses each level in more detail.

3. Message communication mechanism

The message mechanism consists of a set of communication primitives for message passing between agents [9]. Its design objectives are generally applicable, flexible, lightweight, scaleable and simple.

The communication mechanism is based on the concept of message box (mbox). An mbox is an unbounded buffer of messages. Unlike agents, an mbox never moves. All messages are sent to mboxes, and stay there until they are retrieved by agents. Our experiments shown that the mbox communication mechanism is effective and efficient in mobile agent environments. This is confirmed by other researchers in a recent theoretical study [10].

The mbox mechanism consists of 4 primitives.

- *Open*: to create a new mbox, or to fetch a handle of an existing mbox. Every mbox has a reference count. This count is incremented by one after an open operation.
- *Close*: to release a handle of an mbox after use. The reference count of the mbox is decreased by one for every invocation of close operation. The mbox is destroyed when its reference count reaches zero.
- *Send*: to send a message to an mbox. The message is kept in the mbox until its deletion condition is satisfied.
- *Receive*: to receive the next message from an mbox. If the mbox is empty, caller is put to sleep waiting for the next message; otherwise, the receive operation returns the message to the caller and changes its deletion condition. For example, for a read once mbox, one receive operation removes the first message from the mbox.

Each mbox is uniquely identified in the system with an id. It consists of a host id and a local id. However, its location is transparent to the agents. Given an mbox id, the agents can operate the mbox without knowing its physical location, nor need the agents to be on the same computer with the mbox. This allows agents to move freely without lost of communication contact and not to worry whether other agents are moving.

The mbox can be opened by more than two agents at the same time. Thus, in addition to simple 1-1 communication, it also supports 1-n, n-1 or n-n agent communication. For example, a broker agent has an mbox to receive task requests. Multiple agents can send message to this mbox. It is also possible to allow more than one broker agent to retrieve messages from the mbox. The change from one scenario to the other can happen at run time. It is also possible to terminate and restart an agent, either the same version, or an updated version, without lost of communication contact. All these are transparent to other agents engaged in the communication.

4. 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 [5,11]. It can be used as a means for agents to share knowledge, to transfer information and to negotiate their actions [12]. For this reason, we designed an ontology for software testing [13].

The most widely used approaches to ontology modelling include the Knowledge Interchange Format [14], description logic, UML [15], and recently, XML. XML has the advantages of being customisable, extensible, and

most importantly, suitable for web-based applications. The users can define the tags and formats to represent both simple concepts and complex structures. For these reasons, XML is also used in our system. However, the definitions of XML syntax are somehow not very readable. Therefore, in this paper, we use the well known extended BNF to define our ontology rather than DTD or XML schemes.

4.1 Taxonomy of testing concepts

We divide the concepts related to software testing into two groups: *basic concepts* and *compound concepts*. As shown in Figure 3, there are six types of basic concepts related to software testing, which include testers, testing context, activities, methods, resources, and environment. For each basic concept, there may be a number of sub-concepts. For example, a testing activity can be generation of test cases, verification of test results, measurement of test adequacy, etc. A basic concept may also be characterised by a number of properties, which are the parameters of the concept. For example, a software artefact is determined by (a) its format, such as JavaScript, (b) its type, such as the object under test, and (c) its creation and revision history, such as the version number.

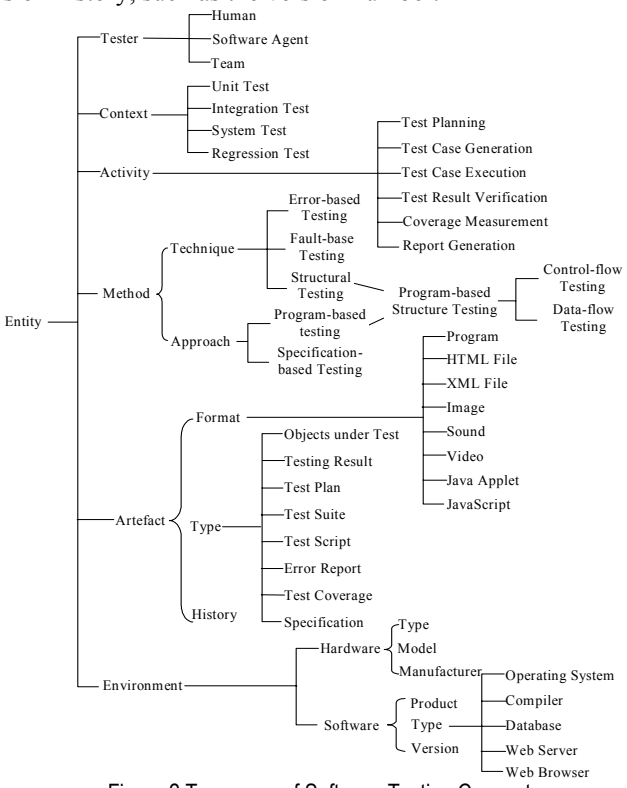

Figure 2 Taxonomy of Software Testing Concept

The following discusses each type of the basic concepts and defines their representations in XML.

(A) *Tester*. A tester refers to a particular party who carries out a testing activity. A tester can be a human being, a software tool, or a team, which consists of one or more testers. The *type* parameter of a tester indicates whether the tester is a human, a piece of software or a team. For the team type, the tester structure can contain a number of other tester structures, and the leader attribute gives the name of the leader of the team. The name parameter indicates the name or an identifier of the tester.

```
<tester> ::= "<" TESTER <tester_parameter> ">"
             { <tester> }  "</" TESTER ">"
<tester_parameter> ::=  TYPE "=" <tester_type>
    NAME "=" <identifier>    LEADER "=" <identifier>
<tester_type>::="HUMAN"|"SOFTWARE"|"TEAM"| …
```

where "**|…**" means that the syntax can be extended to include more tester types. The following are examples of a human tester named Joe and a team that consists of Joe and a software agent with Joe as the leader.

```
<TESTER TYPE="HUMAN" NAME="JOE"> </TESTER>
<TESTER TYPE="TEAM"
      NAME="ATEAM"
      LEADER="JOE">
  <TESTER TYPE="HUMAN" NAME="JOE">  </TESTER>
  <TESTER TYPE="SOFTWARE" NAME="ANAGENT">
      </TESTER>
</TESTER>
```

(B) *Context*. 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, regression testing, and so on.

```
<context> ::=  "<" CONTEXT <context_parameter> ">"
               "</" CONTEXT ">"
<context_parameter> ::= TYPE "=" <context_type>
<context_type>::="UNIT_TEST" | "SYSTEN_TEST"
    | "INTEGRATION_TEST"|"REGRESSION_TEST"
    | …
```

It is worth noting that XML is very flexible and easy to extend. The syntax given above is just for illustration. Other context types can be easily included into our implementation. This also applies to the definition of other concept in the sequel.

(C) *Activity*. 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 the sake of space, the BNF definitions of the syntax are omitted.

(D) *Method*. 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. [12].

(E) *Artefact*. 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. There are different types of objects under test, such as source code in programming languages, HTML files, XML files, embedded images, sound, video, Java applets, JavaScript, etc. Testing results include error reports, test coverage measurements, etc. Each artefact may also be associated with a history of creation and revision.
(F) *Environment*. 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.

4.2 Compound concepts

Compound concepts are defined on the bases of basic concepts, such as testing tasks and agent's capability.

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.

```
<capability> ::= "<" CAPABILITY ">"
        [ <context> ] <activity> <method>
        [ <environment> ] { <capability_data> }
    "</" CAPABILITY ">"
<capability_data> ::= "<" CAPABILITY_DATA
        TYPE "=" <capability_data_type> ">" <artefact>
    "</" CAPABILITY_DATA ">"
<capability_data_type> ::= "INPUT" | "OUTPUT"
```

In the following example of capability description, the agent is capable of doing node coverage test case generation in the context of system testing of hypertext applications represented in HTML.

```
<CAPABILITY>
  <CONTEXT TYPE="SYSTEM_TEST"> </CONTEXT>
  <ACTIVITY TYPE="TEST_CASE_GENERATION">
  </ACTIVITY>
  <METHOD TYPE="NODE_COVERAGE"></METHOD>
  < CAPABILITY _DATA TYPE="INPUT">
    <ARTEFACT
      TYPE="OBJECT_UNDER_TEST" FORMAT="HTML">
    </ARTEFACT>
  </ CAPABILITY _DATA>
  < CAPABILITY _DATA TYPE="OUTPUT">
    <ARTEFACT TYPE="TEST_SUITE"
      FORMAT="NODE_SEQUENCES">
    </ARTEFACT>
  </ CAPABILITY _DATA>
</CAPABILITY>
```

A testing task 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.

```
<task> ::= "<" TASK ">"
    [ <context> ] <activity> <method>
    [ <environment> ] { <task_data> }   "</" TASK ">"
<task_data> ::=
    "<" TASK_DATA TYPE "=" <task_data_type> ">"
        <artefact> "</" TASK_DATA ">"
<task_data_type> ::= "INPUT" | "OUTPUT"
```

However, not all combinations of basic concepts make sense. For example, the node coverage method cannot be combined with any media file types, such as images, sound or videos. A weakness of XML is that it provides very limited power to restrict such illegal combinations.

4.3 Relations between concepts

Relationships between concepts play a significant role in the management of testing activities in our multi-agent system. We identified a number of relationships between basic concepts as well as compound concepts. They are:

- Subsumption relation between testing methods.
- Compatibility relation between artefacts.
- Enhancement relation between environments.
- Inclusion relation between test activities.
- Temporal ordering between test activities.

These relations are all partial orderings. Based on these basic facts and knowledge, more complicated relations can be defined and used through inferences. The following are definitions of the most important ones.

(A) *MorePowerful relation on capability.* Let C represent the set of all capabilities. For all $c_1, c_2 \in C$, we say *MorePowerful*(c_1, c_2) if and only if all of the following statements are true.

- c_1 and c_2 have the same context, and
- c_1 and c_2 have the same activity, and
- The method of c_1 subsumes the method of c_2, and
- The environment of c_2 is an enhancement of the environment of c_1, and
- The input of c_2 is compatible with the input of c_1, and
- The output of c_1 is compatible with the output of c_2.

Informally, *MorePowerful*(c_1, c_2) means that a tester has capability c_1 implies that the tester can do all the tasks that can be done by a tester who has capability c_2.

(B) *Inclusion relation on test tasks.* Let T represent the set of all tasks. For all t_1 and $t_2 \in T$, we say *Include*(t_1, t_2), if and only if all of the following statements are true.

- t_1 and t_2 have the same context, and
- t_1 and t_2 have the same activity, and
- The method of t_1 subsumes the method of t_2, and
- The environment of t_2 is an enhancement of the environment of t_1, and
- The input of t_1 is compatible with the input of t_2, and
- The output of t_2 is compatible with the output of t_1.

Informally, *Include*(t_1, t_2) means that accomplishing task t_1 implies accomplishing task t_2.

(C) *Match between a task and a capability.* In the assignment of a testing task to a tester, a broker agent must an-

swer the question whether the job matches the capability of the tester. For any $c \in C$ and $t \in T$, we say *Match*(c, t), if and only if all of the following statements are true.

- c and t have the same context, and
- c and t have the same activity, and
- The method of c subsumes the method of t, and
- The environment of t is an enhancement of the environment of c, and
- The input of t is compatible with the input of c, and
- The output of c is compatible with the output of t.

Match(c, t) means that a tester with capability c can fulfil the task t. The following properties of the relations form the foundation of the inferences that the broker agent requires in the assignment of testing tasks.

$$MorePowerful(c_1, c_2) \wedge Match(c_2, t) \Rightarrow Match(c_1, t). \quad (1)$$

$$Include(t_1, t_2) \wedge Match(c, t_1) \Rightarrow Match(c, t_2). \quad (2)$$

5. Communication protocol

In our system, agents of similar functionalities may have different capabilities and are implemented with different algorithms, executing on different platforms and specialised 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, broker agents are implemented to negotiate with testing agents to assign and schedule testing activities. Each broker manages a registry of agents and keeps a record of their capabilities and performances. Each agent registers its capability to the brokers when joining the system. Tests tasks are also submitted to the brokers. For each task, the brokers will send it to the most suitable agent use the *Match* relation as a means of inferences.

When an agent sends a message to a broker, 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 [7,8], which can be assertive, directive, commissive, prohibitive, declarative, or expressive. We associate each message a speech-act parameter. Hence, messages have the following structure.

```
<message> ::=
    "<" MESSAGE ACT "=" <massage_act> ">"
        <message_para> "</" MESSAGE ">"
<massage_act> ::= "ASSERTIVE" | "DIRECTIVE"
    | "COMMISSIVE" | "PERMISSIVE"
    | "PROHIBITIVE" | "DECLARATIVE"
    | "EXPRESSIVE"
<message_para> ::= <capability> | <task> | <answer>
<answer> ::=
    "<" ANSWER STATUS "=" <answer_status>
        [ REASON "=" <identifier> ] ">"
        [ <job> ] [ <artefact> ] "</" ANSWER ">"
<answer_status> ::= "SUCCUSS" | "FAIL"
```

Example 1. The following is a sequence of messages between agents A_1 and A_2 and a broker B.

(1) Agent A_1 sends an ASSERTIVE message with a <*capability*> parameter to the broker B. This means that A_1 wants to register to the broker B and claims its capability.

(2) Agent A_2 sends an EXPRESSIVE message to the broker B, with a <*task*> parameter describing a testing task. This means that the agent wants to find some agent to perform the testing task.

(3) The broker B searches its knowledge about registered agents, and finds that agent A_1 is the best match for the task. It then sends a DIRECTIVE message with the <*task*> parameter to agent A_1.

(4) When agent A_1 finishes the task, it sends an ASSERTIVE message with an <*answer*> parameter to the broker. The <*answer*> parameter describes the status of the task and output of the task if it is successful, or the reason of failure or error messages if it is not successful.

(5) The broker B may forward the message to agent A_2, or try to find another agent to carry out the testing task in case the output of agent A_1 is not successful. □

6. Test agents for web applications

As shown in Figure 3, the testing environment consists of a number of agents to fulfil testing tasks for web-based applications. These agents can be distributed to different computers, for example, as in Figure 4, on a media server, a test server, and a client. In fact, agents can be freely distributed according to any specific configuration. They can also be mobile and change their location at runtime.

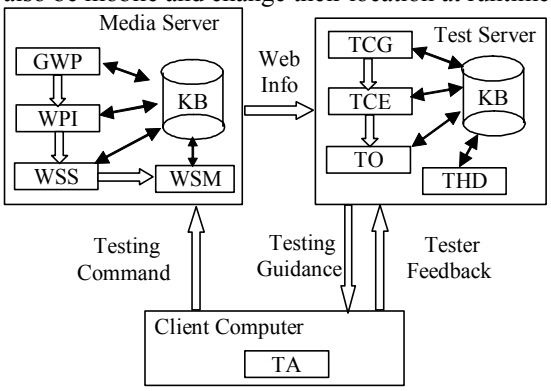

Figure 3 Agents for Testing Web-Based Applications

The following agents have been implemented for testing web-based applications.

Get Web Page (GWP) agents retrieve web pages from a web site.

Web Page Information (WPI) agents analyse the source code of a web page, and extract useful information. It also stores the structure information in a knowledge base.

Web Site Structure (WSS) agents analyse the hyperlink structure of a web site, and generate a node-link-graph describing the structure. This structure is also stored in a knowledge base for other agents to use.

Test Case Generator (TCG) agents generate test cases to

test a web site according to certain testing criteria. Currently, three agents are implemented for node coverage, link coverage and linear independent path coverage criteria, respectively. See [16] for their definitions.

Test Case Executors (TCE) agents execute the test cases, and generate execution results. There are two ways of test case execution. One is to run the test cases interactively in front of the human tester, with the aid of a testing assistant. The other is to playback a recorded test sequence.

Test harness and driver (THD) agents provide flexible interfaces to unit test tools. They play the traditional role of test harness, test driver and module stubs. They enable the integration of various testing tools seamlessly into the multi-agent systems so that components written in different languages can be tested in a unified environment. Some of these agents work on the servers and directly calls the modules; some execute on client side and indirectly calls the modules by sending cgi command (URL).

Test Oracles (TO) agents verify whether the testing results match a given software specification. Different types of information require different kinds of oracles. Some simply compare with the results of previous tests. Some examine the results to make sure it fits into certain patterns. These patterns can be generated automatically or defined by the software engineers.

Testing Assistants (TA) are user interface agents that guide human testers in the process of testing. It helps to get test requirements from the human users, send messages to TCG to generate test cases, present test cases to the user, allow the user to click through links to test each web page, allow the user to enter information of tested pages, record testing history and generating testing report.

WSM (Web Site Monitor) agents monitor the web site and generate testing tasks when changes in the web site is detected.

7. Conclusion

This paper presented an application of agent technology in the testing of web-based applications. A prototype is described and discussed. Its multi-agent architecture and the infrastructure of the system satisfy the requirements of testing web-based applications. It clearly demonstrated that agent techniques are suitable for testing web-based systems. In particular, first, the dynamic nature of web information systems demands constant monitoring the changes of the system and its environment. Sometimes, the change in the system and its environment may require changes in testing strategy and method accordingly. Agents are adaptive, and they can adjust their behaviours based on environment changes. These changes can be integrated to the system lively. Second, the diversity of platforms and the formats of media and data of web information systems demand using a wide variety of test methods and tools. Multi-agent systems can provide a promising solution to this problem. Different agents are built to handle different types of information, to implement different testing methods and to integrate different tools. Thus, each individual agent can be relatively simple while the whole system is powerful and extendible. Third, the distribution of large volume of information and functionality over a large geographic area requires testing tasks carried out at different geographic locations and to transfer information between computer systems. The agents can be distributed among many computers to reduce the communication traffic. Although a single agent can only perform as a normal program, the strength of agents come from the intelligent dynamic task assignment, the dynamic control of agent pool, the dynamic interactions between agents, the live update of agent systems.

References

[1] Crowder, R., Wills, G., and Hall, W., Hypermedia information management: A new paradigm, Proc. of 3rd Int. Conf. on Management Innovation in Manufacture, pp329-334, 1998.

[2] Zhu, H., Hall, P., May, J., Software Unit Test Coverage and Adequacy, ACM Comp. Survey 29(4), pp366~427, 1997.

[3] Zhu, H., Greenwood, S., Huo, Q. and Zhang, Y., Towards agent-oriented quality management of information systems, in Workshop Notes of 2nd International Bi-Conference Workshop on Agent-Oriented Information Systems at AAAI'2000, Austin, USA, July 30, 2000, pp57-64.

[4] Neches, R. *et al*., Enabling Technology for Knowledge Sharing. AI Magazine. Winter issue, 1991. pp36-56.

[5] Uschold, M. and Gruninger M, Ontologies: Principles, Methods, and Applications. Knowledge Engineering Review 11(2), June 1996.

[6] Gruber, T., A translation Approach to portable ontology specifications. Knowledge Acquisition 5, pp199-200, 1993.

[7] Singh, M.P., A semantics for speech acts, Annals of Mathematical and Artificial Intelligence 8(II), pp 47-71, 1993.

[8] Singh, M. P., Agent communication languages: Rethinking the principles, IEEE Computer, Dec 1998, pp40-47.

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

[10] Cao J., Feng, X., Lu, J., and Das, S. K., Mailbox-based scheme for mobile agent communication. Computer, September 2002, pp54-60.

[11] Staab, S. and Maedche, A., Knowledge portals --- Ontology at work, AI Magazine, 21(2), Summer 2001.

[12] Fox, M. S., and Gruninger, M., Ontologies for Enterprise Integration, Proc. of the 2nd Conf. on Cooperative Information Systems, Toronto, 1994.

[13]Huo, Q., Zhu, H., Greenwood, S., Using Ontology in Agent-based Web Testing, Proc. of ICIIT'2002, Beijing, China.

[14] National Committee for Information Technology Standards, Draft proposed American national standard for Knowledge Interchange Format. http://logic.stanford.edu/kif/dpans.html

[15] Cranefield, S., Haustein, S., Purvis M., UML-Based Ontology Modelling for Software Agents, Proc. of Ontologies in Agent Systems Workshop, Agents 2001, Montreal, pp21-28.

[16] Jin, L., Zhu, H., and Hall, P., Adequate testing of hypertext applications, Information and Software Technology, 39(4), pp225-234, 1997.