9

Agent Systems in Software Engineering

Vasilios S. Lazarou[1], Spyridon K. Gardikiotis[2] and Nicos Malevris[2]
[1]National & Kapodistrian University of Athens
[2]Athens University of Economics and Business
Greece

1. Introduction

During the last decade the continuous growth of the Web resulted in a significant development shift from simple types of software applications to distributed multi-tier web-based applications. In general, distributed systems are by nature more complex than centralized systems. As a result, the software engineering tasks of these systems are also complicated.

Unlike traditional software applications, Web-based applications are associated with a plethora of special characteristics that impede the appliance of conventional software engineering techniques. Among them, the most important include the distributed and stateless nature of the Web, the impressively high changing frequency of implementation technologies and the spread of dynamic Web pages. Furthermore, the vital role of databases in both web and distributed applications raises a demand for introducing software engineering techniques tailored for these applications. These applications, known as database applications (DA), contain embedded SQL statements in the source code. Similarly to web applications, the presence of such special statements turns out to impose a number of limitations to the applicability of existing software engineering techniques while also originating new issues.

In this chapter, the use of agent technology to confront with the software engineering task will be illustrated. More precisely, the focus will be on the application of agent systems in order to confront with the requirements of the software engineering process for distributed software systems in general, paying particular attention to distributed database applications and web applications.

Software agents can be described as intelligent and autonomous software entities that have the ability to exhibit proactive behaviour and to collaborate with each other. The software engineering process can be greatly enhanced by utilising agent technology and adopting the architecture of an intelligent, flexible and extensible agent system. The multi-tier architecture of most distributed applications offers a suitable foundation because of its inherent complication that highlights the significant and novel contribution of a multi-agent architecture.

The rationale behind utilizing agent technology has to do with the interoperability of the software resources belonging to potentially disparate application components and disparate domains. Towards this direction, agents offer a unified platform of interaction through agent communication.

The application of agent technology for the software engineering task is certainly a new and promising research area. However, a variety of approaches that attempt to exploit the

benefits of agent technology have already made their appearance and it is expected that this tendency will further evolve. At this point, it needs to be clarified that the chapter will not focus on the research area that deals with the employment of software engineering technology for agent systems. Although similar in title, this research area deals with applying software engineering methodologies to assist the creation of multi-agent systems; something completely different.

The first one has as a goal to provide an agent infrastructure to support software testing. This is realised by suggesting multi-agent frameworks that can be used as a model to build agent systems for testing service-oriented web applications. This research track aims at presenting an agent system for tackling the issues of software maintenance and testing of distributed applications.

Illustrating the research attempts that employ software agents on software engineering tasks, they can be categorised according to two key target levels. The first one has an infrastructural target. Some research work focuses on presenting communication and coordination infrastructures for agents engaged in web software testing. Another research direction targets the creation of a multi-agent framework for software testing but the goal is on how an agent infrastructural framework can assist the job of constructing concrete agents systems for service-oriented applications.

The second one has a more applied target. As a representative work, research in which multi-agent system architectures are used in software testing of web-based applications can be mentioned. Moreover, there is ongoing research where an agent system is being utilised for the software engineering of distributed database applications. The first primary objective is to assess the maintainability and to facilitate the maintenance of such applications in the presence of changes on the schema of the underlying database. The second primary objective is to support another major software engineering task namely structural and regression software testing.

The remainder of this chapter is organised as follows. Section 2 outlines the fundamental background scientific areas of Agent Systems and Software Engineering. Section 3 introduces the first primary research direction where agent frameworks are used in software engineering. Section 4 continues the illustration covering the second primary research direction where multi-agent systems are used in software engineering. Section 5 is about Agent-Oriented Software Engineering and gives a brief description of the opposite view where the idea of an agent is being utilised as a generic software engineering model. Finally, section 6 concludes the chapter by offering an overall analysis of the current research status by highlighting the commonalities and the differences of the above research approaches, in a form of comparative evaluation, and providing a view of the scope of the current approaches and potential future research courses of action.

2. Background concepts

In this section, the background concepts relevant to the chapter are going to be illustrated. However, besides the primary concepts of Software Engineering and Agent Systems, some special topics within the research area of Software Engineering, namely Web-based Software Systems and Service-Oriented Systems, will be particularly described. The reason is that a significant amount of research that applies agent system technology to software engineering has been evolved around these topics. This section concludes by describing the current convergence of the two main concepts of this chapter.

2.1 Software engineering

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE, 1990). The discipline of software engineering includes knowledge, tools, and methods for software requirements, software design, software construction, software testing, and software maintenance tasks (SWEBOK, 2004). Among them, the interest is focused on the processes of software engineering that can be performed by fully automated computing techniques. In particular, such processes are software testing and software maintenance.

Software testing is the process used to assess the quality of computer software. Towards this direction, two objectives are usually identified: the verification and validation of the software. Software verification examines the way that the software is built and verifies that this matches its specifications. Software validation examines the derived software and validates that this product matches the customer requirements. In practice, software testing accomplishes its intended scope by revealing the amount of embedded software faults. Its results guide the software engineering process to reduce the amount of these faults ending up in an acceptable defect rate according to the specific software's nature. Software testing techniques are traditionally divided into *black box* and *white box* techniques. The former type treats the software as a black-box without any understanding of internal behaviour and aims to test its functionality according to its requirements. Examples of black box testing techniques include random testing, equivalence partitioning, boundary value analysis, model-based testing etc. The latter type of testing presumes that the tester has access to the source code of the software and derives tests that satisfy some code coverage or data adequacy criteria. Examples of such criteria include control flow based criteria (e.g. path, branch and statement coverage), text-based adequacy criteria (e.g. LCSAJ) and data flow criteria (e.g. definitions, uses, predicate uses, computational uses etc.).

Software maintenance is the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (IEEE, 2004). Thus, software maintenance includes a number of both pre-delivery and post-delivery processes, which according to (IEEE, 1996) are summarized to the following: process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration and retirement. The maintenance processes can further be classified into categories. Among many alternative suggestions, the (ISO, 2006) proposes the four major categories of software maintenance:

- Corrective maintenance is the reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance is the modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance is the modification of a software product after delivery to improve performance or maintainability.

Preventive maintenance is the modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

2.1.1 Service oriented architecture

Service oriented computing (SOC) is an emerging cross-disciplinary paradigm for distributed computing that is changing the way software applications are designed, architected, delivered and consumed (Erl, 2005). Service Oriented Architecture (SOA) is a form of distributed system architecture; its properties are consolidated by the W3C working

group of web service architecture. Services are autonomous and platform-independent computational elements that can be used to build networks of collaborating applications distributed within and across organizational boundaries.

Service-Oriented Architecture (SOA) and its Web implementation Web Services (WS) promote an open standard-based and loosely coupled architecture for integrating applications in a distributed heterogeneous environment. Such applications are characterized by service orientation, task distribution, collaboration among development parties, run-time behaviour and open standards for interfacing among their components.

Service dependability is critical for establishing a trustworthy service-oriented computing environment. However, the paradigm shift from product-oriented software development to SOA and WS brings many new issues to traditional verification and validation techniques. In SOA, an application is created by dynamically discovering, binding to, and integrating the discovered services from the Internet, possibly created by third party service providers. Due to the open standards and open platform, a large number of services satisfying the same requirements can co-exist, and new services can be published at any time. Hence, during system evolution, the application can dynamically rebind to different services and the architecture can be reconfigured at runtime. The dynamic and collaborative nature of SOA brings new challenges to testing WS applications including system complexity due to the flexibility of system configuration, interoperability among third-party developed components, runtime fault detection and reliability evaluation, dynamic re-composition, and implementation transparency.

2.1.2 Web application testing

A Web application can be considered as a distributed system, with a client-server or multi-tier architecture, including the following main characteristics:

1.   A wide number of users distributed all over the world and accessing it concurrently.
2.   Heterogeneous execution environments composed of different hardware, network connections, operating systems, Web servers and Web browsers.
3.   An extremely heterogeneous nature that depends on the large variety of software components that it usually includes. These components can be constructed of different technologies (i.e., different programming languages and models), and can be of different natures (i.e., new components generated from scratch, legacy ones, hypermedia components).
4.   The ability of generating software components at run time according to user inputs and server status.

Web applications are difficult to understand and test due to lack of abstraction, highly unstructured, heterogeneous representation, mixture of presentation and application logic and dynamic page generation. Web applications testing need to address challenges introduced by new control structures like hyperlinks (navigation, request and redirection), new data flow issues (e.g., scripts that are not compiler checked, HTML/XML documents as variables, storing data as hidden elements, JSP tags-defined variables and parameters and passing data via HTTP hyperlinks) and new dynamic behaviour like navigation behaviour and Web state behaviour.

In (Di Lucca & Fasolino, 2006), they considered testing of the functional requirements with respect to four main aspects, i.e., testing scopes, test models, test strategies, and testing tools. More specifically, testing strategies define the approaches for designing test cases. They can

be responsibility based (also known as black box), implementation based (or white box), or hybrid (also known as grey box). In (Nguyen, 2000) it is said that 'Gray-box testing is well suited for Web application testing because it factors in high-level design, environment, and interoperability conditions. It will reveal problems that are not as easily considered by a black-box or white-box analysis, especially problems of end-to-end information flow and distributed hardware/software system configuration and compatibility. Context-specific errors that are germane to Web systems are commonly uncovered in this process.

2.2 Agents and multi-agent systems

Agents and multi-agent systems (MAS) have recently emerged as a powerful technology to face the complexity of a variety of modern Information Systems (Zambonelli & Omicini, 2004). For instance, several industrial experiences already testify to the advantages of using agents in Web services and Web-based computational markets and distributed network management. In addition, several studies advise on the possibility of exploiting agents and MAS as enabling technologies for a variety of future scenarios, i.e., pervasive computing, grid computing and semantic web.

The core concept of agent-based computing is, of course, that of an agent. However, the definition of an agent comes along with a further set of relevant agent-specific concepts and abstractions. Generally speaking, an agent can be viewed as a software entity with the following characteristics (Jennings, 2001):

- Autonomous: an agent is not passively subject to a global, external flow of control in its actions. That is, an agent has its own internal execution activity (whether a Java thread or some other sort of goal-driven intelligent engine, this is irrelevant in this context), and it is pro-actively oriented to the achievement of a specific task on user's behalf.

- Situated: an agent performs its actions while situated in a particular environment, whether a computational (e.g., a Web site) or a physical one (e.g., a manufacturing pipeline), and it is able to sense and affect (portions of) such an environment in order to meet its design objectives.

- Social: in the majority of cases, agents work in open operational environments hosting the execution of a multiplicity of agents, possibly belonging to different stakeholders (think, e.g., of agent-mediated marketplaces). In these MAS, the global behaviour derives from the interactions among the constituent agents. In fact, agents may communicate/coordinate with each other (in a dynamic way and possibly according to high-level languages and protocols) either to achieve a common objective or because this is necessary for them to achieve their own objectives.

It is clear that an agent system cannot be simply reduced to a group of interacting agents. Instead, the complete modelling of the system requires explicitly focusing also on the environment in which the MAS and its constituent agents are situated and on the society that a group of interacting agents give rise to. Modelling the environment implies identifying its basic features, the resources that can be found in the environment, and the way via which agents can interact with it. Modelling agent societies implies identifying the overall rules that should drive the expected evolution of the MAS and the various roles that agents can play in such a society (Zambonelli et al., 2003).

2.3 Agent systems and software engineering

The emergent general understanding is that agent systems, more than an effective technology, represent indeed a novel general-purpose paradigm for software development.

Agent-based computing promotes designing and developing applications in terms of autonomous software entities (agents), situated in an environment, and that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages.

These features are well suited to tackle the complexity of developing software in modern scenarios since:

1. The autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of system modularity and encapsulation;
2. The flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable scenarios where software is expected to operate (Zambonelli et al., 2001);
3. The concept of agency provides for a unified view of artificial intelligence (AI) results and achievements, by making agents and MAS act as sound and manageable repositories of intelligent behaviours (Russel & Norvig, 2003).

2.3.1 Agent systems and web systems

Engineering distributed systems is a challenging task due to issues such as concurrency, fault tolerance, security and interoperability (Sommerville, 2004; Tsai et al., 2003). With respect to engineering web service systems, applying agent techniques to service orientation field has proven a natural choice. The research on agent-based applications has so far demonstrated that agents can glue together independently developed legacy systems. The control of a system can be distributed among autonomous agents and still maintain global coherence. Moreover, system's capability improves greatly when systems (represented by agents) cooperate.

Therefore, applying the MAS technique in WS has been a focus of WS research, such as service discovery, selection, and orchestration (Buhler & Vidal, 2003; Maamar et al., 2005; Richards et al., 2003; Sycara et al., 2001).

However, agents correspond to a broader concept with respect to services. In (Qi et al., 2005), the notion of agent-based web services (AWS) is proposed, including architecture and meta-model and integration. The key challenge is to develop an integration framework for the two paradigms, agent- and service-oriented, in a way that capitalizes on their individual strengths.

3. Agent infrastructures in software engineering

In this section, the research work relevant to defining agent infrastructural frameworks will be covered. This work targets distributed software systems in general but also web services and web-based applications in particular.

3.1 A multi-agent framework for testing distributed systems

In (Yamany et al., 2006), a design for testing distributed systems is proposed. They use a three-tier distributed system structure consisting of a server, middleware and multiple clients. The server contains the data repository of the distributed application, whereas the middleware is considered to be the software bus associated with those clients.

Agents in the proposed multi-agent architecture consist typically of two generic types: social (immobile) agents and mobile agents. Social agents are used to monitor the three-tier architecture of these distributed systems (i.e. server, middleware and clients) and to execute various scheduled testing types such as unit testing and integration testing. Moreover, mobile agents are used to carry out an urgent testing such as regression testing specified by a tester (i.e. human or an agent). In addition to that, the proposed framework monitors the user usage in order to increase the leverage of the testing process by increasing the chances to discover most of the defects that might appear in both the server and clients sides.

The framework consists of three levels of autonomous and adaptive agents. The first level of agents is on the server side. Basically, it is a single agent that monitors the data of the distributed application and is called the Database Repository Agent (DRA). The second one – Middleware Controller Agent (MCA) – is located at the middleware and is the kernel of the proposed framework. Its main goals are to investigate the middleware behaviour, collect the return feedback from the clients and make an integrated report about the system. Finally, a group of social agents is distributed over the available clients. Each one is named Client Checker Agent (CCA) and is responsible for unit testing.

The framework can be extended to execute more testing procedures at the request of the tester. In some crucial unexpected behaviour of a distributed system, the tester can ask for further testing and this can be done by sending a supportive mobile agent that could help in that mission. This agent's name is Mobile Urgent Agent (MUA).

3.2 Agent fabric for web services

In (Ma et al., 2007), MAS concepts are applied for service autonomy architecture. Service agents have three basic responsibilities. They maintain runtime operations, manage service lifecycle and control trusty communication. The first one is supported by a set of basic functions, such as service discovery, monitoring and composition. The second one is an advanced feature requiring comprehensive service modelling and governance. Agent system trustworthy is also an important issue for agent collaboration.

On the other hand, an effective communication mechanism is very important for an agent system, because autonomous systems do not stand alone without interaction with other parties. This is where fabric comes into place. Fabric in SOA context usually means a messaging environment or communication infrastructure, which makes services or applications integrated. In (Ma et al., 2007), they propose a lightweight agent fabric to serve the communications between autonomous service agents and, furthermore, cross-enterprise applications. According to the aforementioned autonomous system design requirements, XMPP (Saint-Andre, 2005) is employed as the underlying communication and message routing technology to build this kind of lightweight fabric for agents. The existing XMPP technologies are also leveraged for the trusty communication between agents.

3.3 Agent framework for web services

In (Bai et al., 2006), to address the challenges of collaborative and dynamic service-oriented testing, they present a multi-agent framework (called MAST) for testing services with agent-based technology. It is based on (Tsai et al., 2003) to facilitate web service (WS) testing in a coordinated and distributed environment. Test agents are classified into different roles which communicate through XML-based agent test protocols.

The key features of MAST are:

- Testing is decomposed into different tasks including WS specification-based test generation, centralized test planning, distributed test execution, test monitoring, and test result synthesis and analysis. Different agent types are defined to accomplish various tasks.
- Test agents are organized into groups. Each group is responsible for the execution of a test plan and is composed of a group of test runners and monitors, which are coordinated by a test coordinator.
- The mechanism is defined to dynamically generate, organize, coordinate, and monitor test agents so that testing can be adaptive to reconfiguration and re-composition of services.
- A rule-based strategy is introduced to facilitate interactively define, update, and query rules for test planning and agent coordination.

Through the monitoring and coordinating mechanism, the agents can re-adjust the test plan and their behaviour at run-time to be adaptive to the changing environment. The major testing process is decomposed into three parallel and iterative phases:

1. Test script generation to define the test cases and test scenarios;
2. Test scheduling to create and allocate the test plan to agent groups;
3. Test run to exercise the test scripts, monitor execution status, and collect results.

Service specification provides basic information of the services under test such as service interface and service flow. Rule management provides the knowledge for test scheduling. Test analysis analyzes the test data such as failure rate to evaluate the quality of services and test effectiveness. MAST supports the generic testing process and classifies the agents into seven types explained as follows:

Test Master accepts test cases from Test Generator, generates test plans and distributes them to various test groups. A set of test agents that implement a test plan are organized into a test group, which is coordinated by a Test Coordinator. Test Runners execute the test scripts, collect test results and forwards the results to Test Analyzer for quality and reliability analysis. The status of the test agents are monitored by the Test Monitor.

3.4 Agent coordination model for web services

In (Xu et al., 2006), the MAST framework (see 3.2) is utilised to propose a coordination architecture based on the reactive tuple space technique to facilitate dynamic task assignment, agent creation and destruction, agent communication, agent distribution and mobility, and the synchronization and distribution of collaborative test actions. Tuple space defines a shared memory mechanism among agents by which data are structured organized, described by tuples and retrieved by pattern matching. Adding reactivity to the tuple space means the space can have its own state and react to specific agent actions. It is a hybrid approach which combines control-driven and data-driven coordination models.

In this research, two tuple spaces are defined in MAST to manage the coordination channels and to facilitate data sharing and asynchronous coordination among test agents. Through the task tuple space, test tasks are dynamically allocated to different types of test agents according to the process defined in the scheduling. Through the result tuple space, the execution results are communicated from agents to agents. A subscription mechanism is introduced to associate programmable reactions to the events occurred and state changes on the tuple space.

3.5 An agent-based framework for testing web applications

In (Kung, 2004), an agent based framework for Web applications testing is presented. The framework is based on the BDI formalism (Rao & Georgeff, 1995) and the Unified Modelling Language (UML). The BDI architecture associates beliefs, desires and intentions with agents. Beliefs are the agents' observation about the environment and other agents. Desires are goals to be accomplished. Intentions are action plans to achieve goals. Using this framework, Web testing models and other testing objects like knowledge of the component under test (CUT) and test results are modelled as beliefs, test criteria as goals, and test activities as action plans.

The framework defines a number of abstract classes for modelling agent-oriented systems. Application specific agent types are derived from these classes to inherit model-defined features and relationships, and implement inherited abstract features. In this way, the framework enforces the BDI model but also accommodate for application specific behaviours. The abstract classes include: *Belief, Goal, Plan, Agent, Agent Communication Act*, and *Blackboard* (Kavi et al., 2003; Kung et al., 2003).

The framework also introduces a number of new diagrams: **Agent Goal Diagram (AGD)** depicts the relationships between the goals and the environment and defines the roles of agents. **Use Case Goal Diagram (UCGD)** combines the UML Use Case Diagram (UCD) and the AGD to show which use cases affect which goals and vice versa. This provides a high level guidance to Agent Sequence Diagram (ASD) construction. **Agent Domain Model (ADM)** represents the domain knowledge that is internal to an agent, including the definitions of the agent's Beliefs, Goals and Plans and their intrinsic relationships. **Agent Sequence Diagram (ASD)** depicts interactions among the beliefs, goals, plans and other objects of an agent and is a refinement of an agent. These diagrams model the behaviour of a test agent. Other diagrams introduced are the Agent Design Diagram (ADD), to document the design of an agent, and the Agent Activity Diagram (AAD) and Agent State-chart Diagram (ASCD), to model the internal activity and information flows and the internal state behaviours of agents.

3.5.1 Web application test agents

There are various types of test agents for testing the various types of Web documents. A Web application test agent is composed of the various types of a test agent. Since each type of Web document has several categories of testing methods or techniques, there are specialized agents corresponding to different categories of testing methods.

All relevant test objects are modelled as the agent's beliefs including the Web component under test (CUT), the test models representing the test objects for the CUT, the requirements or functional specification of the CUT, the test cases, and test coverage result. Goals include the test requirements or test criteria, for example, percentage of requirements coverage for black-box testing, statement coverage for white-box testing. Goals have utilities which can change due to changes of beliefs. The agent always tries to fulfil the goal with the highest utility.

The action plans of an agent are generated dynamically according to the test goal selected and the current belief of the agent. An action plan is a sequence of actions to be performed by the agent to accomplish the goal. For example, if the current statement coverage is 70%, then what are the sequences of actions that can be executed to accomplish 90% statement coverage? Since each action is associated with a cost, a rational agent should select the sequence of actions that requires the minimal cost.

Finally, the actions of a test agent are implemented by command objects, each of which implements an action and has at least the following: 1) *the activity to be performed* 2) *the costs to perform the activity* 3) *a precondition* to be satisfied and 4) *a post-condition* or effect resulting from the performance of the action.

3.6 A formal agent-based framework for testing web applications

In (Miao et al., 2007), a formal framework for testing Web applications is presented. The goal is to show how the framework assists the design of agent-based Web application testing systems. In this framework, the whole test work of the Web application can be divided into the some small test tasks or subtasks. In this work, the organization-based methodology Gaia (Zambonelli et al., 2003) for multi-agent system analysis and design is employed and extended. Gaia is a methodology for agent-oriented analysis and design. Gaia is founded on the view of a multi-agent system as a computational organization consisting of various interacting roles. For the realisation Object-Z (Smith, 2000), which is a formal specification language for modular design of complex systems, was used.

The executive part of the framework is a multi-agent system (MAS) which implements all the Web test tasks. During the analysis stage, an organization is viewed as a collection of roles. Each test task corresponds to one role. At the run time, the agent takes the role to achieve the test task or interact and cooperate with other agents to finish the test tasks. The agent can not only join or leave agent society at will, but also take or release roles at run time dynamically. The framework can be easily extended by adding new roles to provide much more functionalities for testing Web applications to further enhance the intensity of automation. At the same time, agents and roles are loosely coupled; role classes and agent classes can be designed at the same time by different teams. The internal design of the multi-agent system (MAS) is independent of the Web applications.

If a new test task arrives, and there is no corresponding role in MAS to meet it, a new role can be constructed to satisfy it. Besides, if a test task couldn't be tested enough, the corresponding role can be improved or the corresponding class of role can be re-factored. If the role does not meet the requirement, it can be deleted or replaced by a new one.

The whole framework contains four layers. At the first layer, the Test Tasks Organization defines a set of conceptual test tasks of Web applications and the relationships between test tasks. At the second layer, the Role Organization consists of a set of role classes. At the third layer, the Role Instance Space consists of role instances. Each role instance is an instance of an associated role class which was defined in role organization. At the fourth layer, the Agent Organization one consists of various agents. Agents are free to join or leave the agent organization, and they can take one or more than one role instances. An agent can not only take roles at run time, but also release them if they are not needed any more. The relationships between agents are based on the relationships between roles that are taken.

4. Multi-agent systems approaches in software engineering

In this section, the research work relevant to utilising agent systems as an approach to confront with SE tasks will be highlighted. This work targets web-based applications and distributed database applications.

4.1 An agent-based data-flow testing approach for web applications

In (Qi et al., 2006), an application of the framework introduced in (Kung, 2004) (see 3.5) is presented. In this research, a particular testing approach (Qi et al., 2005) is selected and it is

shown how the framework assists the design of agent-based web application (WA) testing systems.

The testing task can be decomposed into many small subtasks and each subtask can be completed by an autonomous agent. In particular, agent-based data-flow testing is performed at the method level, object level, and object cluster level. Each level of testing is managed by a specific type of test agent. In the process of the recommended data-flow testing, an agent-based WA testing system (WAT) will automatically generate and coordinate test agents to decompose the task of testing an entire WA into a set of subtasks that can be accomplished by test agents.

A high level test agent can create low level test agents and ask them to complete the corresponding low level testing. Based on objects shared by low level test agents, a high level test agent constructs its test models and performs the comparatively high level testing that cannot be accomplished by low level test agents. Consequently, a high level testing task is completed by the cooperation of a set of low level test agents and a high level test agent.

The testing process of the proposed approach is a hybrid of a top-down process, in which a testing task is decomposed into subtasks, and a bottom-up process, in which test agents build test models and perform data-flow testing at corresponding abstraction levels to complete the subtasks.

Similar to the data-flow testing of non-WA, data-flow testing of WA requires adequate test models and proper test criteria. A Control-flow Graph (CFG) annotated with data-flow information is a generally accepted approach to model non-WA. However, a CFG has to be extended to properly handle new features of WA.

In this design, the WAT consists of two types of test agents, a blackboard, and a test case pool. The blackboard serves as the message exchanging centre in WAT and the test case pool that stores all the test cases. The test agent (Rao & Georgeff, 1995) based on the BDI model contains beliefs (observations about the environment and other agents), desires (goals to be accomplished), and intentions (action plans to achieve goals).

4.2 An agent approach to quality assurance and testing web software

In (Zhu, 2004), the application of Lehman's theory (Lehman & Ramil, 2001) of software evolution to web-based applications is studied. It is claimed that web applications are by nature evolutionary and, hence, satisfy Lehman's laws of evolution. The essence of web applications implies that supporting their sustainable long term evolution should play the central role in developing quality assurance and testing techniques and tools. Therefore, two basic requirements of such a software environment can be identified. First, the environment should facilitate flexible integrations of tools for developing, maintaining and testing various kinds of software in a variety of formats over a long period of evolution. Second, it should enable effective communications between human beings and the environment so that the knowledge about the system and its evolution process can be recorded, retrieved and effectively used for future modification of the system.

The solution proposed in (Zhu, 2004) to meet these requirements is a cooperative multi-agent software growth environment (Zhu et al., 2000; Huo et al., 2003). In this environment, various tools are implemented as cooperative agents interacting with each other and with human users at a high level of abstraction using ontology.

The software environment consists of the two types of agents. Service agents provide various supports to the development of software systems in an evolutionary strategy. They fulfil the functional requirements of development and quality assurance and testing,

verification and validation functionalities. Management agents manage service agents and are responsible for the registration of agents' capabilities, task scheduling, and monitoring and recording agents' states and the system's behaviours. Each service agent is specialized to perform a specific functional task and deal with one representation format. They cooperate with each other to fulfil more complicated tasks.

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, management agents are implemented as brokers to negotiate with testing service agents to assign and schedule testing activities to testing service agents. Each broker manages a registry of agents and keeps a record of their capabilities and performances. Each service agent registers its capability to a broker when joining the system. Tests tasks are also submitted to the brokers.

These agents co-exist with the application software system throughout the application system's whole lifecycle to support the modifications of the system. They monitor the evolution process and record the modifications of the system and the rationales behind the modifications. They extract, collect, store and process the information about the application system and its performance, and present such knowledge to human beings or other software tools when requested. They interact with the users and developers cooperatively.

The environment grows with the application system as new tools are integrated into the environment to support the development and maintenance of new components and as the knowledge about the system is accumulated over the time. Such a software environment is called a growth environment. It significantly differs from software development environments and run-time support environments such as middleware, where evolution is not adequately supported.

In order to enable agents to cooperate effectively with each other and with human users, they communicate with each other through a flexible and collaboration protocol and codify the contents of messages in an ontology which represents knowledge about the application domain and software engineering (Zhu & Huo, 2004). The interaction protocol is developed on the basis of speech-act.

Agent	Functionality
GWP: Get Web Page	Retrieve web pages from a web site
WPI : Web Page Information	Analyse the source code of a web page, and extract the metadata, hyperlinks and structural information from the code
WSS: Web Site Structure	Analyse the hyperlink structure of a web site, and generate a node-link-graph describing the structure
TCG: Test Case Generator	Generate test cases to test a web site according to certain testing criteria
TCE: Test Case Executor	Execute the test cases, and generate execution results
TO: Test Oracle	Verify whether the testing results match a given specification
TA: Testing Assistant	Perform as user interface and guide human testers in the process of testing
WSM: Web Site Monitor	Monitor the changes of web sites, and generate new testing tasks accordingly

Table 1. Agents for testing web applications.

4.2.1 Developing a software testing ontology

In (Zhu & Huo, 2004), the design and utilisation of a software testing ontology is proposed. This attempt has the target to enrich the approach presented in (Zhu, 2004). It represents the knowledge of software engineering and codifies the knowledge for computer processing as the contents of an agent communication language. The ontology is represented in UML at a high level of abstraction so that it can be validated by human experts. It is also codified in XML for computer processing to achieve the required flexibility and extendibility. The concepts of the ontology and the relations between them are defined while their properties are also analysed. Speech-act theory is incorporated in the system and combined with the ontology to define communication protocols and to facilitate collaborations between agents.

In order to specify this ontology, a testing concept taxonomy is introduced. Taxonomy is a way to specify and organize domain concepts. Concepts are divided related to software testing into two groups: the basic concepts and compound concepts. There are six types of basic concepts related to software testing, which include *testers*, *context*, *activities*, *methods*, *artefacts*, and *environment*. Compound concepts are those defined on the bases of basic concepts, for example, testing tasks and agent's capability. Relationships between basic concepts as well as compound concepts are also introduced. Basic relations between basic concepts form a very important part of the knowledge of software testing. Therefore, they are stored in a knowledge-base as basic facts.
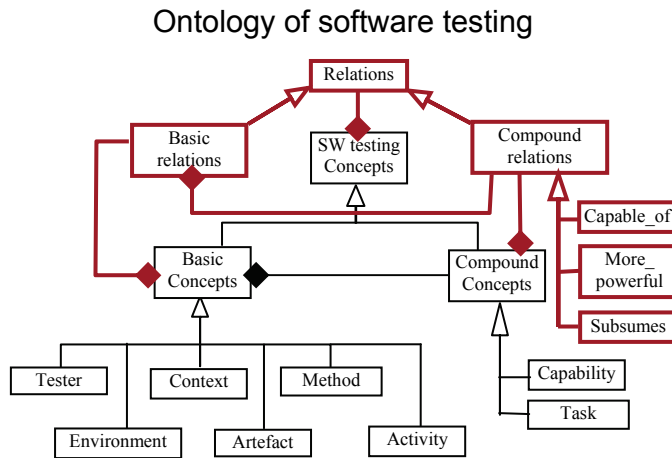
Ontology of software testing

Fig. 1. Ontology of software testing

4.3 An agent approach for the maintenance and testing of database applications

In (Gardikiotis et al., 2007a), an approach for the software engineering of distributed database applications (DA) is presented. The approach is founded on the employment of software agents and adopts the architecture of an intelligent, flexible and extensible agent system that complies with the nature of multi-tier DAs. Among these agents, there are specialized agents that are capable of performing the software maintenance and testing tasks for the DAs' source code by supporting techniques and metrics tailored for this application type. There exist also general-purpose agents that provide significant information that can be used by other DAs' software engineering tasks (Gardikiotis et al., 2007b).

The rationale behind utilizing agent technology has to do with the interoperability of the software resources belonging to potentially disparate application components and disparate domains. Towards this direction, agents offer a unified platform of interaction through agent communication, exhibiting the following characteristics:

- Extensibility and scalability. The presented architecture can easily be extended to support other software engineering tasks. In fact, the presented system is derived from an extended version of previous work described in (Gardikiotis et al., 2007a), which focused solely on software maintenance.
- No performance degradation. The communication overhead caused by agent interaction is minimal in comparison with the process time of each individual software engineering task itself (such as the graph construction, the test case generation etc.).
- Intelligent and pro-active behaviour. The system functions in an adaptive manner by improving its mode of operation according to application complexity and coupling.
- Declarative ontology. This approach manages to encompass a customizable but formal knowledge representation to the overall agent system.

Distributed application nature. The distributed nature of the agent system fits well with the distributed nature of multi-tier applications.

4.3.1 Architecture

The architecture of the presented system is shown in Figure 2. The agents that are general in the DAs' software engineering processes are grey-coloured, whereas the maintenance agents' names are written in italics and the testing agents' names are underlined. Following a top-down approach, the role of each agent involved in the system is described.
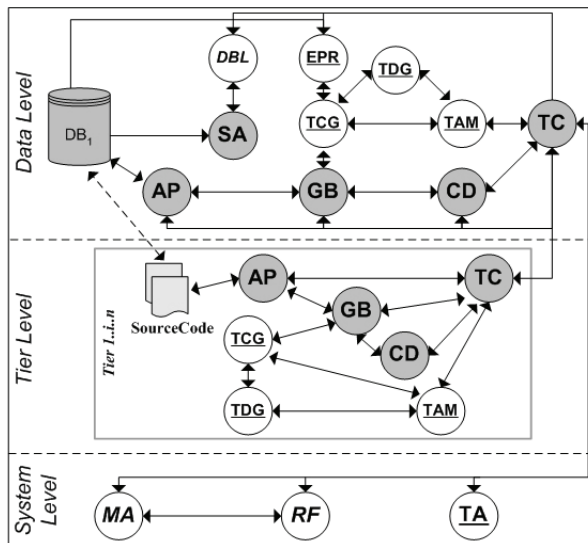

Fig. 2. Architecture

At the data level of the system, the Schema Analyzer (SA) agent stores a representation of the database schema in order to identify inter-dependencies between the database objects. The Database Listener (DBL) agent monitors the underlying databases ($DB_1 \ldots DB_n$) for any

potential changes, requests from the Schema Analyzer the full set of affected database objects and can initiate the process of identifying the impact of each change into the application code by requesting this analysis from the Maintenance Assessor (MA) agent. The Execution Plan Retriever (EPR) agent retrieves from the database/s the execution plan for a specific database statement, which is given by a request from the data Tier Coordinator (TC). The TC is common for all levels, namely the data, the tier and the system level and acts as a broker, i.e. any communication between agents of different levels is transmitted through this agent. Moreover, this agent keeps track of the actual execution traces of the system that will be necessary in case of a dynamic analysis approach.

Apart from the TC, the data and the tier level share also the following agents: the Application Parser (AP), the Graph Builder (GB), the Test Cases Generator (TCG), the Test Data Generator (TDG), the Test Adequacy Measurer (TAM) and the Clustering Detector (CD). The AP parses and analyses the source code for all units included in the specific tier while the GB creates an abstract graph representation of the tier code. This representation has the form of different types of graphs that facilitate program comprehension together with the application of testing, maintenance and clustering techniques. It can also be used for the impact analysis performed by the Maintenance Assessor (MA) agent.

The graphs derived from the GB are used by the CD and the TCG. The former agent investigates the partitioning of the graph based on metrics provided by the TCG and the MA agents. The latter agent generates test cases for the provided graphs according to some adequacy criteria defined and referred to by the TAM. The produced set of test cases is given as input to the TDG which generates the corresponding set of test data.

The system level of the infrastructure includes the Maintenance Assessor (MA), the Refactorer (RF) and the Testing Assistant (TA) agents. The MA assesses the DA's maintainability with reference to the schema of the underlying database and estimates the impact of a potential change in the database schema into the application source code. It has to retrieve the units/statements that are related to the altered database objects in order to offer an indication of the workload with respect to the source code changes that might be needed to retain its operability. The RF provides specific semantic-preserving transformations that aim to increase the DA's maintainability.

Lastly, the TA triggers and controls the overall testing process. The trigger event can be either a human request or a request from the MA, which informs the TA about the effects of the maintenance process on the DAs' source code.

In this system, agents of similar functionalities may have different capabilities and they may deal with heterogeneous information formats. They can also be implemented using different algorithms and they can be executed on different platforms. Agents can enter the system and other agents can abandon the system dynamically. Therefore, agents register their capabilities to a specialized agent that the system offers, namely the matchmaker agent (MM). This agent offers a directory-like service (Lazarou & Clark, 1998) very common to the agent literature. It accepts and stores registrations and de-registrations from other agents in an internal knowledge base (KB). Task requests are also submitted to this agent in order to find other agents that provide a set of desired capabilities. After accepting such a request, the MM has the job to look up in the KB, to retrieve the agent(s) that best match the criteria and to reply to the agent that sent the request with the id(s) of the retrieved agent(s). From this point onwards, agents can employ direct communication.

With respect to ontological issues, in this work the focus is on classifying and representing software engineering concepts. A categorization widely acceptable in the software

engineering community is used. This illustration (Figure 3) is based on (IEEE, 2004) and (SWEBOK, 2004). In addition some topics (e.g. testing levels), which are highly relevant to the tasks of the agents, are further analyzed.
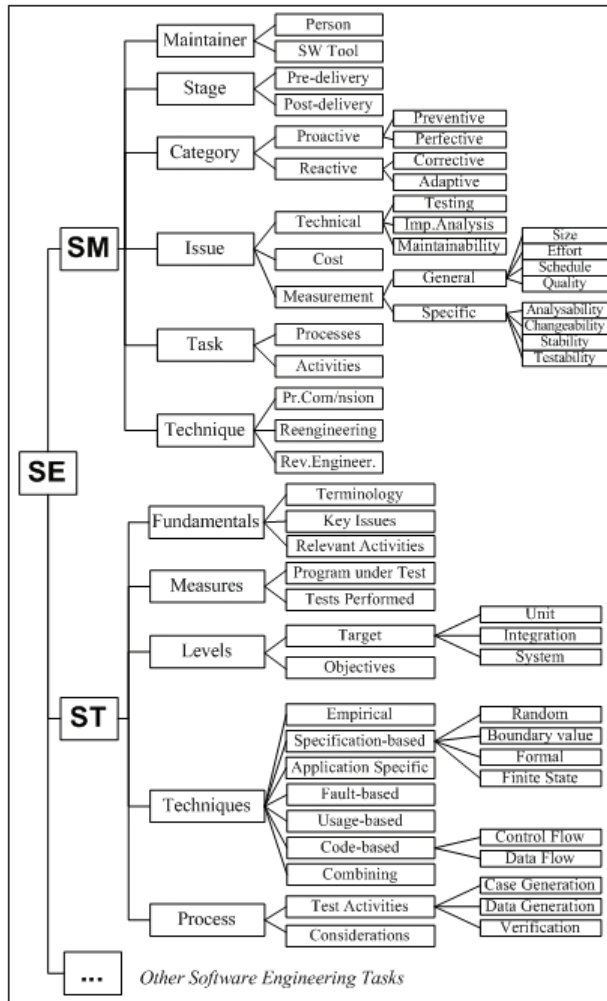

Fig. 3. Software Engineering (SE) Taxonomy

4.3.2 The agents

The agents can be categorized in three groups according to their intending tasks: software maintenance agents, software testing agents and general software engineering purpose agents.

Software Maintenance Agents

Maintenance Assessor (MA): provides an assessment of the DAs' maintainability against schema changes. It is a system-level agent that triggers and guides the maintenance process

receiving a request from the data-level TC that was initially sent by the DBL. A graphical user interface is additionally provided for human user requests. To retrieve the information required for the assessment the MA communicates with the TCs. Upon the completion of its assessment task, the MA may request from the RF a set of refactorings in order to achieve a specified level of maintainability. Furthermore, it can request from the TA to trigger the testing process in order to ensure the DAs' source code validity.

Refactorer (RF): provides a set of refactorings to increase the maintainability of the DA. Refactoring can be defined as a technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour (Fowler, 1999), i.e. practically each refactoring can be viewed as a semantics preserving transformation.

Software Testing Agents

Test Case Generator (TCG): generates a set of test cases that usually refers to an abstract representation of the application source code depending on the supported technique type. The effectiveness of the generation process can be assessed by measuring the coverage of specific test adequacy criteria.

Test Data Generator (TDG): given a set of test cases the TDG automatically produces test data for them using a supported test data generation algorithm.

Test Adequacy Measurer (TAM): based on the specific testing objective the TAM proposes and measures the coverage of a set of test adequacy criteria.

Execution Plan Retriever (EPR): given a database statement the EPR retrieves from the DBMS the corresponding execution plan. This plan is necessary for the TCG to produce test cases for DAs.

Testing Assistant (TA): the TA is a system-level agent that guides the testing process. To trigger testing it either receives a request from the data-level TC or from the system-level MA agents. This request contains a description of the changes in the database schema or the DA's source code respectively. Furthermore, the agent provides a user interface to accept requests from a human tester. The TA decides on the level of testing and the test adequacy criteria based on the available information about coupling and complexity metrics as well as the sizes and the number of DA's clusters.

General Software Engineering Agents

Database Listener (DBL): captures the modifications made in the database schema and triggers the impact assessment.

Application Parser (AP): parses and statically analyses the DA's unit source code. The information gained from the analysis constitutes the basis for the performance of software engineering activities such as testing and maintenance.

Graph Builder (GB): provides a set of graph representations of the DA's source code, which is independent from the implementation language.

Tier Coordinator (TC): the TC agent serves as a local matchmaker agent (MM), i.e. it offers a directory-like service. It is aware of each tier-based agent capabilities (after receiving a corresponding register message) and uses this knowledge upon a request that is submitted by tier independent agents or other TCs located on different tiers/levels.

Schema Analyzer (SA): the SA agent resides in the data-tier and keeps a representation of the database schema in order to effectively detect dependencies between the database objects.

Clustering Detector (CD): detects the possibility of application clustering that will facilitate the software testing activities. Clustering refers to collections of source code units that are more or less relevant to activity's target.

5. Agent-oriented software engineering

It has been already mentioned that the focus of this chapter is not about applying software engineering models to assist the creation of multi-agent systems. However, in the last years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there has been a great deal of research related to the identification and definition of suitable models and techniques to support the development of complex software systems in terms of MAS (Gervais et al., 2004). As a result, in order to augment the completeness of the survey, a brief depiction of this research area follows.

This research, which can be roughly grouped under the term ''agent-oriented software engineering'', proposes a variety of new metaphors, formal modelling approaches, development methodologies and modelling techniques, specifically suited to the agent-oriented paradigm. The current trends in this area are outlined as follows (Zambonelli & Omicini, 2004):

- Agent modelling. Novel formal and practical approaches to component modelling are required, to deal with an agent as an autonomous, pro-active, and situated entity. A variety of agent architectures are being investigated, each of which is suitable to model different types of agents or specific aspects of agents: purely reactive agents, logic agents (Van der Hoek & Wooldridge, 2003), agents based on belief, desire and intentions (Rao et al., 1995). Overall, this research has so far notably clarified the very concept of agency and its different facets.

- MAS architectures. As it is necessary to develop new ways of modelling the components of a MAS, in the same way it is necessary to develop new ways of modelling a MAS as a whole. Detaching from traditional functional-oriented perspectives, a variety of approaches are being investigated to model MAS. In particular, approaches inspired by societal, organisational, and biological metaphors, are the subject of the majority of researches and are already showing the specific suitability of the different metaphors in different application areas.

- MAS methodologies. Traditional methodologies of software development, driving engineers from analysis to design and development, must be tuned to match the abstractions of agent-oriented computing. To this end, a variety of novel methodologies to discipline and support the development process of a MAS have been defined in the past few years (Kolp et al., 2002; Wood et al., 2001), clarifying the various sets of abstractions that must come into play during MAS development and the duties and responsibilities of software engineers.

- Notation techniques. The development of specific notation techniques is needed to express the outcome of the various phases of a MAS development process; traditional object- and component-oriented notation techniques cannot easily apply. In this context, the AUML proposal (Bauer et al., 2001), extending standard UML toward agent-oriented systems, is the subject of a great deal of research and it is rapidly becoming a de facto standard.

- MAS infrastructures. To support the development and execution of MAS, novel tools and novel software infrastructures are needed. In this context, various tools are being proposed to transform standard MAS specifications (i.e., AUML specifications) into actual agent code (Bergenti & Poggi, 2002), and a variety of middleware infrastructures have been deployed to provide proper services supporting the execution of MAS.

With respect to MAS methodologies, research work involves the definition of a common framework for MAS specification, which includes the identification of a minimum set of concepts and methods that can be agreed in the different approaches (Bernon et al., 2006). The tool for defining this framework is meta-modelling. Achieving concrete results in this area would be very useful for several reasons:

1. This partly solves the lack of standardization in this area.
2. This could encourage the development of more flexible and versatile design tools.
3. This is one of the essential steps for reaching a concrete maturity in the study of the whole agent design process.

The definition of MAS meta-models has led to the identification (and formalization) of a unified meta-model. Nevertheless, the research is still in its early stages, and several challenges need to be faced before agent-oriented software engineering can deliver its promises, becoming a widely accepted and a practically usable paradigm for the development of complex software systems.

6. Conclusion

In this chapter, the application of multi-agent systems to tackle the software engineering task was outlined. The concentration was on the employment of agent technology in order to deal with distributed software systems and mainly distributed database applications and web applications.

The rationale behind utilizing agent technology has to do with the multi-tier architecture and the associated inherent complication of distributed applications and the required interoperability of software resources belonging to potentially disparate application components and disparate domains. To meet these requirements, agents offer a unified platform of interaction through agent communication.

The current research status can be classified according to two principal tracks. The first one has as a goal to provide an agent infrastructure to support software testing. This is realised by suggesting multi-agent frameworks that can be used as a model to build agent systems for testing service-oriented web applications. The second one has a more applied nature. This research track aims at presenting an agent system for tackling the issues of software maintenance and testing of distributed applications.

Analysing the aforementioned research attempts some general comments can be stated. A first and important comment is that all approaches have a quite narrow scope. On the one hand, the application domain is related to web services, web applications and database applications. The only exception is the work of (Yamany et al., 2006) but even in this case only 3-tier applications are considered. These domains have a surely specific nature even though they provide a solid basis for introducing the existing attempts.

Moreover, this restriction is made clearer by the fact that the software engineering process is not covered in its complete form. Almost all attempts target software testing with the exception of (Gardikiotis et al., 2007a) where software maintenance is also treated in depth. The above work is also the only one where the existing platform has proven its extensibility by including generic software engineering agents.

Focusing on infrastructural approaches, the work of both (Ma et al., 2007) and (Xu et al., 2006) has a very specific objective which is to support agent collaboration. Besides this commonality, the research of (Xu et al., 2006) is more tailored to software testing encompassing the notion of test tasks while the one of (Ma et al., 2007) recommends an

agent design for web service autonomy. However, in both cases there is no actual system to verify the expected benefits of the two mechanisms.

With respect to agent frameworks (Yamany et al., 2006; Bai et al., 2006; Kung, 2004; Miao et al., 2007), the common aspiration is to model software testing. The testing process is decomposed into phases during test planning while these plans can be executed asynchronously. Additionally, different testing techniques can be chosen by different agents, the agent society is dynamic (agents can enter or exit the system during execution time) while the whole procedure is being coordinated by specialized agents.

The proposals of (Kung, 2004; Miao et al., 2007) offer an additional benefit that they are based on a sound formal ground employing the BDI metaphor and the Gaia methodology respectively. The work of (Kung, 2004) extends UML to put forward novel agent-oriented diagrammatic techniques that are anticipated to assist agent modelling. The research of (Miao et al., 2007) exhibits advanced flexibility since agents can change testing roles dynamically. Finally, in all cases besides (Kung, 2004) no particular approach has been bundled to validate the strength of the model's functionality while the issues of test planning optimization and agent society evolution need further exploration.

Proceeding with multi-agent systems approaches (Qi et al., 2006; Zhu, 2004; Gardikiotis et al., 2007a) they do not share many things in common. In all three approaches, agents can be designed for different tasks, deal with different representation formats and deployed on different platforms. In both (Qi et al., 2006; Zhu, 2004) the application domain is the one of web applications where test tasks are decomposed into subtasks and test agents that undertake these subtasks work together to complete the testing task. In (Qi et al., 2006) the objective is to show an implementation of (Kung, 2004) by adjusting a data-flow testing method to properly handle web applications.

The approaches of (Zhu, 2004; Gardikiotis et al., 2007a) suggest enriched architectures since they have an evolutionary and adaptive nature where existing techniques can be adapted to new application environments while new techniques can be also plugged in. Furthermore, ontological aspects are taken into consideration. Nevertheless, ontological treatment is substantially different. In (Zhu, 2004) a specialized taxonomical scheme is devised by the author to support software testing. The key offering is that besides basic concepts, compound concepts and concept relationships can be expressed. On the other hand, in (Gardikiotis et al., 2007a) the ontological representation is grounded on IEEE standards making it undoubtedly acceptable in almost any application environment. And although currently no compound concepts or concept relationships are defined, the selected representation leaves room to encompass such features in the future. In addition, a drawback of the ontological scheme proposed in (Zhu, 2004) is that it is represented in two different notations, UML and XML. This raises an issue of how to definitely ensure the consistency between them.

Concluding, the level of agent sophistication is also dissimilar. In (Zhu, 2004) agent functionalities are relatively straightforward since the focus is in other aspects. On the contrary, there are several agents that employ advanced intelligent techniques; for example the ones responsible to endorse the tasks of clustering and refactoring.

6.1 Future work

There are different future directions with respect to applying agent systems technology in software engineering. Starting with the current research status that introduces agent infrastructural frameworks, the following can be stated:

- Investigating the application of agent technology to model software engineering tasks other than software testing is obviously a desired future path.
- Applying the framework in a variety of distributed systems is absolutely necessary to optimise the model's functionality.
- Since this work has a somewhat theoretical nature, it is important that tools are developed to verify and validate the models through the use of a set of concrete test agents.
- Integrating third-party technology, methods or tools to the framework is expected to constantly increase its functionalities.
- Designing of more specific role organizations (that have to be consistent with corresponding agent organizations) and more formal definition of the mechanism of test planning is also advisable. This can include rule-based test planning, partially order plan generation and plan partitioning.

Continuing with current research relevant to agent multi-agent systems approaches, some of the remarks to be stated share some similarity to the above ones. More specifically:

- Completing the picture of the software engineering process would be a nice step forward.
- Expanding the work to handle a diversity of distributed software systems is also needed.
- The current approaches have reached a prototype level. Thoroughly testing, evaluating and deploying the agent systems, is in demand so that these approaches reach the level of a full-fledged ready to use system.
- Implementing an even richer variety of test agents. Especially, it would be really significant to employ deeper intelligent techniques (coming from the Machine Learning literature for example) in order to enhance the agent capabilities.
- Establishing a common ontological representation. This representation has the goal to be on the one hand readable and declarative from the human point of view and on the other hand flexible and able to be captured from the machine part. An agent-oriented modelling language such as AUML could prove necessary to catch the agents' autonomous and social behaviours.

A more detailed comment about web systems is that extending the current work to handle dynamically generated Web pages and to incorporate automatic test case generation techniques such as navigation testing and object state testing would refine the agent approach.

7. References

Bai, X.; Dai, G.; Xu, D. & Tsai, W.T. (2006). "A Multi-Agent Based Framework for Collaborative Testing on Web Services", Proceedings of the 2nd International Workshop on Collaborative Computing, Integration, and Assurance, WCCIA 2006, Page(s): 205-210.

Bauer, B.; Muller, J. P. & Odell, J. (2001). ''Agent UML: A formalism for specifying multi-agent software systems,'' Int. J. Soft. Eng. Knowl. Eng. vol. 11, no. 3, pp. 207 – 230.

Bergenti, F. & Poggi, A. (2002). ''Agent-oriented software construction with UML,'' in The Handbook of Software Engineering and Knowledge Engineering - volume 2 - Emerging Technologies, World Scientific: Singapore, pp. 757 – 769.

Bernon, C.; Cossentino, M. & Pavon, J. (2005). Agent-oriented software engineering. *The Knowledge Engineering Review*, Vol. 20, no. 2, pp. 99-116, June 2005

Buhler, P. & Vidal, J. M. (2003). "Semantic Web Services as Agent Behaviours," in Agent-cities: Challenges in Open Agent Environments, LNCS/LNAI, B. Burg, J. Dale, et al., Eds. Berlin: Springer-Verlag.

Di Lucca, G.A. & Fasolino, A.R. (2006). Testing Web-based applications: The state of the art and future trends. Information and Software Technology. 48, 12 (Dec. 2006), 1172-1186.

Erl, T. (2005). Service-oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, NY, US.

Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley, 0201485672, 1999.

Gardikiotis, S. K.; Lazarou, V. S. & Malevris, N. (2007a). "An Agent-based Approach for the Maintenance of Database Applications", *Proc. 5th Int. Conference on Software Engineering Research and Applications*, IEEE Computer Society, pp.558-565, Busan, Korea, August 2007.

Gardikiotis, S. K.; Lazarou, V. S. & Malevris, N. (2007b). "Employing Agents towards Database Applications Testing", *Proc.* 21st International Conference on Tools with Artificial Intelligence (ICTAI'07), IEEE Computer Society, pp. 157-166, Patras, Greece, October 2007.

Gervais, M.; Gomez, J. & Weiss, G. (2004). ''A survey on agent-oriented software engineering researches,'' in: Methodologies and Software Engineering for Agent Systems, Kluwer: New York (NY).

Huo, Q.; Zhu, H. & Greenwood, S. (2003). A Multi-Agent Software Environment for Testing Web-based Applications, Proc. of COMPSAC'03, Dallas, 2003, 210-215.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology, 610.12-1990

IEEE (2004). IEEE Standard for Software Maintenance, IEEE, 2004 {IEEE1219-2004}.

IEEE (1996). IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, Industry Implementation of Int. Std.ISO/IEC 12207:95, Standard for Information Technology - Software Life Cycle Processes, IEEE, 1996 {IEEE12207.0-96}.

ISO (2006). ISO/IEC Standard for Software Engineering - Software Life Cycle Processes – Maintenance, ISO/IEC, 2006 {ISO/IEC 14764}.

Jennings, N. R. (2001). ''An agent-based approach for building complex software systems,'' Commun. ACM, vol. 44, no. 4, pp. 35 – 41.

Kavi, K.; Kung, D.; Bhambhani, H.; Pancholi, G. Kanikarla, M. & Shah, R. (2003). "Extending UML to Modelling and Design of Multi-Agent Systems," In Proc. of ICSE 2003 Workshop on Software Engineering for Large Multi-Agent Systems (SELMAS), Portland, Oregon, May 3–4, 2003.

Kolp, M.; Giorgini, P. & Mylopoulos, J. (2002). ''A goal-based organizational perspective on multi-agent architectures,'' in Intelligent Agents VIII: Agent Theories, Architectures, and Languages, vol. 2333 of LNAI, Springer-Verlag, pp. 128 – 140.

Kung, D.; Bhambhani, H.; Nwokoro, S.; Okasha, W.; Kambalakatta, R. & Sankuratri, P. (2003). "Lessons learned from software engineering multi-agent systems," Proc. of IEEE COMPSAC'03, Dallas, Texas, November 3–6, 2003.

Kung, D. (2004). "An agent-based framework for testing Web applications", Proceedings of the 28th Annual International Computer Software and Applications Conference, COMPSAC 2004, vol.2, Page(s): 174-177.

Lazarou, V. S. & Clark, K. L. (1998). "Agents for Hypermedia Information Discovery", Agents' World 98, *Co-operative Information Agents, Springer-Verlag Lecture Notes in Artificial Intelligence* (1435), 1998.

Lehman, M. M. & Ramil, J. F. (2001). Rules and Tools for Software Evolution Planning and Management. Annals of Software Engineering, Special Issue on Software Management, 11(1), 15-44.

Ma, Y.F.; Li, H.X. & Sun, P. (2007). A Lightweight Agent Fabric for Service Autonomy. Proc. of International Workshop on Service-Oriented Computing: Agents, Semantics, and Engineering, 2007, LNCS 4504, pp. 63-77, Springer-Verlag

Maamar, Z.; Kouadri-Most´efaoui, S. & Yahyaoui, H. (2005). "Towards an Agent-based and Context-oriented Approach for Web Services Composition," IEEE Transactions on Knowledge and Data Engineering, Vol. 17, No. 5, pp. 686-697.

Miao, H.; Chen, S. & Qian, Z. (2007). A Formal Open Framework Based on Agent for Testing Web Applications, International Conference on Computational Intelligence and Security, 2007, pp. 281-285, 0-7695-3072-9

Nguyen, H.Q. (2000). Testing Applications on the Web: Test Planning for Internet-Based Systems, John Wiley & Sons, Inc.

Qi, Y.; Kung, D. & Wong, E. (2005). "An Agent-Based Testing Approach for Web Applications", Proceedings of Computer Software and Applications Conference, COMPSAC 2005, Volume 2, Page(s): 45-50.

Qi, Y.; Kung, D. & Wong, E. (2006). "An agent-based data-flow testing approach for Web applications." Information and Software Technology. 48, 12 (Dec. 2006), 1159-1171.

Rao, A.S. & Georgeff, M. (1995). BDI agents: from theory to practice, in: Proceedings of the First International Conference on Multi-Agent System (ICMAS'95), San Francisco, CA, USA, pp. 312–319.

Richards, D.; van Splunter, S.; Brazier, E. & Sabou, M. (2003). "Composing web services using an agent factory," In Prc. of the 1st Workshop Web Services arid Agent Based Engineering, Sydney, Australia.

Russel, S. & Norvig, P. (2003). Artificial Intelligence: A Modern Approach, Prentice Hall/Pearson Education International: Englewood Cliffs (NJ), (2nd Edn), 2003.

Saint-Andre, P. (2005). Streaming XML with Jabber/XMPP, IEEE Internet Computing, Volume 9, Issue 5, Page(s):82 – 89.

Smith, G. (2000). The Object-Z Specification Language. Advances in Formal Methods. Kluwer Academic Publishers.

Sommerville, I. (2004). Software Engineering, 7th edition, Addison-Wesley, 2004.

SWEBOK (2004). Guide to the Software Engineering Body of Knowledge (February 6, 2004). Retrieved on 2008-02-21.

Sycara, K.; Paolucci, M.; Soundry, J. & Srinivasan, N. (2001). "Dynamic discovery and coordination of agent-based semantic web services," IEEE Computing, 8(3):66-73.

Tsai, W.T.; Paul, R.; Yu, L.; Saimi, A. & Cao, Z. (2003). "Scenario-Based Web Service Testing with Distributed Agents," IEICE Transaction on Information and System, Vol. E86-D, No. 10, pp. 2130-2144.

Van der Hoek, W. & Wooldridge, M. (2003). ''Towards a logic of rational agency,'' Logic J. IGPL, vol. 11, no. 2, pp. 135 – 160.

Wood, M.; DeLoach, S. A.; & Sparkman, C. (2001). ''Multi-agent system engineering'', Int. J. Software Eng. Knowl. Eng., vol. 11, no. 3, pp. 231 – 258.

Xu, D.; Bai, X.; & Dai, G. (2006). "A Tuple-Space-Based Coordination Architecture for Test Agents in the MAST Framework," Second IEEE International Symposium on Service- Oriented System Engineering (SOSE'06), pp. 57-66, 2006

El Yamany, H.F., Capretz, M.A.M. & Capretz, L.F. (2006) A Multi-Agent Framework for Testing Distributed Systems, IEEE COMPSAC 3rd International Workshop on Quality Assurance and Testing Web-Based Applications (QATWBA'2006), Chicago, IL, USA, September, pages 151–156.

Zambonelli, F.; Jennings, N.; Omicini, A.; & Wooldridge, M. (2001). ''Agent-oriented software engineering for internet applications,'' in Coordination of Internet Agents: Models, Technologies, and Applications, Springer-Verlag: Berlin (D), pp. 326 – 346.

Zambonelli, F.; Jennings, N.; & Wooldridge, M. (2003). ''Developing multi-agent systems: The Gaia methodology,'' ACM Trans. Soft. Eng. Meth., vol. 12, no. 3, pp.417 – 470.

Zambonelli, F. & Omicini, A. (2004). Challenges and Research Directions in Agent-Oriented Software Engineering, *Journal of Autonomous Agents and Multi-agent Systems* 9(3), pp. 253-284.

Zhu, H., Greenwood, S., Huo, Q. & Zhang, Y. (2000). Towards agent-oriented quality management of information systems, Workshop Notes of 2nd International Workshop on Agent-Oriented Information Systems at AAAI'2000, Austin, USA, July 30, 2000, 57-64.

Zhu, H. & Huo, Q. (2004). Developing A Software Testing Ontology in UML for A Software Growth Environment of Web-Based Applications, Software Evolution with UML and XML, Hongji Yang (eds.), Idea Group Inc, 2004.

Zhu, H. (2004). "Cooperative agent approach to quality assurance and testing Web software", Proceedings of COMPSAC 2004 vol.2, Page(s): 110-113.
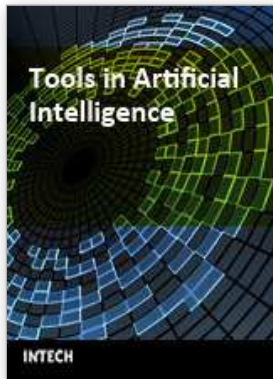
Tools in Artificial Intelligence

Edited by Paula Fritzsche

This book offers in 27 chapters a collection of all the technical aspects of specifying, developing, and evaluating the theoretical underpinnings and applied mechanisms of AI tools. Topics covered include neural networks, fuzzy controls, decision trees, rule-based systems, data mining, genetic algorithm and agent systems, among many others. The goal of this book is to show some potential applications and give a partial picture of the current state-of-the-art of AI. Also, it is useful to inspire some future research ideas by identifying potential research directions. It is dedicated to students, researchers and practitioners in this area or in related fields.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Vasilios S. Lazarou, Spyridon K. Gardikiotis and Nicos Malevris (2008). Agent Systems in Software Engineering, Tools in Artificial Intelligence, Paula Fritzsche (Ed.), ISBN: 978-953-7619-03-9, InTech, Available from:

http://www.intechopen.com/books/tools_in_artificial_intelligence/agent_systems_in_software_engineering

INTECH
open science | open minds