DEVS Systems-Theory Framework for Reusable Testing of I/O Behaviors in Service Oriented Architectures

Xiaolin Hu[1], Bernard P. Zeigler[2], Moon Ho Hwang[2], Eddie Mak[2]

1 Georgia State University, Computer Science Department, Atlanta, GA 30303
2 University of Arizona, Arizona Center for Integrative Modeling and Simulation, Tucson, AZ 85721

The 2007 IEEE International Conference on Information Reuse and Integration, Las Vegas, NV, August 2007.

Abstract

This paper presents a framework for testing at the I/O behavior level in a service-oriented architecture (SOA) environment. The framework derives minimal testable I/O pairs from a service component's behavior specifications. These minimal testable I/O pairs are mapped to reusable primitives and then synthesized into test models in the Discrete Event System Specification (DEVS) formalism to meet different test objectives. We present the system theoretic foundation for I/O behavior testing and describe the inherently re-usable test development framework.

1. Introduction

Service-oriented architecture (SOA) is becoming one of the enabling technologies for the emerging large-scale net-centric systems, such as distributed supply chain management systems, networked crisis response systems, and the military Global Information Grid. This new paradigm brings new design and testing challenges in software development [1, 2]. Among them how to test the collaborations among multiple dynamically integrated services is becoming an increasingly important issue. Requirements for testing service collaborations include those derived from their decision-making, net-centric, large-scale, and dynamic behaviors. Two of the factors underlying the challenges posed by collaboration testing are: 1) the increasing complexity of service-oriented computing results in sophisticated interaction patterns among services. In particular, stateful web services, i.e., services that access and manipulate state and/or stateful resources in a timely fashion are widely deployed. A stateful service may implement a series of operations such that the result of one operation depends on a prior operation and/or prepares for a subsequent operation [8]. This results in temporal and logical dependencies between its input/output messages, which make the test difficult. 2) Service-oriented applications are by nature evolutionary and dynamical. This nature implies that supporting the sustainable long term evolution of service applications should play a central role in developing quality assurance and testing techniques and tools [6]. Service-oriented computing emphasizes service integration and composition where new services are dynamically discovered and integrated. This evolutionary nature calls for new test mechanisms such as "on-demand" test development to support this paradigm.

Software model is considered as a promising technology for supporting service-oriented computing in the open distributed Grid environment [9]. In the related work of applying modeling technology to testing web services and web applications, one focus has been on testing at the function level by exploiting the internal structure and/or behavior of a service component([4, 5]). Test plans are derived from data flow analyses of web applications and test agents are constructed to carry out these test plans [5]. Another focus is at the software component level. For example, [3] utilizes "validation agents" to automate the testing of software components. These validation agents depend on the "component aspects" that specify the functional and non-functional cross-cutting concerns of software components to construct and run automated tests. Test environments for large scale service collaborations are also developed. The Agentcities testbed [7] is an agent-based test environment developed to support large-scale dynamic service synthesis testing. In relation to the objectives of this paper, two research issues emphasized by current work are: 1) automating the construction of test models, and 2) deploying multiple models in test federations to collaborate in testing complex distributed interaction patterns. To achieve these requires a mature methodology based on a formal mathematical framework.

This paper presents a framework for developing test models for service test in a service-oriented computing environment. The test models (also referred to as test agents) are well-defined software components that can carry specific objectives to drive or to look for opportunities for test, have characteristics such as on-demand creation and self-destruction, and capability of working together for collaborative test. Within this context, this paper focuses on the aspect of automated model construction and its corresponding theoretical foundation for I/O behavior test. I/O behavior test is considered as a preferred approach in a service-oriented environment where services are perceived by their

interfaces and I/O behaviors. Systematic test of these behaviors, i.e., the logical and/or temporal relationships between sequences of inputs and outputs, can provide direct measurements of how well the service will interact with others. The I/O behavior test also removes the dependency on a service's internal representations that are typically inaccessible in a service-oriented environment. To carry out service test at the I/O behavior level, we take the systems theory point of view that an online service (e.g. software components, process models, or service organizations) is a system with I/O behaviors. Based on this foundation, we develop techniques to extract testable behaviors of the system/services under test (SUT). Testable behaviors are information exchanges among system/service components that are publicly observable over a network and do not require inspection into the components themselves. Development of such testable behaviors is guided by desiderata such as level of coverage of critical interactions, support for determination of appropriate measures of effectiveness (MOE) and measures of performance (MOP).

Model implementation and execution in this work is based on the DEVS (Discrete Event System Specification) [11] modeling and simulation/execution framework. DEVS is a formalism derived from generic dynamic systems theory. It has well-defined concepts of coupling of components, hierarchical, modular model construction, and an object-oriented substrate supporting repository reuse. There are two kinds of models in DEVS: atomic model and coupled model. An atomic model is a basic component. It has input/output ports to represent its interface, state transition functions, time advance function, and output function to specify its dynamic behavior. A coupled model tells how to couple several component models together to form a new model. This latter model can itself be employed as a component in a larger coupled model, thus giving rise to hierarchical construction. The event-driven nature and modular model construction of DEVS prove to be essential to develop test agents for testing at the I/O behavior level. More information about the DEVS formalisms and the DEVS modeling and simulation/execution framework can be found in [11].

2. System Specification and Testable I/O Behaviors

A service component at the I/O behavior level can be viewed as a system that receives inputs (messages) and generates outputs through its interfaces. These inputs/outputs and their logical/temporal relationships represent the I/O behavior of the system. From the design point of view, given such a system behavior specification, the developers' job is to realize a system to fulfill the required behavior; and the testers' job is to verify that the developed system conforms to the behavior that has been specified. This view of system design & verification based on system specification is rooted in systems theory [11] and serves as a theoretic foundation of this work. A major subject of systems theory deals with a hierarchy of system specifications [11] which defines levels at which a system may be known or specified. Among the most relevant to this work is the Level 2 specification, i.e., the I/O Function level specification, which specifies the collection of input/output pairs constituting the allowed behavior partitioned according to the initial state the system is in when the input is applied.

Although the hierarchy of system specifications provides a good framework for structuring the design and testing processes, a number of its aspects have to be extended in greater depth to enable it to support actual testing. One of these aspects is the development of a more compact form of the I/O Function Specification that is usable for development of automated test suites. Thus in this section we turn our discussion to this issue by first reviewing the concepts of input/output behavior and their relation to the internal system specification in greater depth. This will put us in position to understand the complexities that arise in specifying test sequences and approaches to their management.

2.1. Testable Form of I/O Specification

For a more in-depth consideration of input/output behavior, we start with the top of Figure 1 which illustrates an input/output (I/O) segment pair. The input segment represents message types X with content x and Y with content y arrive at times t_1 and t_2, respectively. Similarly, the output segment represents message type Z occurring twice with content z and z', at times t_3 and t_4, respectively. At the bottom of the figure we show a more compact representation of this same information, in which arrows above the time line represent inputs received and those below represent outputs sent.

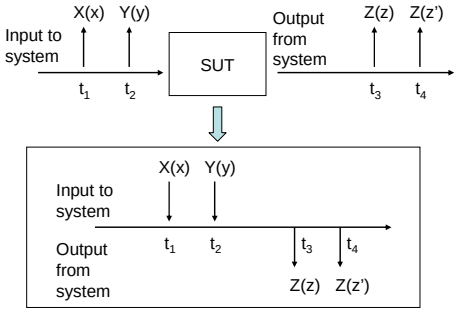

Figure 1: Representing an Input/Output Pair

To illustrate the specification of behavior at the I/O level we consider a simple system – an adder – all it does is adding values received on its input ports and transmitting their sum as output. Note that this example is used in several places in the paper to demonstrate the test framework. However simple this basic *adding* operation

is, there are still many possibilities to consider to characterize its I/O behavior such as which input values, (arriving at different times) are paired to produce an output value and the order in which the inputs must arrive to be placed in such a pairing. Figure 2 (left hand side) portrays two possibilities, each described as a DEVS model at the I/O System Level of the Specification Hierarchy. In a), after the first inputs of types X and Y have arrived, their values are saved and subsequent inputs of the respective types refresh these saved values. The output of message type Z is generated after the arrival of an input and its value is the sum of the saved values. In b), starting from the initial state, both types of messages must arrive before an output is generated (from their most recent values) and the system is reset to its initial state after the output is generated. This example shows that even for a simple function, such as adding two values, there can be considerable complexity involved in the specification of behavior when the temporal pattern of the messages bearing such values is considered. Two implications are immediate. One is that there may be considerable incompleteness and/or ambiguity in a semi-formal specification where explicit temporal considerations are often not made. The second implication follows from the first: an approach is desirable to represent the effects of timing in as unambiguous a manner as possible and in such a way, that a discriminating test suite can be derived from this representation. We outline such an approach in the following.

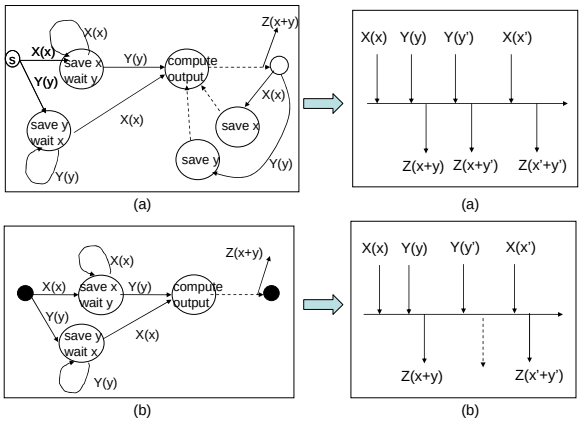

Figure 2**:** Variants of Behavior and Corresponding I/O Pairs

Figure 2 (right hand side) shows some I/O pairs that serve to distinguish between the triplet of behaviors described above. In a), after initial arrivals of message types X and Y, successive arrivals of either message type, X or Y, results in an output of type Z with values shown. In b), however, a second Y message in a row does not result in an output, the latter will be produced when the next arrival is an X message. A dashed arrow is used to explicitly indicate that an output should *not* be generated. Note that these patterns are discernable at the Input/Output behavior level and form the basis for testing where access to the SUT is through input/output interfaces. Development of these patterns follows from understanding and manipulation of the state-based I/O System Level specification.

2.2. Testable Form of I/O Specification

An I/O segment pair may have any finite number of input messages and output messages in its interval of definition. It is much easier to deal with segments of limited complexity and synthesize tests from such segments. Thus we introduce the concept of *minimal testable I/O pair*. This is an I/O pair in which the output segment has at most one event at the end of the segment. Such segments are illustrated in Figure 3. We can easily extract such pairs from an I/O specification at Level 2, by going from a given state to the next one that results in an output event, because each input segment produces a unique output segment if the initial state is given. Figures 3a) and 3c) illustrate this process for the corresponding variants given in Figure 2.

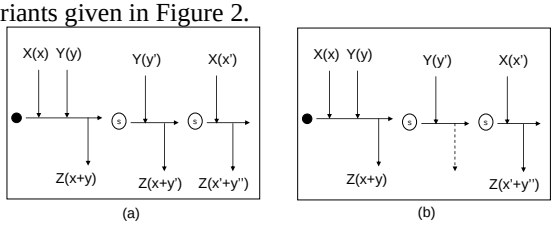

Figure 3: Some Minimal Testable Input/Output Pairs

An I/O Specification at Level 2 for which all pairs are minimally testable is said to be a *minimal testable* I/O *representation*. A minimal testable I/O representation can be captured by the System Entity Structure (SES) [11] and documented in XML format. More details about this can be found in [10]. The concept of minimal testable I/O pairs and their representation in SES and XML provide the foundation for automated construction of test models. The motivation to do this is that each minimal I/O pair can be mapped directly into a DEVS test model, which can then be coupled together to represent a test scenario. Note that time deadlines can also be incorporated and captured by the DEVE model. We can go from an I/O Behavior Specification at Level 2, and in particular from its minimal testable representation, to a test model that interacts with a SUT to assess whether it satisfies the specification. The test models can be synthesized from the segments because each I/O pair has a finite number of input messages and output message in its interval.

3. The Test Development Framework

3.1. Overview of the Design Process

One of the main goals of this work is to support automated model construction from system behavior

specification. To support this, a design process has been developed. This process starts from the system behavior specification of the SUT and goes through several stages, where each stage is dependent on the artifacts from its previous stage and work on them to generate new models. This process ends when test agents for specific test scenarios are created.

The first step of this process is to extract minimal testable I/O pairs from the system behavior specification of the SUT. The second step is to sequence such I/O pairs into test scenarios (see examples in Figure 2(B)) to be embodied by test agents. The third step is to apply a model mapping concept to implement the minimal testable I/O pairs that are included in a test scenario as DEVS primitives. The DEVS primitives are simple atomic models such as: *processDetect*, *waitReceive*, and *waitNotReceive* that will be described in section 3.3. These primitives act as the building blocks to form *basic test models* corresponding to the minimal testable I/O pairs. In the fourth step, the basic test models are coupled together according to the test scenarios and their compositions become test agents with specific characteristics such as autonomy, modularity, on-demand creation and self-destruction. Finally, the created test agents are deployed in test federations to inject or observe inputs sent to, and outputs emitted by, the SUT. Given the underlying DEVS framework, test models can be stored in repositories for subsequent reuse.

3.2. Controlled and Opportunistic Testing

Before delving into the details of model construction, we describe the two types of test that this test framework intends to support: controlled test and opportunistic test. Figure 4 uses an example scenario that includes one minimal testable I/O pair to illustrate the interactions between a test agent and the SUT for controlled and opportunistic tests respectively. This minimal testable I/O pair has inputs *Mx1(data1)*, *Mx2(data2)* and output *Mx3(data3)*, denoted as <*Mx1(data1)*, *Mx2(data2)*→ *Mx3(data3)*>, and is shown by the gray box in the figure.

In the controlled test as shown in Figure 4(a), the SUT is coupled solely to the test agent, which fully controls the test scenario by sending inputs to the SUT and checking if the output responses of the SUT are correct. The sequence of inputs injected by the test agent and the kind of output that the test agent expects are derived from the test scenario of the SUT. Specifically, based on the minimal testable I/O pair <*Mx1(data1)*, *Mx2(data2)*→ *Mx3(data3)*>, the test agent first generates message type *Mx1* with value *data1*, then message type *Mx2* with value *data2* and sends these to the SUT in timed, sequential order. After that, the model waits for the output generated by the SUT and checks if it is *Mx3* with value *data3*. The test agent issues a Pass or Fail for this test scenario according to whether *Mx3(data3)* was received within an allowed time window. The controlled test is typically conducted before a system is deployed to its executing environment, because the test agent assumes full control of the SUT.

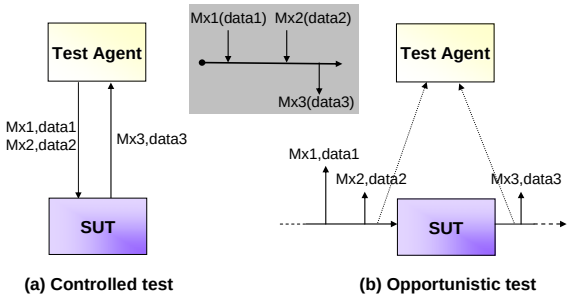

Figure 4. Controlled Test and Opportunistic Test

In the opportunistic test (Figure 4(b)), a test agent does not control the test scenario. Instead, it observes the I/O behaviors of an operating SUT and looks for opportunities to carry out the test. Thus in opportunistic test a test agent, also called *test detector*, is only loosely coupled to the SUT. Opportunistic test can be employed after a SUT is deployed to its executing environment where it interacts with other systems/services. An opportunistic test agent, once deployed, will start to observe the inputs and outputs of the SUT and activate a test when its conditions are met. For example, the test agent (detector) in Figure 4(b) is setup to test the I/O pair <*Mx1(data1), Mx2(data2)*→ *Mx3(data3)*>. If this agent observes input messages *Mx1(data1)* and *Mx2(data2)* in the right timed, sequential order (this means the agent's test condition is met), it will check if the right output *Mx3(data3)* will be generated by the SUT within the allowed time. Otherwise, e.g., if the agent notices the first input message is *Mx1(data1)* but the following message is not *Mx2(data2)*, the agent will withdraw the test by reinitializing itself and continue to look for other opportunities to conduct the test.

The opportunistic test mode has the advantage that the test agents do not interfere with the operation of the SUT. This loose coupling nature makes it possible to dynamically create and deploy multiple agents in cooperation to test (and/or monitor) the operation of a distributed system, where each agent test part of the system and they collaborate to detect errors at the system level, e.g., detecting an illegal sequence of interaction, or detecting a lost message in an unstable network. Multiple agents can also be deployed to the same SUT in parallel with each agent having its unique objective to test a specific aspect of I/O behavior. For example, one agent can test if the SUT generates the correct output for a given sequence of inputs while another agent can test if a particular deadline is met. This capability of dynamically create and deploy multiple agents for collaborative testing/monitoring is especially useful in a service-

oriented environment where services are integrated dynamically and evolve continuously.

3.3. Test Model Construction

The functional difference of test agents in controlled and opportunistic tests implies that the building blocks of these agents are also different. In controlled test, an agent needs to send inputs to the SUT and wait for output from the SUT. Thus the agent is constructed from primitives: *holdSend*, *waitReceive*, and *waitNotReceive*. In opportunistic test, an agent needs to detect inputs sent to the SUT and wait for output from the SUT. Thus the agent is constructed from primitives: *processDetect*, *waitReceive*, and *waitNotReceive*. Despite this, both controlled and opportunistic test modes share the same underlying design process, i.e., mapping minimal testable I/O pairs to test primitives and then automatically synthesizing models into test agents. Below we describe how a test agent in opportunistic test mode is constructed. The description can be easily adapted to cover the case of controlled test mode.

As presented in section 3.1, the starting point of model construction is the minimal testable I/O pairs derived from the system behavior specification of the SUT. According to a minimal testable I/O pair, a test agent (detector) needs to watch for messages arriving to, and departing from, the SUT. This can be realized by three atomic models (also referred to as primitives): *processDetect*, *waitReceive*, and *waitNotReceive*. A *processDetect* primitive waits for a message to arrive to the SUT within a prescribed interval. A *waitReceive* primitive waits for a SUT response within a pre-defined time interval, and determines the pass-fail condition by comparing it with the expected response. On the other hand, a *waitNotReceive* primitive watches for messages that are not supposed to be sent by the SUT. This is for the case of the I/O minimal test pair that does not end in an output (see an example in Figure 3(b)). This primitive holds for a pre-defined time interval and determines failure if the SUT produces a message on the watched list in this interval. Each primitive is implemented as a DEVS atomic model. To give an example, Figure 5 shows the DEVS implementation for the *waitReceive* primitive. The *waitReceive* primitive has two input ports: *start*, and *In_msg*, and one output port: *pass*. The behavior of the model is described as follows. The model begins at the "Passive" state. When it receives a start event, the external transition function dictates a new state, "Wait," with the wait time provided by the constructor. If the model receives a message event before the specified time expires, it compares the received message with the message that it expects. If they are identical, it will log a successful outcome, and the state will change to "Success", where the output function will generate a "pass" message, and then the internal transition function will change the model's state to "Passive". Otherwise, a failure will be logged and the state will change to "Passive." Implementations of *processDetect* and *waitNotReceive* primitives follow the same pattern. More details can be found in [10].

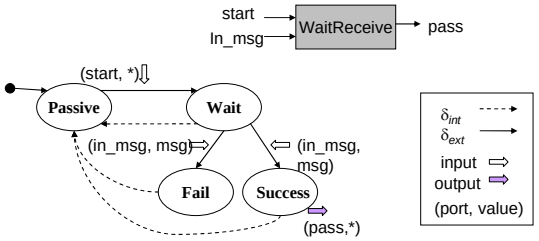

Figure 5: The *WaitReceive* test primitive

The three primitives described above act as the building blocks to compose basic test models according to the minimal testable I/O pairs. Specifically, for each minimal testable I/O pair, a basic test model is constructed by coupling one or more *processDetect* primitives in sequential order corresponding to the inputs, and then one *waitReceive* (or *waitNotReceive*) primitive corresponding to the output. Figure 6 illustrates two basic test models constructed from these primitives. In this figure, the behavior of the SUT is described by the first two minimal I/O pairs shown in Figure 3(a). Corresponding to the first minimal testable I/O pair <*X(x), Y(y)→ Z(x+y)*>, the initial *processDetect(X)* watches for the arrival of a value on port X; when such a value, x arrives, it sends a start message containing x to the second *processDetect(Y)* which looks for a value coming on the Y port. When such a value, y arrives it sends the pair of values x, y to the *waitReceive(Z)*, which waits for a value on the Z port. When such a value, z arrives it compares it to x+y and emits a pass or fail decision accordingly. Note that communication from one primitive to the next establishes the context or information needed to maintain the state information needed for the SUT. This first level composition of primitives is called a *basic test model*.

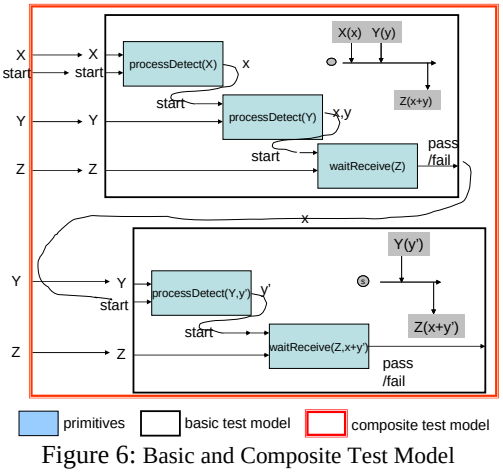

Figure 6: Basic and Composite Test Model

When the basic test models derived from the minimal testable I/O pairs are cascaded together, the second level composition is called a *composite test model* and implements a test scenario (defined by one or more minimal testable pairs in sequential order). The basic and composite test models are triggered by a start event through the *start* port. When the test of one basic test model is finished, it will trigger another basic test model. For example, the basic test model shown in the lower part of Figure 6 is for the I/O minimal segment shown which checks for arrival of a message on the Y port subsequent to processing from the previous state. This model will be triggered by the first basic test model (shown in the top part of Figure 6) and start to watch for the Y message. When receiving a value y' it sends the pair (x, y') to the *waitReceive(Z)* which operates as just described. Consider now the SUT described by Figure 3(b). Here the second basic model in the cascade is a *waitNotReceive (Z)*, rather than a *waitReceive (Z)*, since the SUT is not supposed to respond unless both inputs are refreshed. We can take this idea further by coupling sets of composite test models together in hierarchical fashion to form complex test scenarios. We refer to such a test model (either a composite model or a set of hierarchically coupled composite models) as a test agent.

It is important to automate the transformation from the minimal testable pairs to the test models. Such automated model transformation also provides traceability support between the required SUT behavior and the DEVS test models. In our approach, both the minimal testable I/O pairs and test model structures are described in SES and written in XML format. Transformation from minimal testable pair to test model is based on the mapping relationship between the corresponding nodes in the two models. For example, the initial state entity in a minimal testable I/O pair is mapped to the initial state requirement variable of the *BasicTestModel* entity. The *InputMessages* entity is mapped to the *processDetect_Components* entity. The *OutputMessage* entity is mapped to either *waitReceive_Components* or *waitNotReceive_Components* depending on whether it is required to be, or not to be, produced, respectively. Details can be found in [10].

4. Conclusions

This paper presents a theoretical foundation and a test development framework for service test at the I/O behavior level in a service-oriented computing environment. It supports automatically constructing test models and then composing them for complex test scenarios. The test framework is being developed in the context of testing service collaborations on net-centric implementations of service oriented architectures [10, 12]. The correctness and effectiveness of these collaborations are heavily dependent on the dynamic behaviors of services and service organizations. The model-based test framework presented in this paper provides a foundation for developing rigorous models and methodologies for automatic and dynamic behavior test of such collaborations. Application of the framework to distributed testing of SOA collaborations at multiple levels (syntactic, semantic, pragmatic) simultaneously is discussed in [13].

5. References

1 Manes, A.T. 2005, *VantagePoint 2005-2006 SOA Reality Check Version*: 1.0, Burton Group Publication, Jun 29

2 Hull R., and Su, J., Tools for Composite Web Services: A Short Overview, *ACM SIGMOD Record*, Vol 34, No. 2, June, 2005

3 Grundy, J.C., Ding, G., and Hosking, J.G., Deployed Software Component Testing using Dynamic Validation Agents, *Journal of Systems and Software: Special Issue on Automated Component-based Software Engineering*, vol. 74, no. 1, January 2005, Elsevier, pp. 5-14

4 Qi, Y., Kung, D., Wong, E., An Agent-Based Testing Approach for Web Applications, *compsac*, pp. 45-50, 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 2, 2005

5 Huo, Q., Zhu, H., Greenwood, S., A Multi-Model Software Environment for Testing Web-based Applications, *compsac*, p. 210, 27th Annual International Computer Software and Applications Conference, 2003

6 Zhu, H., Cooperative Model Approach to Quality Assurance and Testing Web Software, *compsac*, pp. 110-113, 28th Annual International Computer Software and Applications Conference (COMPSAC'04), 2004

7 Willmott, S., Thompson, S., Bonnefoy, D., Charlton, P., Constantinescu, I., Dale, J., Zhang, T., Model Based Dynamic Service Synthesis in Large-Scale Open Environments: Experiences from the Agentcities Testbed, pp. 1318-1319, AAMAS'04, 2004

8 Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T. and Weerawaranna, S. *Modeling Stateful Resources with Web Services*. Globus Alliance, 2004

9 Foster, I., Jennings, N. R., and Kesselman, C., Brain meets brawn: Why grid and agents need each other. *Autonomous Agents and Multi-Agent Systems (AAMAS'04)*, 2004

10 Mak, E., *Automated Testing using XML and DEVS*, Thesis, University of Arizona, http://www.acims.arizona.edu/PUBLICATIONS/PDF/Thesis_EMak.pdf

11 Zeigler, B.P., Kim, T.G., and Praehofer, H.,: *Theory of Modeling and Simulation.* 2 ed. 2000, New York, NY: Academic Press

12 Zeigler, B.P., Fulton, D., Hammonds P., and Nutaro, J. Framework for M&S–Based System Development and Testing In a Net-Centric Environment, *ITEA Journal of Test and Evaluation*, Vol. 26, No. 3, 21-34, 20

13 Zeigler, B.P., and P. Hammonds, "Modeling&Simulation-Based Data Engineering: Introducing Pragmatics into Ontologies for Net-Centric Information Exchange", in press, New York, NY: Academic Press.