Specification, Validation and Run-time Monitoring of SOA Based System-of-Systems Temporal Behaviors

Thomas S. Cook
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
tscoo1@nps.edu

Doron Drusinksy
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
and Time Rover, Inc.
Cupertino, CA, USA
ddrusins@nps.edu, www.time-rover.com

Man-Tak Shing
Computer Science Department
Naval Postgraduate School
Monterey, CA, USA
shing@nps.edu

Abstract - *This paper is concerned with the correct specification and validation of temporal behaviors in a Service-Oriented Architecture based system-of-systems. It presents a new formalism, called Message Sequence Chart Assertions (MSC-Assertions), for the specification of global system behaviors, and describes a specification validation technique using scenario simulation based on the JUnit Test Framework. We also describe the armor-plating of system-of-systems using runtime execution monitoring of MSC-Assertions.*

Keywords: Message Sequence Chart (MSC) Assertions, JUnit testing, formal specification, run-time execution monitoring, validation, verification, service-oriented architecture, web services.

1 Introduction

Large systems-of-systems (SoSes) are typically made up of a federation of existing systems and developing systems interacting with each other over a network to provide an enhanced capability greater than that of any of the individual systems within the system-of-systems. Service-oriented architecture (SOA) and the supporting Web Services (WS) technology hold promise to create SoSes that are interoperable, composable, extensible, and dynamically reconfigurable.

A SOA, in the most generic sense, is an architecture that supports the discovery of, binding to, and execution of some resource (service) or composition of resources (services) on a network. Services are applications designed to be reusable, loosely coupled, composable, autonomous, stateless, discoverable, defined by a formal contract, and to conceal underlying logic. As the supporting technology evolves, so too does the concept of a SOA. Today, many large organizations, including the US Department of Defense, are thinking and acting in accordance with the SOA definition found in [10], where SOA is defined as "an architecture that promotes service-orientation through the use of Web Services".

The increasing use of SOA and WS for complex system-of-systems has raised concerns regarding the correctness and trustworthiness[1] of the SOA-based SoSes [2]. At the highest abstract level, the functionality of a SOA-based SoS can be viewed as a set of Web Services. A Web Service has two components: a contract defining its external behavior from the clients' point of view, and a business process describing its internal logic via the coordination and composition of other Web Services. The specification of these complex business processes is error prone due to concurrency in activities execution, possibility of communication delay and error, as well as faults in the remote service providers. Often, even if the specifications are correct for an ideal operating environment, they do not operate as well in a less than ideal situation, thus render the resultant SOA-based SoS untrustworthy.

Many researchers proposed to enhance the trustworthiness of the Web Services via formal specification and verification of their business processes [3, 4, 11-14, 16], using light-weight formal methods consisting of the following four steps:

Step 1. Specify the business process in some semi-formal languages (e.g. the Business Processing Execution Language (BPEL));

Step 2. Translate the specifications into formal models (e.g. Linear temporal logic, state machines, Petri nets, process algebras);

Step 3. Specify the desirable functional and non-functional properties of the business process as formal assertions;

Step 4. Verify the formal business process models against the properties using theorem proofing, model checking, or specification-based testing.

[1] The term trustworthy software is synonymous with dependable software and hence can be expressed as a composite vector of the software's dependability attributes that include *availability*, *reliability*, *safety*, *security* (*confidentiality* and *integrity*), and *maintainability*.

However, for the aforementioned methods to effectively produce trustworthy SoSes, we must address the following concerns:

(1) We need to validate the accuracy and the correctness of the formal-assertion representation of the mission-essential and safety-critical properties of the SoSes. Otherwise, we risk wasting our efforts in verifying the Business Process Models against the wrong set of properties. To address this concern, we present a new formalism and an iterative process for engineers to create and validate system behavior assertions based on Use Case scenarios.

(2) Even if we can correctly capture and specify the system properties, and successfully verify the Business Process models against these properties, there is no guarantee that the SoS will behave as expected during run-time due to possible communication delays and service provider failures. To address this concern, we propose to combine run-time execution monitoring and assertion checking as a means to armor-plate the SOA-based SoSes.

The rest of the paper is organized as follows. Section 2 presents Message Sequence Chart Assertion for specifying distributed system behaviors. Section 3 describes the process for the development and validation of MSC Assertions. Section 4 illustrates armor-plating the SoSes using MSC Assertions and Run-Time Execution Monitoring. Section 5 presents a discussion on the approach and draws the conclusion.

2 Message Sequence Chart Assertions

MSC Assertions are a formal-language extension of UML MSC's superimposed with UML statecharts. They have the look and feel of UML MSC's and UML statecharts, yet they are formal and executable. For example, unlike UML MSC's, MSC-Assertions are capable of making a distinction between events that *can* occur and those that *must* occur. In addition, MSC Assertions are capable of specifying infinite sets of scenarios.

MSC Assertions are based on Statechart diagrams superimposed on MSC diagrams and augmented with Java (or C++) conditions and actions. For example, Figure 1 shows the MSC Assertion for a time-bound requirement of a travel agent service: "*R1: The travel agent must obtain bids from at least two airlines and two hotels and return a flight and a hotel matching the customer's request within 30 seconds from the time the customer issues his travel request*".

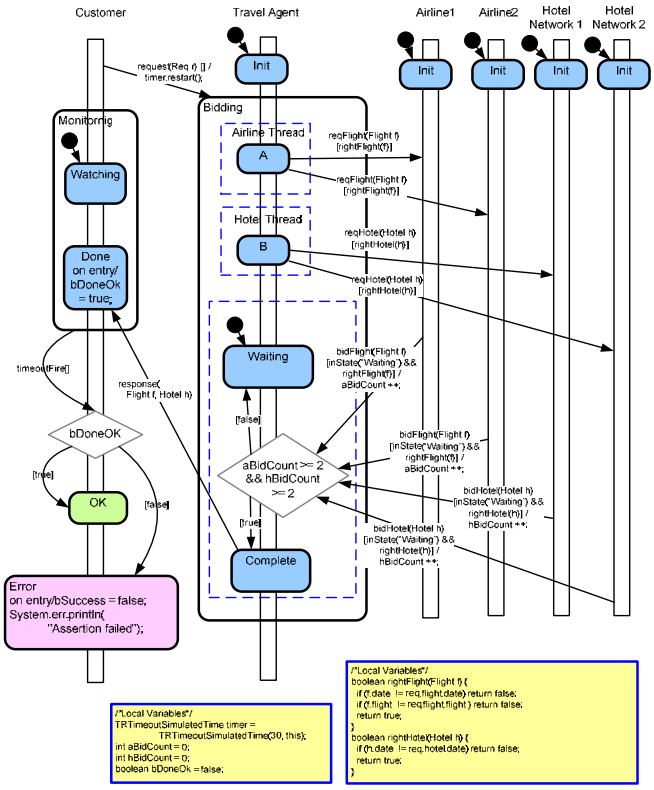

Figure 1. A MSC Assertion for the Travel Agent Service

The MSC-Assertion of Figure 1 looks, for the most part, like a UML MSC, but it enjoys the following unique features:

1. An MSC Assertion is written from the standpoint of an *observer*, and can be used for run-time monitoring of the target application. Consider for example the message *reqFlight(Flight f)* sent from the Travel Agent to Airline #1. While a UML-MSC might consider an interpretation where this event is *generated* by the Travel Agent, for an MSC Assertion, it is meant that the MSC Assertion should monitor-for, or listen-for, this event flowing from Travel Agent to Airline #1. Note that while the Travel Agent service may send out many requests for bids, the MSC Assertion only needs to observe two of such requests to satisfy the requirement R1.
2. An MSC Assertion *allows loops and transitions back up the vertical task bar*. In Figure 1 for example, the Travel Agent will return to the *Waiting* state if the condition $aBidCount \geq 2$ && $hBidCount \geq 2$ is false. This feature is in contrast to UML MSC's where a vertical task bar represents a *timeline* and where clearly a task cannot move back in time. An MSC Assertion however, considers a vertical task bar as a *progression of states*, like a state diagram drawn vertically. It therefore permits loops.
3. *States and actions.* As discussed above and as illustrated in Figure 1, an MSC Assertion task might contain both implicit and explicit states. The purpose

of explicit states is to specify actions, which are code snippets (written in Java or C++, depending on the code generator chosen) to be performed, such as *aBidCount++* or *rightFlight(Flight h)*. For example, the Customer will remain in its implicit initial state until the event *request(Req r)* is observed leaving the *Customer.* The Customer then enters the *Monitoring* state. The Customer will remain in *Watching* state until either the event *response(Flight f, Hotel h)* is observed arriving at the *Done* state, or the timeout event is detected.

4. *Java/C++ underlying language and code generation*. An MSC Assertion is a diagrammatic representation of a Java or C++ class that implements the requirement as a monitor. Hence, all variables and functions declared in the local-variables boxes of Figure 1 are actually properties of this generated class.
5. *Parameterized events*. An MSC Assertion event can contain objects as actual parameters. In Figure 1, the transition annotated with the message *bidFlight(Flight f)*, from Airline #1 to the Travel Agent, is sent with some *Flight* object as an argument. Condition guards range over local properties and event arguments (e.g., *rightFlight(f)*).
6. *An MSC Assertion is an assertion*. It uses the same approach described in [6] for assertion statecharts where it announces a success or failure for every witnessed input scenario. It does so using the built-in *bSuccess* property. The boolean *bSuccess* is true by default. The developer assigns *bSuccess=false* as an action wherever s/he wants the assertion to fail. The JUnit test-case then inspects this property to decide whether a particular test-run failed or not.

Figure 1 realized requirement R1 as follows. First note that, in the style of the UML-MSC notation, the assertion contains six tasks, denoted by the six vertical task bars. Also, the assertion contains local variables, *timer, aBidCount, hBidCount* and *bDoneOK,* as well as two Boolean functions *rightFlight()* and *rightHotel()* for checking the correctness of the itinerary. The MSC Assertion monitoring starts as a *request(Req r)* event is observed from the Customer task to the Travel Agent task while the Customer task is in its implicit inital state. The 30 second timer is triggered and the Customer task enters its *Watching* state. The Customer will remain in *Watching* state until either the event *response(Flight f, Hotel h)* from the Travel Agent task (while the latter is in its *Complete* state) or the timeout event is detected. If the Customer receives the *response()* message before the timeout event, it will enter the *Done* state. *bDoneOK* will be set to true and the subsequent timeout event will cause the Customer task to enter the *OK* final state. If the Customer task does not receive the *response()* message before the timeout event, *bDoneOK* will remain false and the timeout event will cause the Customer task to enter the *Error* final state; *bSuccess* will be set to false indicating the violation of the requirement.

The Travel Agent task will remain in its *Init* state until it receives the event *request(Req r)*, then it will transition to the *Bidding* state. The *Bidding* state consists of three concurrent threads, in the style of the UML statechart threads [6]. The Travel Agent task will remain in the *Waiting* state until it has received at least two airline bids and two hotel bids. It will then transition to the *Complete* state where the MSC Assertion is ready to observe the event *response(Flight f, Hotel h)* from the Travel Agent task to the Customer task. Clearly, the Travel Agent task must ensure that the bids received indeed satisfy the customer's request. This constraint is manifested as a condition guard *righFlight(f)* or *rightHotel(h)* on the message transition (in other words, MSC Assertion message transitions have the same *event[guard]/action* look and feel as UML-statechart transitions).

Since we are not interested in the detailed temporal behavior of the Airline tasks and Hotel Network tasks in the requirement R1, we treat these tasks as black boxes. The MSC Assertion only wants to observe the fact that each of these tasks returns a bid to the Travel Agent task only after they have received a request for bid from the Travel Agent task as follows. Each of these four tasks remains in its *Init* state until it receives the request for bid message from the Travel Agent task. It then enters its implicit working state. It will transition from its working state to its implicit terminal state when the MSC Assertion observes that the task returns a bid to the Travel Agent task.

3 Validation of Message Sequence Chart Assertions

It is important to validate the correctness of the assertions early in the software development process. Unfortunately, users often discover, late in the development process, that their assertions are incorrect and do not work as intended. Possible reasons for incorrect assertions are listed in Table 1 below.

Table 1. Possible reasons for assertion failure

Case	Reason
1.	Incorrect translation of the natural language specification to a formal specification.
2.	Incorrect translation of the requirement, as understood by the modeler, to natural language.
3.	Incorrect cognitive understanding of the requirement. This situation typically occurs when the requirement was driven from the use case's main success scenario, with insufficient investigation of other scenarios.

In [8], we propose the following iterative process for assertion development (Figure 2).

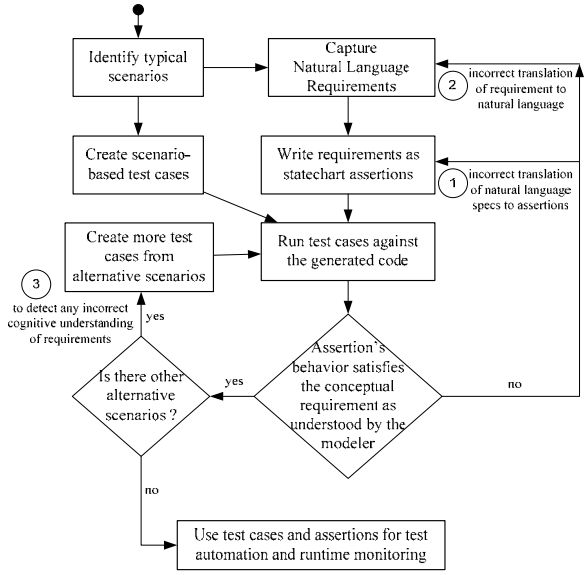

Figure 2. Iterative process for assertion development

Our methodology, as presented in [6-8], is that formal requirements ought to be simulated to assure that the cognitive understanding of the requirement matches the formal specification. To that end, we developed a run-time monitor for MSC Assertions that is fully integrated with the popular JUnit testing framework [1], and created a set of scenarios, using the JUnit testing framework. The scenarios test the MSC Assertion to assure that it announces a success if-and-only if the travel agent returns an acceptable itinerary to the Customer after requesting and obtaining at least two airline bids and at least two hotel bids matching the Customer's request in the prescribed time limit. For example, the following hand-code test case describes a scenario in which the Travel Agent successfully completes its service 25 seconds after receiving the request from the Customer.

```
import junit.framework.*;

public class TestMSC_Assertion extends TestCase {
  private MSC_Assertion mSC_Assertion = null;

  protected void setUp() throws Exception {
    super.setUp();
    mSC_Assertion = new MSC_Assertion();
  }

  protected void tearDown() throws Exception {
    mSC_Assertion = null;
    super.tearDown();
  }

  // Test Scenario 1
  public void testExecTReventDispatcher() {
    Flight f = new Flight(3, 1);
    Hotel h = new Hotel(3);
    Req req = new Req(f, h);
    // send request from Customer to Travel Agent
    mSC_Assertion.request(req,
             new Integer(MSC_Assertion.CUST),
             new Integer(MSC_Assertion.TA));
    // send request for Airline bids
    mSC_Assertion.reqFlight(f,
             new Integer(MSC_Assertion.TA),
             new Integer(MSC_Assertion.AL1));
    mSC_Assertion.reqFlight(f,
             new Integer(MSC_Assertion.TA),
             new Integer(MSC_Assertion.AL2));
    // send request for Hotel bids
    mSC_Assertion.reqHotel(h,
             new Integer(MSC_Assertion.TA),
             new Integer(MSC_Assertion.H1));
    // receive bid from Airline 1
    mSC_Assertion.bidFlight(f,
             new Integer(MSC_Assertion.AL1),
             new Integer(MSC_Assertion.TA));
    // send request for second Hotel bid
    mSC_Assertion.reqHotel(h,
             new Integer(MSC_Assertion.TA),
             new Integer(MSC_Assertion.H2));
    // advance clock by 25 seconds
    mSC_Assertion.incrTime(25);
    // receive bids from Hotel 2
    mSC_Assertion.bidHotel(h,
             new Integer(MSC_Assertion.H2),
             new Integer(MSC_Assertion.TA));
    // receive bid from Airline 2
    mSC_Assertion.bidFlight(f,
             new Integer(MSC_Assertion.AL2),
             new Integer(MSC_Assertion.TA));
    // receive bids from Hotel 1
    mSC_Assertion.bidHotel(h,
             new Integer(MSC_Assertion.H1),
             new Integer(MSC_Assertion.TA));
    // send response to Customer
    mSC_Assertion.response(f, h,
             new Integer(MSC_Assertion.TA),
             new Integer(MSC_Assertion.CUST));
    // advanced clock by 10 seconds
     mSC_Assertion.incrTime(10);
     assertTrue(mSC_Assertion.isSuccess());
   }
}
```

To test the correctness of the assertion, we created two more scenarios by replacing the body of the *testExecTReventDispatcher()* method with the following code.

```
// Test Scenario 2
public void testExecTReventDispatcher() {
  Flight f = new Flight(3, 1);
  Hotel h = new Hotel(3);
  Req req = new Req(f, h);
  mSC_Assertion.request(req,
           new Integer(MSC_Assertion.CUST),
           new Integer(MSC_Assertion.TA));
  mSC_Assertion.reqFlight(f,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.AL1));
  mSC_Assertion.incrTime(35);
  mSC_Assertion.reqHotel(h,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.H2));
  // too late
  assertFalse(mSC_Assertion.isSuccess());
}
```

```
// Test Scenario 3
public void testExecTReventDispatcher() {
  Flight f = new Flight(3, 1);
  Hotel h = new Hotel(3);
  Req req = new Req(f, h);
  mSC_Assertion.request(req,
           new Integer(MSC_Assertion.CUST),
           new Integer(MSC_Assertion.TA));
  // send request for Airline bids
```

```
  mSC_Assertion.reqFlight(f,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.AL1));
  mSC_Assertion.reqFlight(f,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.AL2));
  // send request for Hotel bids
  mSC_Assertion.reqHotel(h,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.H1));
  mSC_Assertion.reqHotel(h,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.H2));

  // receive bid from Airline 2
  mSC_Assertion.bidFlight(f,
           new Integer(MSC_Assertion.AL2),
           new Integer(MSC_Assertion.TA));
  // receive bid from Hotels
  mSC_Assertion.bidHotel(h,
           new Integer(MSC_Assertion.H1),
           new Integer(MSC_Assertion.TA));
  mSC_Assertion.bidHotel(h,
           new Integer(MSC_Assertion.H2),
           new Integer(MSC_Assertion.TA));

  // assert that the Travel Agent task in not
  // in the Complete state
  assertFalse(mSC_Assertion.inState("Complete"));

  // the MSC Assertion will ignore the
  // following event since it is not sent while
  // the Travel Agent task is in its Complete
  // state
  mSC_Assertion.response(f, h,
           new Integer(MSC_Assertion.TA),
           new Integer(MSC_Assertion.CUST));

  // advanced clock by 35 seconds
  mSC_Assertion.incrTime(35);
  assertFalse(mSC_Assertion.isSuccess());
}
```

Scenario 2 represents the case where the Travel Agent missed its deadline when it returned the *response()* back to the Customer. Scenario 3 represents an interesting case where the Travel Agent tried to return a *response()* back to the Customer without getting at least two Airline bids. Note that the MSC Assertion detects the error and ignores the observed *response()* event because it was not sent while the Travel Agent task was in its *Complete* state. The entry action in the *Error* flowchart box in Figure 1 sets the variable *bSuccess* to false, which in turn causes *mSC_Assertion.isSuccess()* to return false and *assertFalse()* to true.

4 Run-Time Execution Monitoring of Message Sequence Chart Assertions

Runtime Execution Monitoring (REM) is a class of methods for tracking the temporal behavior of an underlying application. REM methods range from simple print statement logging methods to run-time tracking of complete formal requirements for verification purposes. NASA used REM to verify the flight code for its Deep Impact project [9]. A recent paper by the authors describes run-time verification of the CARA infusion pump using UML-statechart models combined with statechart assertions for formal requirement specification [8].

Due to the possibility of communication delay and error, as well as faults in the remote service providers, verification alone can never give us the kind of assurance one expects to have for trustworthy SOA-based SoSes. We need to increase the robustness of these systems by *armor-plating* them against unexpected behaviors. In [5], Drusinsky proposed one form of armor-plating that fortifies the software's exception-handling ability via runtime monitoring of temporal assertions, where formal specifications are translated by a code generator into C, C++, or Java statements to be deployed for catching exceptions in the final product during runtime. Here, we can armor-plate the SoSes by embedding the code generated from the MSC Assertions in the Web Services target code to detect run-time assertion failure. In addition, we can add calls to exception-handlers in the *Error* flowchart box of the MSC Assertion to enable run-time recovery whenever the MSC Assertion fails during execution of the Web Services.

5 Discussion and Conclusion

In this paper, we presented a scenario-based, iterative process and UML-MSC like assertions to help ensure the correctness of formal requirements per the modeler's expectations early in the development process. It is easier for system designers to create and understand MSC Assertions than text-based temporal assertions of the kind found in the literature, such as Linear-time Temporal Logic (LTL) [15] because MSC Assertions are similar to the intuitive and familiar UML MSC's.

Various formal verification techniques suggested in the literature [3, 12-14] approach the correctness of Web Services temporal behaviors by expressing these temporal properties as LTL statements and subsequent model checking. Most model checkers, like SPIN, do not support specifications with real-life constraints such as real-time and time-series. Moreover, LTL has a rather weak expressive power (LTL is sub-regular). In contrast, MSC Assertions use Java/C++ as an underlying language and therefore enjoy Turing-equivalent descriptive power

Acknowledgement

The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

References

[1] K. Beck and E. Gamma, "Test infected: Programmers love writing tests", *Java Report*, 3(7) , pp. 37-50, 1998.

[2] K. Birman, R. Hillman and S. Pleisch, "Building network-centric military applications over service oriented architectures", Proc. SPIE Defense and Security Symposium, pp. 255-266, 29-31 Mar. 2005.

[3] F. van Breugel and M. Koshkina, "Models and Verification of BPEL", Dept. of Computer Science and Engineering, York University, Toronto, Canada, September 2006. http://www.cse.yorku.ca/~franck/research/drafts/tutorial.pdf

[4] J. Bhuiyan, S. Nepal and J.Zic, "Checking Conformance Between Business Processes And Web Service Contract In Service Oriented Applications", *Proc. Australian Software Engineering Conference*, 18-21 April 2006.

[5] D. Drusinsky, "Specs Can Handle Exceptions", *Embedded Developers Journal*, pp. 10-14, Nov. 2001.

[6] D. Drusinsky, *Modeling and Verification Using UML Statecharts*, Elsevier Publishing, 2006.

[7] D. Drusinsky and M. Shing, Creation and Evaluation of Formal Specifications for System-of-Systems Development., *Proc 2005 IEEE International Conference on Systems, Man and Cybernetics*, Waikoloa, Hawaii, pp. 1864-1869, Oct 2005.

[8] D. Drusinsky, M. Shing and K. Demir, "Creation and Validation of Embedded Assertions Statecharts", *Proc 17th IEEE International Workshop on Rapid Systems Prototyping*, Chania, Greece, pp. 17-23, June 2006.

[9] D. Drusinsky and G. Watney, "Applying Run-Time Monitoring to the Deep-Impact Fault Protection Engine", *Proc 28th IEEE/NASA Software Engineering Workshop*, pp. 127-133, Dec 2003.

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

[11] X. Fu, T. Bultan, and J. Su, "WSAT: A Tool for Formal Analysis of Web Services", *Proc. 16th Conf. on Computer Aided Verification*, pp. 510-514, 13-17 July 2004.

[12] J. Fisteus, L. Fernández and C. Kloos, "Applying model checking to BPEL4WS business collaborations", *Proc. ACM Symposium on Applied Computing*, pp. 826-830, 13-17 Mar. 2005.

[13] J. Garcia-Fanjul, J. Tuya, and de la Riva, "Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN", *Proc. Int, Workshop on Web Services Modeling and Testing (WS-MaTe 2006)*, 9 June 2006.

[14] H. Huang and R. Mason, "Model checking technologies for Web services", *Proc 2nd Int. Workshop on Collaborative Computing, Integration, and Assurance and 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, 27-28 April 2006.

[15] A. Pnueli, "The Temporal Logic of Programs", *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pp. 46-57, 1977.

[16] W. Tsai, X. Wei, Y. Chen and R. Paul, "A robust testing framework for verifying Web services by completeness and consistency analysis", *Proc. 2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE'05)*, pp. 151-158, 20-21 Oct. 2005.