An Industry Use Case: testing SOA systems with MAS simulators

Pier-Giovanni Taranti and
Carlos José Pereira de Lucena
PUC-Rio, Rua M. de São Vicente 225
Rio de Janeiro/RJ, Brazil
pier.taranti@les.inf.puc-rio.br, lucena@inf.puc-rio.br

Ricardo Choren
SE/8 - IME, Pça General Tibúrcio, 80
Rio de Janeiro/RJ, Brazil
choren@ime.eb.br

***Abstract*—System Test in architectures composed of several asynchronous subsystems is a hard task. The simulation of external systems is usually performed in a limited way, considering only test cases provided, one by one. This paper presents a simulator based on MAS, used to test critical operation software. The simulator, with simple architecture and construction, has supported interface and integration testing phases and also is used to obtain performance metrics to validate non-functional requirements.**

I. Introduction

Service Oriented Architecture (SOA) is an integrated software infrastructure and design approach to deliver business functions as shared and reusable services. SOA offers more flexibility and looser coupling being more suitable for internet computing [1]. SOA is often used both in intra-enterprise-integration (e.g. Message Oriented Messaging systems) and in inter-enterprise-integration (e.g. Web services integration) [2]. Indeed, SOA separates functions into distinct units, or services, which developers make accessible over a network in order that users can combine and reuse to build different applications [3]. The system provides the user requirements by orchestrating an activity between two or more services.

SOA systems are often asynchronous thus testing can be very challenging. The continuously increasing size and distribution of SOA systems make the testing task more complex and increase the size of test code [4], [5]. Indeed, the dynamic an adaptive nature of SOA makes most of the existing testing techniques not directly applicable to test services and service-orchestrated systems [6].

Testing of such systems therefore, often goes hand in hand with setting up test systems performing some message exchanges and to analyse the results [2]. Moreover, testers may not have access to the source code of the services provided by the other parties and they may have no control over the executable code, which may run in any computer over the internet. Thus SOA testing can be very time consuming and inefficient, as manual intervention is needed.

Asynchronous supporting systems (simulators) that are able to act both as a provider and as a consumer of SOA application services can be used for testing purposes. Simulators can provide valid and controlled data for running tests, acting as external actors, to verify the expected behavior of orchestrated systems. The use of simulators is regarded as an effective way to verify and validate SOA applications before these are deployed and executed [7].

To observe the internal behavior of a SOA application, a simulator should send requests to application, receive answers from it and verify if the answers are appropriate, according to pre-defined test cases. Besides, in large SOA applications, not all services may be implemented before testing activities begin. Thus a simulator may also be used to fill in for missing services. Multi-agent distributed simulators provide the flexibility, modularity and scalability desired for simulating complex systems [8]. A multi-agent distributed simulator is a multi-agent system (MAS), i.e. a system composed of interacting software agents [9]. A software agent is a software component that is able to perceive its environment and act according to its design goals [9], [10].

In this paper we present a MAS simulator to perform integration and performance testing in SOA applications before they are deployed into production environment. MAS are capable of simulating systems with large number of heterogeneous entities behaving differently in dynamic situations [9]. Therefore MAS are more suitable for evaluating distributed systems that involve complex interaction between entities, e.g. service orchestration, human interaction with SOA applications. As agents can simulate these interactions, it can be used to test SOA applications since early stages of development, using less resources.

This paper shows an application of the proposed simulator in a maritime system testing. This system was developed by the Brazilian Navy for ship monitoring in international trips in order to improve safety and security at sea. This system was developed using SOA because it should be integrated with the Long Range Information and Tracking (LRIT) system, which was created by the Fifth Amendment to the International Convention for the Safety of Life at Sea (SOLAS).

The rest of the paper is organized as follows. In section 2 we give an overview on testing SOA applications developed using Web services. The LRIT system, focusing on testing issues, is presented in section 3. In section 4 we present the proposed simulator, discussing its use. Section 5 presents some related word. Finally section 6 presents the concluding remarks and points out possible future work directions.

II. Testing SOA Systems

In a SOA system, services may be developed by different teams, from different organizations, and the complete application can be orchestrated later. Testing a SOA system presents many issues, including [11]:

- lack of software artifacts (code and structure)
- dealing with incomplete systems (services bind at run time)
- lack of control over components
- lack of trust in information provided by components
- cost of testing

An integration test technique aims at effective observations of the interfaces between parts of software systems through the development and use of executable test scripts. These scripts, implemented using drivers, hard-code the variables and the expected results for each case. This approach helps to detect a considerable amount of faults, but it is not feasible to test all the possible service interdependencies. Changes in a service source code and interface may require changes in the test script. This is not always a simple task since very often there is no stable test environment. Besides, a service may have to rely on other services to properly perform its functionality. Thus adequate testing may have to be postponed to when all service binding actually happens.

When working with SOA, it would be desirable for a test driver to continuously and actively perform case tests, i.e. the driver should be able to keep generating requests to the SOA application. The driver should also be able to process large volume of data to test non-functional requirements such as availability and performance. To automate the execution of these drivers, service consumers are usually instantiated and executed in application servers. This can be extremely resource and time consuming, and error-prone.

A possible alternative to testing SOA applications is the use of a multi-agent-based simulation system. The pro-active nature of agents can be used to verify the interactions between entities. The agents can generate requests to the existing services and check if their response are appropriate. An agent can also simulate human interaction by generating user-interface related data.

Another advantage of using the agent paradigm to test SOA applications is that an agent can simulate a whole service. For inter-interprise integration SOA applications, this is very important. Agents can be used to "'replace"' services that are under development by other organizations, thus simulating the behavior of the whole application, even without all services. The MAS approach has other advantages: the simulator can be evolved along the project in order to support all project phases; agents can be used to perform non-functional requirements testing, and; agents can be used to generate reports.

III. Testing the LRIT Data Center

In 2006, the Maritime Safety Committee created the LRIT system to allow the long-range identification and tracking of ships. Tracking of any applicable ship begins with LRIT information being transmitted from the shipborne equipment. The LRIT information transmitted includes the ship's position, time and identification. The LRIT system consists of the following components (fig. 1) [12]:

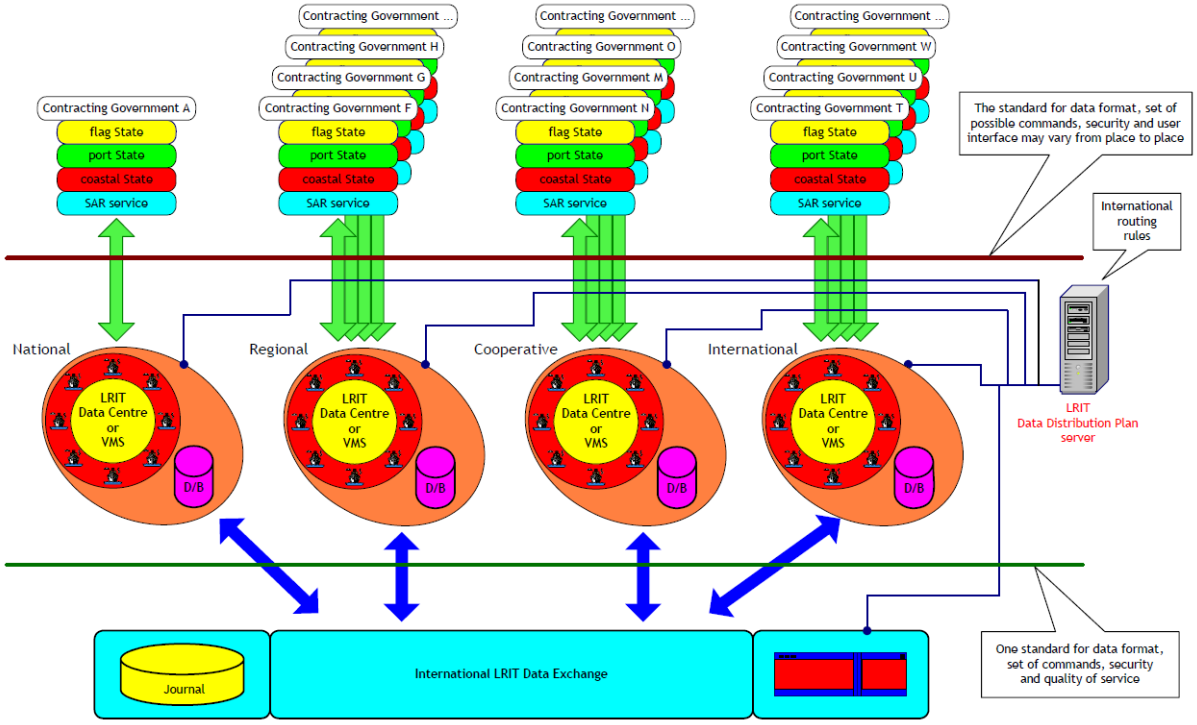

Fig. 1. The LRIT system architecture

Application Service Provider (ASP): the ASP receives the LRIT information from the ship, adds additional information to the LRIT message and passes along the expanded message to its associated Data Center. It provides the functionality required for the programming and communicating of commands to the shipborne equipment.

Data Center (DC): the DC should store all incoming LRIT information from ships instructed by their administrations to transmit LRIT information to that DC. DCs disseminate LRIT information to LRIT data users according to the LRIT Data Distribution Plan (DDP). The DCs process all LRIT messages to and from the International Data Exchange. A DC can provide services to one or more contracting governments.

Data Distribution Plan Server (DDP Server): the DDP contains the information required by the DCs for determining how LRIT information is distributed to the various contracting governments. The DDP contains information such as standing orders from contracting governments and geographical polygons relating to contracting governments' coastal waters.

International Data Exchange (IDE): the IDE routes the message to the appropriate DC based upon the address in the message and the URL/URI in the DDP server. The IDE neither processes nor stores the information contained within LRIT messages.

The Brazilian DC is under development by the Brazilian Navy as a SOA application. To enable the adequate testing of the Brazilian DC services, MAS simulators were used. It is important to mention that the Brazilian DC services were developed independently (and without any coordination) from other LRIT components. Initially, interface simulators were built. They were passive and were used to simulate requests whose results were evaluated manually.

After the first few months of development, a large amount of data needed to be handled at runtime to allow business rules' testing. Business rules' testing involved querying ships that were sailing in real time, changing the frequency of information for individual ships and requests for archived data and all these actions, when performed, needs to consider the data distribution plan for the time when the information was generated. These tests required more advanced functionalities. More specifically, the simulator needed to simulate ships with sailing behavior in given maritime areas and to respond to requests from the Brazilian DC (tested SOA application), following the LRIT rules. Some of these ships needed to have erroneous behavior to verify the data validation of the application.

Thus, the simulator should have an active behavior to simulate the ships that were interacting with the application (e.g. DCs). These ships could change their behavior at runtime, following the communication protocol used by real ships.

IV. The Agent-Oriented Testing Simulator

The software agent abstraction is appropriate to handle the problems above agents can represent mobile objects in a georeferenced system. The ShipSim simulator was developed using the maritime domain knowledge acquired when developing the Dominium [13]. The ShipSim design uses agents to carry all the active behavior expected for an ASP and ships in the system. DC and SimShip interact exchanging SOAP messages, as described bellow:

SimShip receives SOAP messages from the DC through the ASPSim, which is a passive component that persists incoming SOAP messages in a table of the georeferenced database, used as a blackboard. An AspGateway agent was created to collect the requests sent to ASP from DC. This agent checks the data in blackboard, translates it to ACL [14], and sends it to the agent that simulates the requested ship in the MAS.

Each shipAgent performs the control of compliance with the LRIT communications protocol, responding as expected (or not, if required by a case test). The agent that is simulating a ship starts executing the expected behavior upon receiving a request. The main requests for a ship in the LRIT system are: poll position; change of frequency rate for sending position, and; requests to stop or restart the transmission of positions. Agents simulating ships can be updated to consider the ship's course and speed (fig. 2).

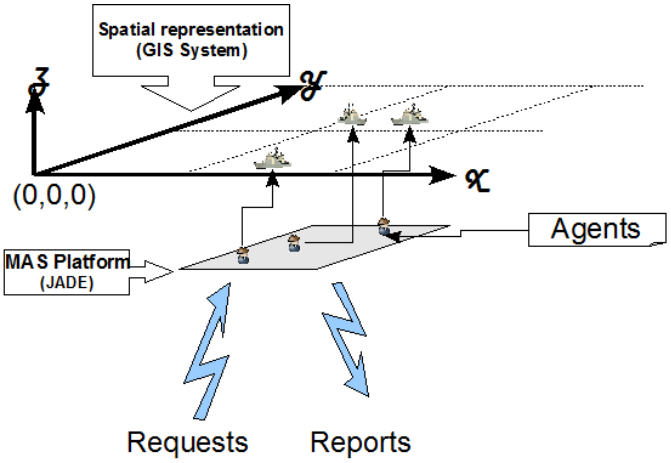

Fig. 2. Ship simulation in the ShipSim

These agents are also capable to control the start/stop and frequency to sending messages for all requestors countries (requests arrives through DC). To send a message to the DC, shipAgents connect directly a DC Web service. This architecture was designed to avoid an ACL to SOAP translation bottleneck. To test if this bottleneck was really overcome, it was necessary to perform tests like maximum number of responded requests per second and number of requests persisted per minute. After the test, the set of reports was used to estimate the maximum merchant fleet that can be supported by the Brazilian DC in the LRIT System. Some designed features for the ShipSim were not implemented at first, like reports and spreadsheets to be used in analyses. However, all features were developed and tested in later iterations.

The architecture used to run the tests is presented in fig. 3. The MAS simulator is the ShipSim, DC is the tested SOA application and the other components are passive interface

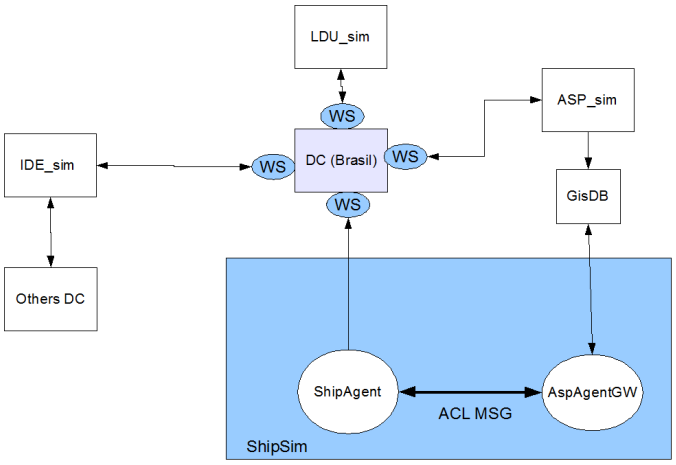

Fig. 3. The testing environment

simulators, implemented using EJB3, running in an application server. These passive simulators basically perform the XML schema validation and allow the exchange of messages using SOAP to the DC (the message is edited in an external application). Since October 2008, IMO had established a complete testing environment, with DCs, IDE and DDP server, however it is not possible test a DC without data from ships, so the ShipSim is still used to support test activities.

A. Desing and Implementation Details

The methodology presented in Nikraz et al. [15] was used for the testing simulator analyses and design. In this approach agent responsibilities and acquaintances are identified and later mapped to behaviors and communications. ASP and ship services were mapped as responsibilities and acquaintances in the early modeling, and after agent refinement only two types of agent were maintained: the ShipAgent and the ASPAgentGW.

The ASPAgentGW is a transducer agent, who monitors and performs queries on a SQL database, and translate the information to ShipAgent in ACL messages. The ShipAgent aggregate all other test functions: each agent instance simulate an individual ship and is responsible for maintaining a list of all received requests (one for each country) and for answering these requests in correct frequency, informing the current position. The position is calculated using the course and speed of the simulated ship, and those are altered when needed to avoid the exit of a limited area. When sending information, some data is collected, like successful delivery or fail, time to delivery and the sent information.

The testing simulator and the Brazilian DC SOA application were implemented using open source software. Besides, all LRIT standards are open. The simulator was originally developed over the SUSE enterprise version 10.2 operational system, and it is now migrating to version 11.1. The hardware used as runtime environment for the simulator was similar to the LRIT system production environment used by DC: HP servers with two processors quad core and 8 GB of RAM memory.

The ShipSim implementation was done using Java 6, JADE framework [16] , Eclipse platform, Java Topology Suite Library (JTS) [17], GIS database (PostgreSQL + PostGIS [18]). The Web services connection was created with the JEE eclipse plug-in, which uses Axis, and supports the SOAP 1.2 protocol. The simulator was modified along the testing to use cryptography with mutual certification over SSL.

The Brazilian DC SOA application was developed with a three layer architecture: the interface layer (for humans and external systems), based in Apache web servers; the application layer, using JEE2 technologies, deployed in a cluster of application servers, and; the persistence layer, using the Hibernate framework. To allow the test execution, the following variables were parameterized:

- square area where agents are created (defined by longitude and latitude limits);
- number of agents (i.e. ships) in simulation;
- time interval between two agents creation (milliseconds);
- interval for each agent sending messages;
- identifiers for agents (i.e an unique identifier);
- logical port to be used for each build of the simulator (this variable allows the initialization of several simulators simultaneously in the same machine);
- IP address for the tested DC (the project has more than one test environment);
- switch to off/on the cryptography (TSL 1.1 protocol with mutual authentication).

For simpler tests, simulations were executed directly from the Eclipse platform. Whenever it was necessary to place ships in different areas from around the world, a new instance of the simulator (with the specific set of ships) was instantiated. The DC non-functional requirements included information about performance.

The simulator was deployed in different servers, with quad core processors an 8GB of RAM memory. At this environment the simulation was executed with 20.000 agents simultaneously. The simulation RAM consume rate at this situation was 5.6GB. It is important to mention that the initial development effort for the ShipSim required 40 hours of a medium skilled developer, but with experience using JADE and theoretical knowledge about GIS and MAS. This developer has expertise in the LRIT system business rules and maritime environment.

The ShipSim had been modified many times during the LRIT system testing phase. These modifications included simple tasks, mainly because of the JADE architecture. Currently, the ShipSim is used by the Brazilian DC test group, who is responsible for its maintenance. The current ShipSim have four packages: agents; behaviors, objects and support. The simulation is started by the SimulationStarter class, responsible for to charge the jade container and all agents.

B. Sample Testing

This section exemplifies the use of the ShipSim, to show the provided test coverage. At late stages in the development of the LRIT system, seven DCs were incorporated to the system prototype. To allow the Brazilian DC test with them, five

simulators were instantiated to create virtual ships in the South Korea, USA, Canada, Bahamas and Liberia areas (following the polygons published in DDP server).

These simulators generated positions to mobile ships and sent them to the Brazilian DC to verify if the DC could handle a number of messages bigger than the specified in its requirements. All messages were forwarded to the proper DC, and these could make any requests to the ships, because they were active, i.e. they were simulating all behaviors expected for a real ship in the system. Therefore, almost all case tests provided in the IMO Testing Protocol were performed using the ShipSim support.

Some important metrics were obtained from the ShipSim, such as the maximum number of requests the system is capable to receive and persist in a queue per second; the maximum number of requests processed per second; the time the system was unavailable (denial of service), and; the maximum number of ships that the DC can safely support. It is important to stress that all these metrics can be obtained using testing frameworks, but the confidence level is different: using the simulators these metrics can be obtained in a condition near to observed by the real integrated SOA architecture. Figure 4 presents an example of testing simulation with 8000 ships flowing information in the LRIT System.

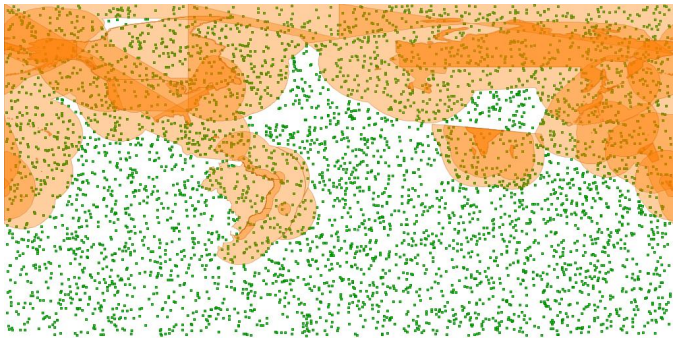

Fig. 4. 8000 ships simulation

V. Related Works

Canfora and Di Penta [11] present a survey on SOA testing. This work stressed the importance of this activity and it points out some key problems for test execution in SOA applications, especially when a service relies on other services to perform its functionality (late binding between components). None of the works cited in [11] presents a solution for testing a service with well-known interface (WSDL file) had interdependencies with other external (developed by other service providers) services. The proposed simulator approach deals with testing of incomplete systems, using MAS simulation to compose the overall system and then providing a reliable environment for system testing.

Frantzen et al. [19] present a Web services testing technique, using a Model-based testing approach. In this approach, Web services are tested as black-box components. This work points out the need for WSDL descriptions improvement since a WSDL file only describe the service interface, just like a method signature in Java. The work presents the Symbolic Transition System (STS - a state machine variant) for describing not only the interfaces, but also the executing sequence of a service in order to aid the testing activity. However, the technique focus on Web service testing, i.e. unit testing. It does not consider the specifics of service composition such as late binding, asynchronous communication, unexpected service usage and overall non-functional requirements testing. The proposed simulator approach deals with these issues. The internal logics of the testing stubs are implemented in the agents that can pro-actively interact with the SOA application, thus it does not require a more rigid state machine representation. Testing specifics such as asynchronous communication and unexpected service usage are treated by the simulator agents.

The work presented in [20] shows a discussion about the role of testing and monitoring SOA applications. Testing is considered a preventive activity, to be performed before application delivery (or before using the services). The work states that testing should go beyond monitoring - checking the correctness of the regular service usage - which will be done after the service has been executed. The monitoring could verify Service Level Agreements (SLA) or a specified Quality of Service (QoS). The work considers the need of using both testing and runtime monitoring to improve the confidence in developed SOA applications.

The proposed approach uses software agents to allow SOA application testing and monitoring. Agents can be used to test: a single service; the orchestration of services (i.e. perform integration testing) even when not all services are developed (i.e. an agent acting as a service stub, which is particularly important in inter-enterprise integration applications), and; other qualities (non-functional requirements). In the Brazilian DC system, for instance, the simulators were used to perform quality testing, such as the maximum number of connections, Server Application setting, memory management and code optimizations. This information is usually monitored during system execution, but the agents can verify them during the application development. In this sense, the confidence that the SOA application delivers its functionalities does not rely on post-deployment monitoring, but rather on application verification.

VI. Conclusions and Future Works

This paper presented an approach for SOA application testing using software agents. The work shows the results of a research on SOA testing that is ongoing for an year. The presented approach intends to allow testing during SOA application development through the use of MAS simulators. The approach deals with some SOA testing issues such as: lack of access to the source code of the parts to be integrated; lack of means of observation on system behavior (mainly due to system incompleteness); lack of control over the services implementation, and; the increasing difficulty and cost of SOA testing.

This paper also presented an experiment that describes a real project. The LRIT system is a distributed system and its components are being developed by different teams. The agent simulators were used to test the overall functionalities and qualities of the Brazilian DC (a LRIT sub-system, developed using the Web service technology by the Brazilian Navy).

The next steps in our work include the specification of a methodology to support agent-based simulator development for SOA application testing. We are seeking to develop a framework to aid the instantiation of simulators. Additionally, we are researching how agent capabilities, such as proactiveness, can help continuous integration activities.

REFERENCES

[1] C. Lau and A. Ryman, "Developing xml web services with websphere studio application developer," *IBM Systems Journal*, vol. 41, no. 2, pp. 178–197, 2002.

[2] S. Dustdar and S. Haslinger, "Testing of service oriented architectures: A practical approach," in *Proceedings of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (LNCS 3263)*, 2004.

[3] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005.

[4] J. Tian, *Software quality engineering: testing, quality assurance, and quantifiable improvement*. Wiley, 2005.

[5] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, *Testing and quality assurance for component-based software*. Artech House, 2003.

[6] G. Canfora and M. Di Penta, "Testing services and service-centric systems: Challenges and opportunities," *IT Professional*, vol. 8, pp. 10–17, 2006.

[7] W. T. Tsai, Z. Cao, X. Wei, P. Ray, Q. Huang, and X. Sun, "Modeling and simulation in service-oriented software development," *Simulation*, vol. 83, no. 1, pp. 7–32, 2007.

[8] S. Karnouskos and M. M. J. Tariq, "An agent-based simulation of soa-ready devices," in *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on*. Cambridge, UK: IEEE, 2008, pp. 330–335.

[9] M. J. Wooldridge, *An Introduction to MultiAgent Systems*. John Wiley & Sons, Inc., 2002.

[10] M. Wooldridge and N. Jennings, "Intelligent Agents: Theory and Practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, 1995.

[11] G. Canfora and M. Di Penta, "Service-oriented architectures testing: A survey," in *International Summer Schools, ISSSE 2006-2008, Revised Tutorial Lectures (LNCS 5413)*, ser. Software Engineering. Springer Berlin / Heidelberg, 2009, pp. 78–105.

[12] IMO, "Long range identification and tracking system: Technical documentation (part i)," International Maritime Organization (IMO), December 2008.

[13] P. Taranti and R. Choren, "Dominium: an approach to regulate agent societies in dynamic environments," in *Proceedings of the First International Workshop on Agent supported Cooperative Work (ACW) at ICDIM'07*, vol. 2. IEEE, 2007, pp. 811–816.

[14] FIPA, "Foundation for intelligent physical agents," http://www.fipa.org/, 2009.

[15] M. Nikraz, G. Caire, and P. A. Bahria, "A methodology for the analysis and design of multi-agent systems using jade," *International Journal of Computer Systems Science and Engineering*, vol. 21, no. 2, 2006.

[16] JADE, "Java agent development framework," http://jade.tilab.com, 2009.

[17] JTS, "Topology suite," http://www.vividsolutions.com/jts/jtshome.htm, 2009.

[18] PostGIS, http://postgis.refractions.net/, 2009.

[19] L. Frantzen, J. Tretmans, and R. de Vries, "Towards model-based testing ofweb services," in *Proceedings of the International Workshop on Web Services - Modeling and Testing (WS-MaTe 2006)*, 2006.

[20] G. Canfora and M. Di Penta, "Soa: Testing and self-cheking," in *Proceedings of the International Workshop on Web Services ? Modeling and Testing (WS-MaTe 2006)*, 2006.