Automated Link 16 Testing using DEVS and XML

Eddie Mak, Saurabh Mittal, Moon-Ho Hwang
Arizona Center of Integrative Modeling and Simulation (ACIMS)
ECE Department, University of Arizona
Tucson, AZ 85721
{emak, saurabh, mhhwang }@ece.arizona.edu

James J. Nutaro
Oak Ridge National Laboratories
Oak Ridge, TN 38731
nutarojj@ornl.gov

Abstract

With the modernization of Department of Defense (DoD) systems and the growing complexity of communication equipment, traditional test methods and processes have to evolve in order to maintain their effectiveness. DoD acquisition policy requires using modeling and simulation in all phases of system development life-cycles to ensure technical certification and mission effectiveness. The complexity of these systems poses significant challenges over traditional interoperability test methodologies. The Automated Test Case Generator (ATC-Gen) funded by the Joint Interoperability Test Command (JITC) captures Military Standard (MIL-STD) 6016C document and translates it into rules. These are in turn formalized into test cases using Discrete Event System (DEVS) Specification. In this paper, we present a new methodology to generate the test models and perform conformance testing using system theory, the DEVS modeling and simulation (M&S) framework, the System Entity Structure (SES), and the Extensible Markup Language (XML). This new methodology promotes the separation of the models, the simulator, and the distributed simulation. These separations distinguish and promote reusability by developing models, simulator and distributed simulation independently. The DEVS test models are generated from the test cases by the Test Model Generator using the system specifications. These models are written in an XML-SES format; the resulting C++ DEVS source code is generated based on the test model XML file. The Test Driver was designed based on Model/Simulator/View/Control (MSVC) design pattern and developed to execute the DEVS test models. MSVC supports models and simulators separation design. It was also designed to support multiple network simulation protocols and rapid software modifications in order to incorporate new network protocols to the simulation software. This methodology was used to verify the conformance of the Integrated Architecture Behavior Model (IABM) to the MIL-STD 6016C, and the results of the test scenarios were validated using JITC's Simple J network packet monitoring tool. The network packet monitor captured the transmissions and the receipt of the tactical data messages from the Test Driver. The system analyst interpreted and verified the messages, and determined whether these messages were the intended behavior of the Test Driver.

1. Introduction

The extensive use of simulation in military training exercises resulted in significantly improved training progress during the 80s and 90s, which in turn led to the development of widely-used simulation protocols in the defense community, such as Distributed Interactive Simulation (DIS) and High Level Architecture (HLA) [1]. Since then, system development has increasingly made use of M&S and the performance of testing on military systems has become a greater challenge. As detailed in recent DoD reports [2,3], when modeling and simulation is properly used, it provides assistance to formulate system capabilities, compares the cost/benefit ratios of various alternative designs and evaluates their projected effectiveness.

With the modernization of Department of Defense (DoD) systems and the growing complexity of communication equipment, traditional test methods and processes have of necessity evolved to maintain their effectiveness. Testing systems of systems in their operational environment rather than in the lab environment has become necessary, and in an interoperability testing environment, the level of complexity has increased substantially owing to the presence of many different types of military equipment and systems being connected together using diverse middleware and network simulation protocols. Systems interoperate using either common simulation protocols or protocol

translation gateways. With the increased system complexity, testing methodology has to become more rigorous and thorough.

In this paper, we discuss an automated testing framework based on Discrete Event System Specification (DEVS) modeling and simulation formalism, Extensible Markup Language (XML), and System Entity Structure (SES), being introduced at DoD's Joint Interoperability Test Command (JITC) for interoperability testing. This framework supports the separation of experimentation, models, and simulators. The experimental frames are developed to support reusable models and simulators based on the DEVS formalism and dynamic system theory. The hierarchical structures of the models are represented by SES and written in XML format to promote extensibility and interoperability. In order to support the separation of models and simulators in the software development, the Model/Simulator/View/Controller design pattern provides the framework to support model execution and multiple network simulation protocols.

1.1 Motivation

The successful of the military simulation training and developments described above led directly to more intensive use of (M&S) in system development and a revised policy for acquiring new systems known as Simulation-Based Acquisition. The policy required the using of M&S in all phases of system development life-cycles. JITC took the initiative to employ M&S to increase its simulation-based testing capabilities and automation of the testing processes, increasing the effectiveness and responsiveness of the testing [4]. One of the critical areas JITC identified was that Military Standard (MIL-STD) 6016C (also known as TADIL-J standard) had been found to present certain obstacles to traditional test approaches that had not been overcome. Thus, the development of an M&S-based approach automating the Link-16 testing process and applying to standards conformance testing, was initiated. Link-16 (also known as TADIL-J) is the data exchange format of the North Alliance Treaty Organization (NATO) of all the military tactical units.

The automated testing framework introduced in this paper is a part of the Automated Test Case Generator (ATC-Gen) research project funded by JITC to support the mission of standards compliance and certification. With the simulation-based acquisition initiative, the test requirements in the simulated environments become challenges. These challenges include how to automate and define the scope, the extent, and the methodology to update conformance testing. The incorporation of the Systems Theory, the Modeling and Simulation concept, DEVS, SES, XML and the MSVC design pattern all contribute to the development of ATC-Gen increasing the productivity and effectiveness of the TADIL-J conformance testing at JITC.

Many testing methodologies were developed to assist the test engineers to perform conformance testing over the past decades [1]. ATC-Gen is interested in a particular methodology in which the test exercises are carefully planned and the participants are given test scripts and roles to play in a simulated environment. ATC-Gen becomes a participant in a testing environment that can monitor a specific function of the MIL-STD 6016C and exchange tactical data messages with the other players. By interpreting the transmissions and the receipt of the tactical data messages, ATC-Gen is able to determine the degree to which a system conforms to the TADIL-J standard.

The automated testing framework is developed based on three concepts: SES, DEVS, and XML. SES can represent a family of hierarchical DEVS models, and serves as a means of organizing the configuration of a model to be designed, which is extracted from a pruning process. Pruning reduces the number of probable models to meet the system requirement. In the automated testing framework, the minimal testable I/O pair and the test model are represented by Pruned Entity Structures (PES). The test models obtained via PES are in executable form. XML uses elements to break up the test model into hierarchical form, and it can be used to represent the SES hierarchical structure. PES is directly mapped into XML, and the three SES modes become XML elements. XML-PES offers simplicity, extensibility, and interoperability. The test models are represented in XML-PES, which can be transformed into DEVS C++ source code.

1.2 Distributed Simulation

The three components required for the development of simulation software in distributed simulations are model, simulation framework, and middleware. Model is a physical or logical representation of a proposed or real system. A simulation model can be a set of instructions, equations, or constraints for generating I/O behavior. A simulation framework is a defined simulation structure which obeys the instructions of the model and is capable of executing

the model to generate its dynamic behavior. Middleware provides a set of services allowing the data exchange between simulation applications, such as DIS and HLA. Test Driver development is focused on a design which promotes model reuse by developing models independently of the simulation engine. This design distinguishes the separation of model and simulator, and the simulation framework is adapted to the network protocols or middleware.

1.3 Component based design in distributed simulation

Simulation software development has been focusing on component-based design. It is different from the traditional object-oriented (OO) design, which allows the reuse of object classes at the design and implementation level. In contrast, component-based design enables the reuse of the components at the deployment level. Component-based design has increasingly gained popularity because of its reusability and portability. It reduces the cost and the time of software development without compromising software quality. Furthermore, the components can be reused in the same or different applications. When component-based design is incorporated with M&S, it provides separation between models, simulators, and distributed computing. This approach allows the model to remain unchanged and independent of the simulation framework, which is in turn adapted to the selected network simulation protocols. Components separation allows for the addition of middleware to the simulation framework. In some cases, the time management of the middleware is limited by the simulation framework [6]. This approach seems restrictive, but it provides a framework to validate the models and verify the simulators.

1.4 Supporting multiple network protocols

It is common to require a single network simulation software application to support multiple network protocols or middleware in an interoperability testing environment, such as training exercises and systems evaluation. Interoperability testing links multiple platforms or systems together in a heterogeneous environment, such as hardware in the loop systems, live systems, and other simulation software systems. And, these systems often support only selected network simulation protocols. To permit the collaboration in such an environment, the simulation software must be able to support multiple network protocols/middleware simultaneously. For example, a control manager is often used to remotely supervise the operations of all test systems and simulation software in a large-scale testing environment, such as time synchronization and software start/stop using a selected middleware application. The simulation software may transmit its simulation results to other platforms or software using another network protocol. To increase the ability of the simulation software to support multiple network protocols during its lifetime, a new architecture is required to rapidly modify the software in support of new network simulation protocols, taking advantage of the separation between models, simulators, and middleware. If the model and simulator can be distinguished in the simulation software, it will guarantee the model behavior will be unchanged by other simulation environment, and any middleware can be added to the existing simulation framework.

1.5 Extensible Markup Language (XML)

Extensible Markup Language (XML) [7] is an open standard meta-markup language defined by the World Wide Web Consortium (W3C) for text documents. It uses a generic syntax to markup the data. The markup data enhances the structure of the information and identifies the relationships between data. The basic unit of XML markup is called an element, and the elements are used to break up a document into a smaller structure and organize it into a hierarchical form. Tags are used to mark the boundaries of elements. Additional data, such as comments and special instructions, can be specified in the tag. The XML standard allows for the development of XML parsers and accesses any XML document. The parser relies on the tags to break down the documents and processes the XML document. The XML document is well-formed if it satisfies the XML standard.

A valid document is an XML document whose syntax has been validated, most commonly via Schemas or Document Type Definitions (DTDs). A DTD is a collection of declarations describing elements, attributes, parameters, and other markup. It is written to describe precisely which elements can appear in a particular location and what the contents are. Any element type not declared in the DTD but used in the document is illegal. The disadvantages of DTD are the complexity and the lack of restrictions. An XML schema is an alternative to the DTD. It is more powerful and precise than a DTD. The XML schema language is referred to as XML Schemas Definition (XSD). Schemas can create simple and complex data types, restrict the data type, inherit syntax from other schemas, restrict schema inheritance, create attribute groups and support namespaces, and more.

1.6 System Entity Structure (SES)

System Entity Structure (SES) formalism is a knowledge representation scheme for systematically organizing a family of hierarchical DEVS models containing decomposition, taxonomy, and coupling relationships among entities. Decomposition represents how entity can be decomposed into sub-entities. Taxonomy represents how entities can be categorized and sub-classified. Coupling represents how sub-entities are joined together to recompose the entity. The representation of SES is a labeled tree with attached variables that satisfy the following six axioms [8]:

1. Alternating mode: Entity is the root mode. A node and its successor node always have the opposite modes. For example, if a node is entity, its successor is either aspect or specialization.
2. Strict hierarchy: A label only appears once in any path of the tree.
3. Uniformity: If the nodes have the same names, they will have identical variables and isomorphic sub-trees.
4. Valid brothers: No two brothers have the same label.
5. Attached variables: variable names must be unique in each node.
6. Inheritance: every entity in a specialization inherits all the variables, aspects, and specializations from the parent of the specialization.

There are three types of nodes: entity, aspect, and specialization. Entity is a real world object, and it may have attached variables. It can be identified as either a composite entity or an atomic entity. Atomic entities cannot be broken down into sub-entities, while composite entities can. An aspect represents a decomposition of the entity. The children of an aspect are entities representing the decomposed components. A specialization defines the taxonomy of the entity, and it is used to classify the general entity into specialized entities. The children of a specialization are entities representing the variation of its parents.

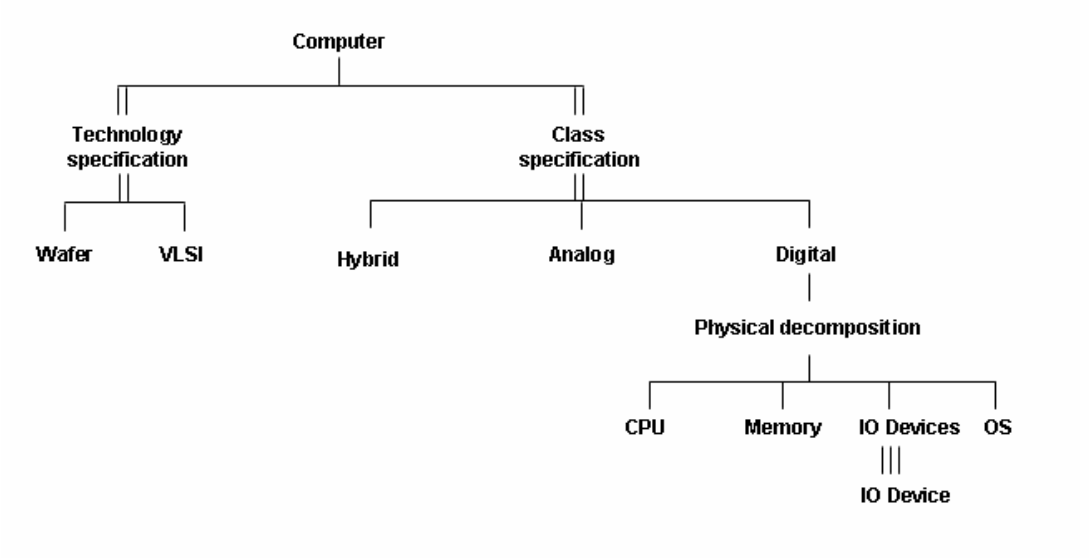


Figure 1: System Entity Structure for computer

Figure 1 illustrates an SES example for computer. The root entity is *computer* and it has two specifications. The *technology specification* has two entities, *wafer* and *VLSI*. The *class specification* has *hybrid*, *analog*, and *digital* entities. *Digital* can decompose into *CPU*, *memory*, *IO devices*, and *operating system* entities. The triple vertical lines connecting *IO Devices* and *IO Device* represent the multiple decompositions. It represents a digital computer may have multiple IO devices.

1.7 Plan of Paper

The next section of the paper provides an overview of Automated Test Case Generator. Section 3 provides the backgrounds of the System Theory and the framework of Modeling and Simulation. In Section 4, the automated testing methodology is introduced. Test Model Generator transforms minimal Input/Output pairs to DEVS test models. Test Driver is developed using MSVC pattern to enhance the reusability of the simulation software and allows for the test models, the simulators, and the middleware developed separately. Section 5 shows the auto

correlation experiment and its result. The scenario was tested by the Joint SIAP System Engineering Organization (JSSEO) using the Integrated Architecture Behavior Model (IABM) and verified by the ATC-Gen Test Driver. Section 6 provides a discussion of various other layered paradigms and its comparison with our MSVC architecture. Finally, Section 7 concludes the paper with the discussion of future work.

2. Proposed Solution overview: Automated Test Case Generator (ATC-Gen) Concept

ATC-Gen development is composed of four stages that are developed in conjunction with DEVS formalism as illustrated in Figure 2. It applies DEVS to the formalization of Military Standard (MIL-STD) 6016C. The MIL-STD is written in natural language, and can be formalized into the system theory framework by putting a set of requirements in the natural language. By combining system theory and DEVS, the formalization can be transformed into an executable simulation model, and the model can be implemented for testing. By using software tools and modeling packages, the test model can be derived and generated from the natural language. Then, these test models are transformed into an executable format and deployed by the Test Driver to perform testing on the SUT. The processes described above become automated testing.

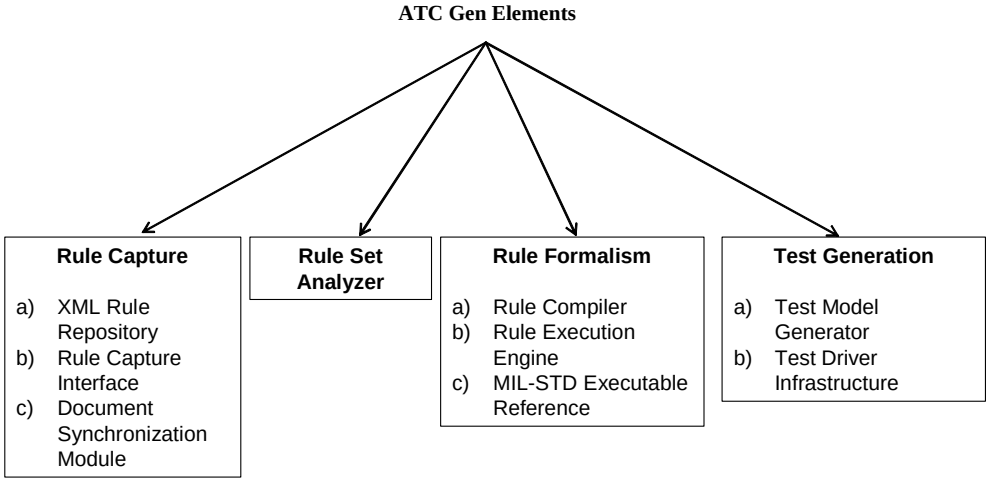


Figure 2: ATC-Gen Development

The first stage is Rule Capturing, which captures and formalizes the MIL-STD 6016C in XML format. The military standard is written in the form of natural language, but do not support the systematic study of large-scale intelligent system. By translating the MIL-STD to a constrained form of natural language that is used in describing system behavior, analysis will be easier. Natural language statements, such as "IF, THEN" used in knowledge-based expert system and artificial intelligence will be suitable to describe the system behavior. The disadvantage of the natural language statement is that it is incapable of describing the time behavior of the system. It can be overcome by using a finite state machine which will be described in stage 3. Capturing requires analysts to read and interpret the standard. Formalizing requires the analysts to identity ambiguous requirements and extracts the state variables and rules. The rules are written in the "If, Then" format, and these rules are not associated with time.

To illustrate this, let us consider a simple system consisting of a vending machine and a customer. The vending machine is in idle state if there is no or not enough money inserted. In addition, the vending machine does not dispense any item if the customer does not put correct amount of money. It dispenses an item if the correct amount of money is inserted into the machine. This simple system can be described by three statements without any time reference:

1. If the vending machine is idle, there is either no or not enough money inserted.
2. If the customer doesn't insert the correct amount of money, no item will be dispensed.
3. If the customer inserts enough money, an item will be dispensed from the machine.

There are three state variables in the above example: idea, money, and item. Idle represents there is either no or not enough money inserted by the customer. Money represents the amount of money required to purchase the item, and item represents the product that the customer wishes to get from the machine. The “IF, THEN” statements can be written into the XML format. Tags are created to enhance the structure and identify the relationship in the document, and they are the legal building blocks of the XML document. Each statement in Figure 3 is considered as a rule. Each rule is composed of conditions and actions. Conditions and actions can have state variables. The combination of all rules in an example is a rule set. Based on these guidelines, the vending machine example is translated to the XML document. A XML Document Type Definition (DTD) or schema must be created to validate and provide the correct syntax to the XML document.

```
<RuleSet>
        <name>Vending machine example</name>
        <rule name="1">
                <condition txt="If vending machine is idle">
                        <var name="money" varType="currency"/>
                </condition>
                <action txt="no action"/>
        </rule>
        <rule name="2">
                <condition txt="If customer inserts insufficient money">
                        <var name="money" varType="currency"/>
                </condition>
                <action txt="idle and no item is dispensed">
                        <var name="item" varType="String"/>
                </action>
        </rule>
        <rule name="3">
                <condition txt="If customer inserts enough money">
                        <var name="money" varType="currency"/>
                </condition>
                <action txt="dispense item that is chosen by the customer">
                        <var name="item" varType="String"/>
                </action>
        </rule>
</RuleSet>
```

Figure 3: XML RuleSet

Stage 2 consists of the Rule Set Analyzer. It employs the Dependency Analyzer (DA) to determine useful relationships among rules. The DA uses DEVS formalism to validate the XML models and provides a visual display of dependencies, allowing selection of test sequences by the test engineer. We could have used other XML analysis and validation tools but in order to develop a consistent methodology we preferred to use DEVS as the common denominator. The DA uses DTDs specially written for the project to validate the syntax of the XML files. As mentioned briefly above, the DTD ensures the correctness of the XML files before further processing. Once the syntax is validated, all the rule sets in the XML files will be stored in memory. The DA will determine, manipulate and reorganize all the rules and variables, allowing potential dependencies to surface if shared state variables are identified between pairs of rules. Finally, all the rules and variables will be stored in a single new XML file, which will be used when creating test sequences in the next stage.

Stage 3 is Rule Formalization, which consists of selecting and formulating the test sequences; test models are generated from these sequences. The test engineer formulates test sequences in accordance with the structure of the testing requirements, and converts them into executable simulation models. The DA is executed in order to restore the XML files and the rules created at the end of stage 2, producing a file containing all the possible paths through the simulation and the information required to build a visual representation of the rule connections. By invoking the GUI, it displays the rules by level and shows the sequence of rule firing, providing a visual organization of the rules and their interrelationships and allowing the test engineer to examine the paths that are created between rules in

order to finds any potential errors. Although the DA shows all the possible paths, an identification of all possible paths is impractical owing to the fact that not all paths are useful. The test engineer manually examines all feasible paths and creates a test case according to the specification and requirement. The test case is the description of the desired SUT behavior in the minimal testable input/output representation. Based on the minimal table I/O pairs, the test model generates the DEVS test model in C++.

Stage 4 is Test Generation, which consists of generating DEVS C++ test models and executing the test models against a real hardware/software system using the Test Driver. The Test Model Generator generates C++ DEVS model in two steps. First, it converts the test cases to XML test models. Second, the XML test models are converted into C++ DEVS model. The Test Driver is an experimental frame which is capable of executing the test model behavior and interacts with and connects to the System Under Test (SUT) via a High-Level Architecture (HLA) or Simple J interface. The Test Driver performs SUT conformance testing by inducing the testable behavior expressed in the models into the SUT and checking the responses for accuracy.

3. Background: System Theory and Framework of Modeling and Simulation

In this section, we review some of the backgrounds required for the discussion of the automated Link-16 testing approach described in this paper.

3.1 DEVS Specification

The DEVS formalism was introduced by Bernard Zeigler [5] to provide a mean of modeling discrete event systems in a hierarchical and modular way. DEVS exhibits the concepts of system theory and modeling, and supports capturing the system model in structural and behavioral perspectives. A DEVS model can be either an atomic or coupled model. In the DEVS formalism, a large system can be modeled by both atomic and coupled models. The atomic model is the basic model that describes the behavior of a component. A Discrete Event System specification (DEVS) atomic model is defined by the structure in Figure 4.

$$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta \rangle$$

where

X is the set of input values
S is the set of state
Y is the set of output values
δ_{int}: $S \rightarrow S$ is the internal transition function
δ_{ext}: $Q \times X \rightarrow S$ is the external transition function, where
$Q = \{(s,e) | s \in S, 0 <= e <= ta(s)\}$ is the total state set,
and e is the time elapsed since last transition
λ: $S \rightarrow Y$ is the output function
ta: $S \rightarrow R_{0^+,inf}$ is the time advance function

Figure 4: Classic DEVS Specification

Atomic and coupled models can be simulated using sequential computation or various forms of parallelism. The basic parallel DEVS formalism extends the classic DEVS by allowing bags of inputs to the external transition function, and it introduces the confluent transition function to control the collision behavior when receiving external events at the time of the internal transition. The parallel DEVS atomic model is defined by the structure in Figure 5.

$M = \langle X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta \rangle$

where

X is the set of input values
S is the set of state
Y is the set of output values
δ_{int}: S $\rightarrow$ S is the internal transition function
δ_{ext}: Q x $X^b \rightarrow$ S is the external transition function,
where X^b is a set of bags over elements in X, Q is the total state set.
δ_{con}: S x $X^b \rightarrow$ S is the confluent transition function,
subject to $\delta_{con}(s, \Phi) = \delta_{int}(s)$
λ: S $\rightarrow Y^b$ is the output function
ta: S $\rightarrow$ $R_{0^+,inf}$ is the time advance function

Figure 5: Parallel DEVS Specification

A DEVS-coupled model designates how atomic models can be coupled together and how they interact with each other to form a complex model. The coupled model can be employed as a component in a larger coupled model and can construct complex models in a hierarchical way. The specification provides component and coupling information. The coupled DEVS model is defined as the structure in Figure 6.

$M = \langle X, Y, D, \{M_{ij}\}, \{I_j\}, \{Z_{ij}\} \rangle$

Where

X is a set of inputs
Y is a set of outputs
D is a set of DEVS component names
For each i $\in$ D,
$\mathbf{M_i}$ is a DEVS component model
$\mathbf{I_i}$ is the set of influences for I
For each j $\in$ I_i,
$\mathbf{Z_{ij}}$ is the i-to-j output translation function.

Figure 6: Coupled DEVS Specification

Three different DEVS formalisms have been introduced. The classic DEVS formalism treats components sequentially, and the parallel DEVS formalism treats components concurrently. These formalisms also include the means to build coupled model from atomic models.

3.2 The Hierarchy of System Specifications

System theory deals with a hierarchy of system specification which defines levels at which a system may be known or specified. Table 1 shows this Hierarchy of System Specifications (in simplified form, see [9]).

- At level 0, we deal with the input and output interface of a system over a time base.
- At level 1, we observe the behavior of the system by gathering a collection of all I/O pairs.
- At level 2, we add the initial states to the specification. When the initial states are known, there is a functional relationship between the inputs and outputs.
- At level 3, the system is described by the state space and the state transition functions. The transition function describes the state changes as the system responds to inputs and generates outputs.
- At level 4, we specify how the system is composed of interacting components in a coupling structure. Each component is a system on its own with state set and state transition functions. One property of a coupled system, called “closure under coupling,” guarantees that a coupled system at level 3 itself specifies a

system. This property allows hierarchical construction of systems, i.e., that coupled systems can be used as components in larger coupled systems.

Level	Name	What we specify at this level
4	Coupled Systems	System built up by several component systems which are coupled together
3	I/O System	System with state and state transitions to generate the behavior
2	I/O Function	Collection of input/output pairs constituting the allowed behavior partitioned according to the initial state the system is in when the input is applied
1	I/O Behavior	Collection of input/output pairs constituting the allowed behavior of the system from an external Black Box view
0	I/O Frame	Input and output variables and ports together with allowed values

Table 1: Hierarchy of System Specification

As we shall see in a moment, the system specification hierarchy provides a mathematical underpinning to define a framework for modeling and simulation. Each of the entities (e.g., real world, model, simulation, and experimental frame) will be described as a system known or specified at some level of specification. The essence of modeling and simulation lies in establishing relations between pairs of system descriptions. These relations pertain to the validity of a system description at one level of specification relative to another system description at a different (higher, lower, or equal) level of specification.

3.3 Framework for Modeling and Simulation

The modeling and simulation framework defines entities and their relationships that are central to the M&S enterprise [5]. The basic entities of the framework are source system, model, simulator, and experimental frame as illustrated in Figure 7, and they are linked to the modeling and simulation relationships.

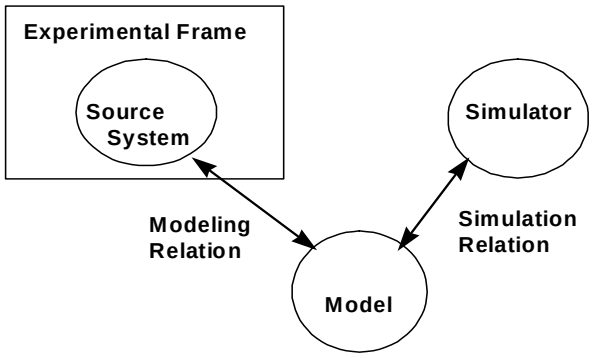


Figure 7: Basic M&S Entities and their relationships

The source system is the real or virtual environment that we are interested in modeling. It is viewed as a source of observable data in the form of time-indexed trajectories of variables. The data that has been gathered from observing or otherwise experimenting with a system is called the system behavior database. The data is viewed or acquired through experimental frames of interest to the modeler.

The experimental frame is a specification of the conditions under which the system is observed or experimented. It reflects the objectives of the experiment performed on a real system or simulation. An experimental frame specification consists of four major subsections: input stimuli, control, metrics, and analysis [5].

Multiple experimental frames can be used for a single system, and the single experimental frame can be applied to many systems. Conversely, there are multiple objectives to test a system, and a single objective is applied to many systems. There are two valid views of an experimental frame. A frame can be viewed as data element type that is entered into a database. Also, the frame can be viewed as an observer, which has a generator, an acceptor, and a transducer, as illustrated in Figure 8. It can interact with the System Under Test (SUT) to obtain experimental data under specified conditions. The generator describes the inputs applied to the system or model. The transducer observes and analyzes the system output. The acceptor monitors the experiment to see the experimental condition, and compares the generator inputs with the transducer outputs. Figure 8 shows the component structure diagram of the experimental frame.

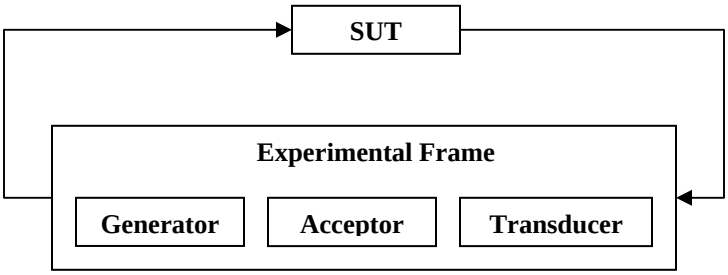


Figure 8: Experimental Frame

3.4 Model Continuity

Model continuity refers to the ability to transition as much as possible of a model specification through the stages of a development process. The component models of a distributed system can be tested incrementally and deployed to a distributed environment for execution. It supports a design and test process in four steps [5]:

1. Conventional simulation to analyze the system under test within a model of the environment linked by abstract sensor/actuator interfaces.
2. Real-time simulation, in which simulators are replaced by real-time execution engines while leaving the models unchanged.
3. Hardware-in-the-Loop (HIL) simulation in which the environment model is simulated by a DEVS real-time simulator on one computer while the model under test is executed by a DEVS real-time execution engine on the real hardware.
4. Real execution, in which DEVS models interact with the real environment through the earlier established sensor/actuator interfaces that have been appropriately instantiated under DEVS real-time execution.

Model continuity reduces the occurrence of design discrepancies throughout the development process, thus increasing the confidence that the final system realizes the specification as desired. Furthermore, it makes the design process easier to manage since continuity between models of different stages is retained.

4. Automated Testing Methodology

The automated testing approach combines the systems theory, Modeling and Simulation framework, and model-continuity concepts, and applies the Bifurcated Model-Continuity-based Life-cycle Process [4] to the Link 16 conformance testing. In this section, the two processes of the ATC-Gen stage 4 are discussed – Test Model Generator and Test Driver.

4.1 Test Model Generator

The objective of the Test Model Generator is to a create DEVS test models based on minimal testable I/O pairs. In this paper, we are performing a state reachable graph analysis and not generating a complete system behavior. The test scenario is defined in the form of inputs and outputs according to the MIL-STD 6016C definition. The collection of I/O function is infinite in principle because there are numerous states to start from and the inputs can be extended indefinitely. For practice purposes, we restrict our testing focus to messages, and assuming they are the only automatable observables available for testing. These tests are performed against the military hardware/software systems to study its conformity to the MIL-STD.

The DEVS test models are in the form of an experimental frame and allow the Test Driver to perform experiments against the System Under Test. The test engineer analyzes the requirements and creates the test scenarios which describe the behaviors of the SUT based on the MIL-STD 6016C. The requirements are written in minimal testable input/output representation, and the test models are created by applying the model mirroring concept that reverse the minimal testable I/O pairs. Both the minimal testable file and test models are written in XML format and represented by SES, allowing for the transformation between the two XML files. The inputs/output pairs are now represented by three atomic models: *holdSend*, *waitReceive*, and *waitNotReceive*. Since the input/output are in sequential order, only one atomic model is active each time, and the rest of the atomic models are passive. In order to try out these test models against the real system, they are converted to software programming source code. This allows quick incorporation of the test models into the Test Driver. Figure 9 illustrates the sequence flow of processes used in automated test model generation described above.

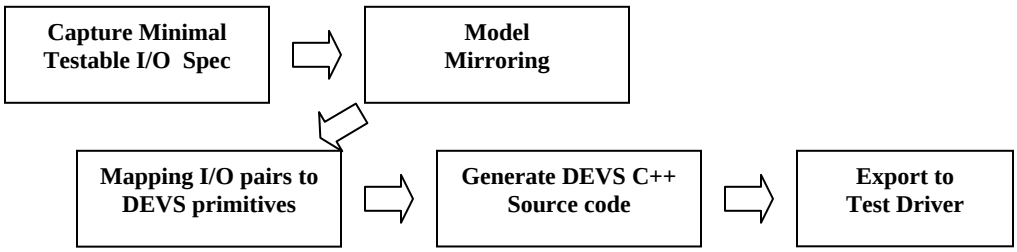


Figure 9: Test Model Generator

4.1.1 Minimal Testable Input/Output Representation

If X and Y are the inputs and Z is the output, figure 10 illustrates the complexity involved in the Level 1 of the system specifications. There are many possibilities to characterize the I/O behavior at the Level 1 in a basic operation [4]. Some additional information is required to determine how the system responses to the inputs and the order of the inputs and output pairing. In Figure 10, *X(x)* denotes that *X* is a function receiving/sending *x* as input. Similarly *Z(x+y)* denotes that *Z* is a function that takes both *x* and *y* as inputs simultaneously.

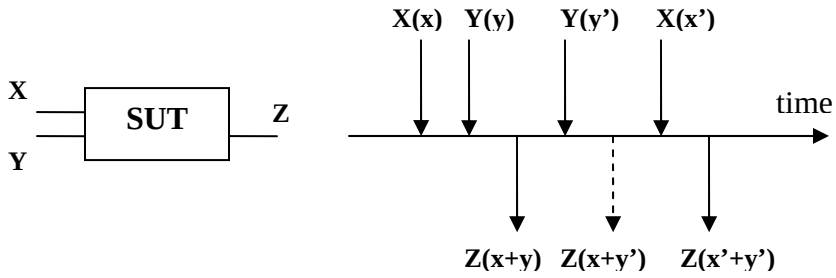


Figure 10: Variants of Input/Output Pairs

Zeigler [4] introduced the concept of minimal testable pair to handle the complexity of the I/O segments described above. In each minimal testable I/O pair, there are a finite number of input messages and at most one output message. Each minimal testable pair also refers as a segment with limited complexity. In such, the test cases are easier to synthesize from the segments. Figure 11 illustrates the separation of I/O pairs into the minimal testable input/output pairs. Since each input stimulus produces a unique output with a given initial state, the I/O pair can be extracted from the level 2 of the system specification, and it is said to be a minimal testable I/O representation. Figure 11 illustrates how an input set is mapped to a corresponding output set. We've shown that by extracting the mappings from Figure 10 above.

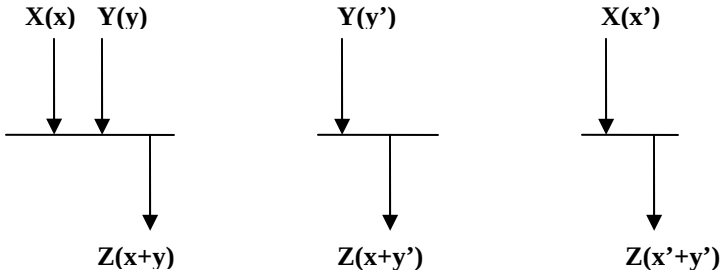
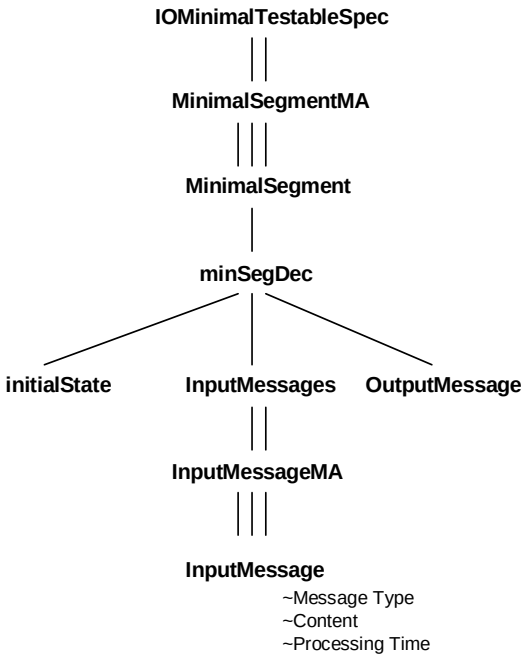


Figure 11: Minimal Testable Input/Output Pairs

4.1.2 Capturing IO Minimal Testable XML file

The IO Minimal Testable file is generated by the test engineer, and it describes the input and output behavior of the System Under Test. The behavior is described based on the minimal testable input/output representation, in which each pair has a set of inputs and at most one unique output, and in which a scenario is composed of multiple pairs in sequential order. Each scenario can be represented by SES and written in XML format. Representing SES in XML format is very straightforward. Each SES node has a mode which can be entity, aspect, or specialization, and each mode of a node represents an XML element. Figure 12 shows the transition from SES to XML. SES is using nodes to represent the structure of a test scenario, and XML is using elements to break up the structure and represent the SES nodes.




```
<IOMinimalTestableSpec>
  <aspectsOfIOMinimalTestableSpec>
    <minimalSegmentMA>
      <minimalSegment>
        <aspectsOfminimalSegment>
          <minSegDec>

            <initialState>
            </initialState>

            <inputMessages>
              <aspectsOfinputMessages>
                <inputMessagesMA>
                  <inputMessage>
                  </inputMessage>
                </inputMessagesMA>
              </aspectsOfinputMessages>
            </inputMessages>

            <outputMessage>
            </outputMessage>

          </minSegDec>
        </aspectsOfminimalSegment>
      </minimalSegment>
    </minimalSegmentMA>
  </aspectsOfIOMinimalTestableSpec>
</IOMinimalTestableSpec>
```

Figure 12: IO Minimal Testable SES Diagram & XML Tags

4.1.3 Mirror Image of I/O Pairs

The minimal testable I/O pairs describe incoming and outgoing messages in the SUT. The Test Model I/O pairs can be generated by reversing the roles of input and output in the SUT as shown in Figure 13. For example, a test

scenario requirement generated by the test engineer is given which specified that the SUT will receive message a, b, and c. After receiving these messages, message d will be sent. The test model is generated by reversing the SUT procedures, which sends message a, b, and c, and the test model will wait to receive message d. The mirror image concept allows creating a test model for any minimal testable pair in the SUT's Input/Output Specification. The image still exhibits the characteristic of the minimal testable I/O pair: a set of inputs produces a unique output.

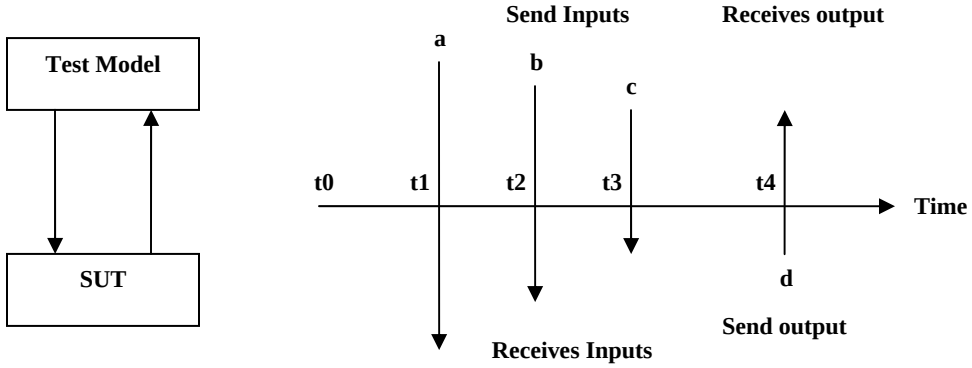


Figure 13: Mirror Image of Input/Output Pairs

4.1.4 Primitives of Basic Test Model

Based on the minimal testable I/O representation, there are two basic operations in the Test Model: send message and receive message. These two operations can be represented by three atomic models: *holdSend*, *waitReceive*, and *waitNotReceive*. *holdSend* sends a message after the resting time expires. The receive message operation is represented by two atomic models: *waitReceive* waits for a incoming message at a pre-defined time interval, and determines the pass-fail condition by comparing the pre-defined value to the value of the received message, while *waitNotReceive* doesn't wait for any message and determines the condition but idles the model for a pre-defined time interval. In some instances, the minimal testable pair of the SUT does not transmit any message at the end. In order to represent this behavior in minimal testable I/O representation, *waitNotReceive* will be used to represent a unique output.

Figure 14 illustrates how to construct the basic test model using these primitives. The behavior of the SUT is described by the minimal testable I/O pair, and the SUT sequences are receive, receive, and transmit. The test model is constructed by the mirror image concept described in Section 3.3.2, and the sequences of the SUT image are transmit, transmit, and receive. The transmission is represented by the *holdSend* model, and the reception is represented by *waitReceive* model. The test model sends a J3.2 message with content data1 at time t1 and is followed by a second J3.2 with content data2 at time t2. After transmitting two messages, the test model will wait for J7.2 message between time t2 and time t3. When the SUT receives the two J3.2 messages, it will transmit a J7.2 with content data3 at time t3. If the test model receives the message within the time interval, *waitReceive* will process the J7.2 message.

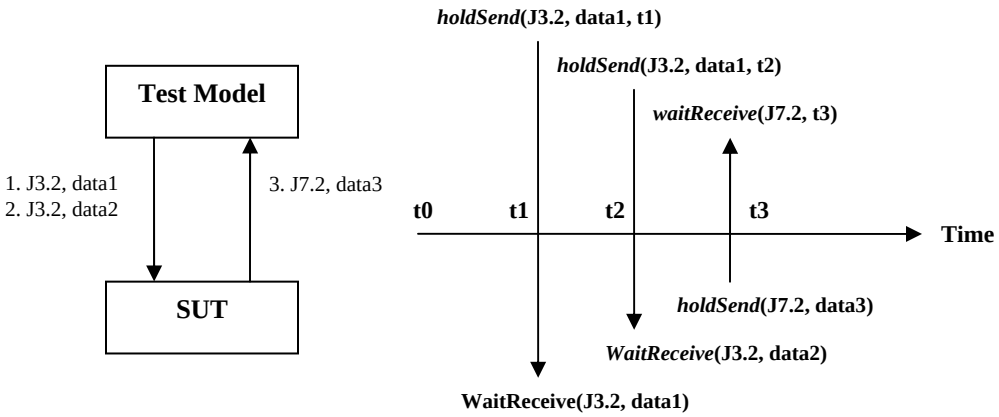


Figure 14: Using the primitives to construct test model

The test model is in the form of an experimental frame in which the *holdSend* atomic model is a generator and the *waitReceive* atomic model is the acceptor. The outputs of the atomic model are connected to the inputs of the next atomic model to form the basic test coupled model. Each atomic model and coupled model has two input ports: start and in_Jmsg, and two output ports: out_Jmsg and pass. The minimal testable I/O pair is represented by an atomic models , and the basic test coupled model is formed by coupling the atomic models together in sequential order as shown in Figure 15.

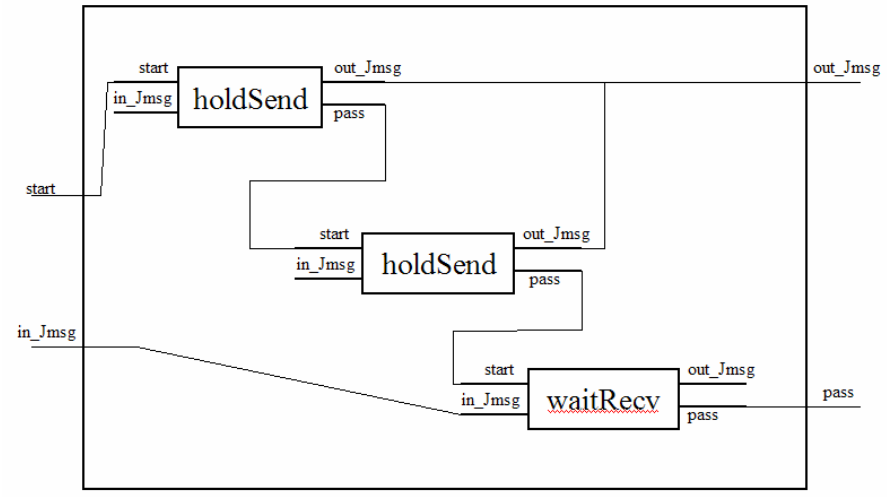


Figure 15: Basic test coupled model showing DEVS component atomic models

4.1.5 Composite Test Model

When the test models derived from the minimal testable I/O pairs are cascaded together, it is called a composite test model. A SUT scenario is usually defined by one or more minimal testable pairs, which are represented in sequential order after which the test models are cascaded together. The atomic models and the coupled models are triggered by a start event through the start port. When the condition of the basic test model is met, it will trigger another basic test model.

Figure 16 illustrates the composite test model derived from cascading the basic test models, and the basic test model is formed by cascading the atomic models. The initial correlation test model has three operations: transmit twice and receive, while the confirmation test model has two operations: transmit and receive, and each operation is represented by an atomic model. In this composite model, the initial correlation model sends the J3.2 messages and

then waits for the given response. If the response is correct, it starts up the next model; otherwise, it stops and reports the failure.

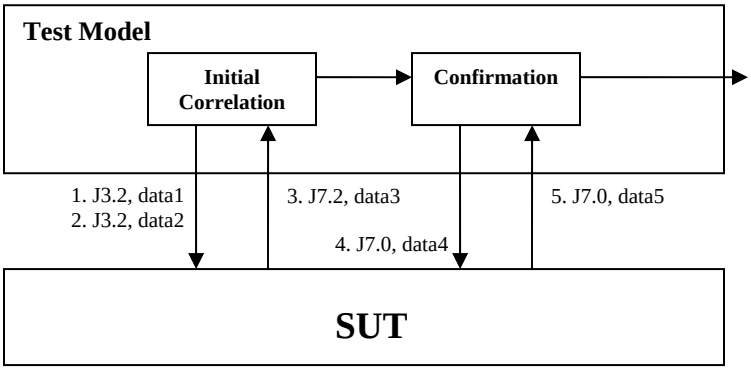


Figure 16: Composite Test model

Based on the composite model, we can cascade a sequence of composite test models together to form a more complex test scenario. Each composite test model represents a unique scenario which it waits for the correct response in order to start up the next model.

4.1.6 Automating the Mapping from Minimal Testable Pairs to Test Models

It is important to automate the transformation from the minimal testable pairs to the test models, because the test models can be executed efficiently in a distributed simulation environment using the Test Driver. Also, it provides traceability between the SUT behavior and the DEVS test models. The minimal testable file and test model files are described in SES and written in XML format. The SES for the test model is shown in Figure 17, and each node of the test model SES diagram is an XML element. The mapping is performed by a software program which generates an XML file called TestGen.xml. The format of the TestGen XML file is validated by a pre-defined DTD.

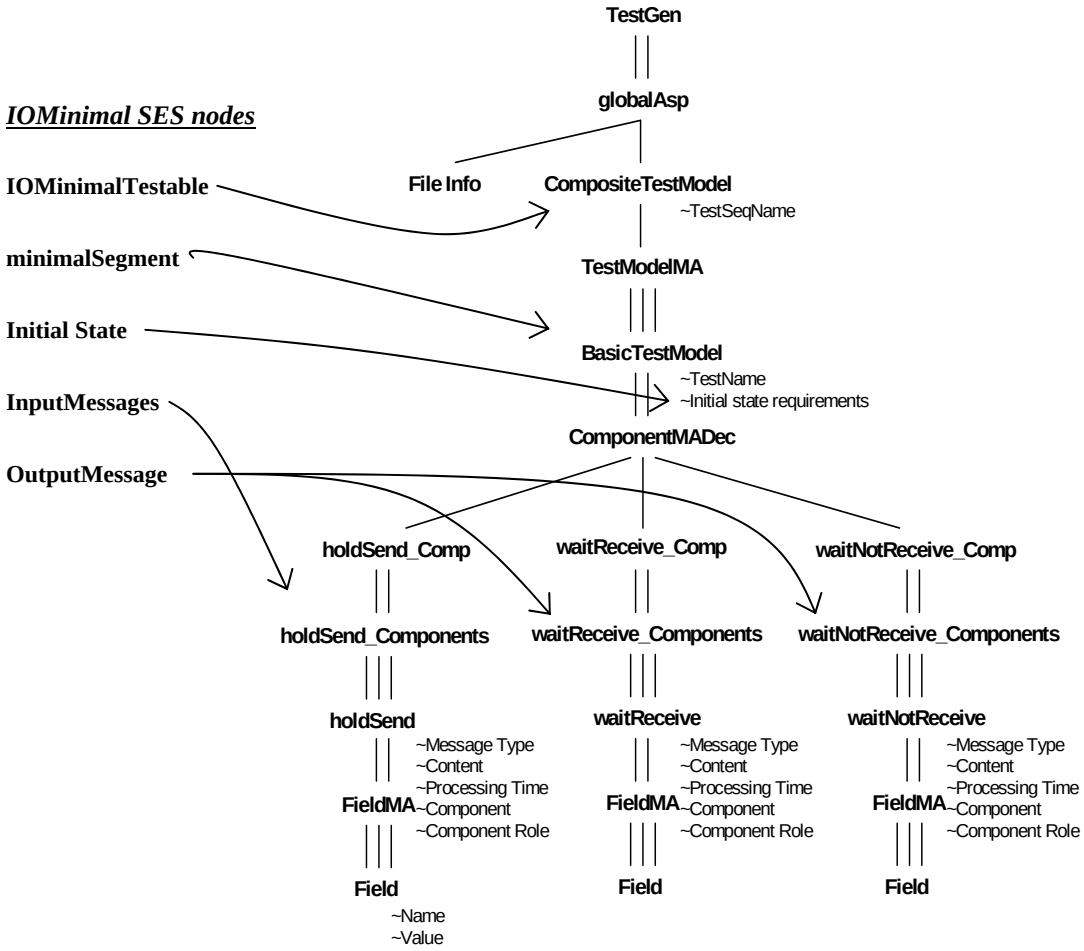


Figure 17: Mapping Minimal Testable pairs into Test Model SES

Figure 17 also illustrates the mapping from minimal testable to test model. For example, the initial state node is mapped to the initial state requirement variable of the BasicTestModel node. The InputMessages node is mapped to the *holdSend*_Components node. The OutputMessage node is mapped to either *waitReceive*_Components or *waitNotReceive*_Components.

4.1.7 SUT Model

SUT Model is a DEVS model that is created to mimic the behavior of the true SUT and to verify the correctness of the test models. The SUT model is implemented into the Test Driver and verify against the test model version via HLA middleware or SimpleJ protocol. This DEVS inversion concept was first introduced by Song and Kim [10].

The SUT model is generated based on the test model generation concept. Instead of going through the model mirroring process, the minimal testable I/O pairs directly transform into the SUT DEVS models. The minimal testable representation has two operations: send message and receive message. The *holdSend* atomic model is used to send message, and the *waitReceive* atomic model is used to receive message. The transformation process follows the opposite of the minimal testable pair concept in which an I/O pair has either a set or an empty set of outputs followed by an input. Figure 18 illustrates mapping three minimal testable pairs into three DEVS atomic models. The first and second minimal testable pairs have no outputs and only one input, and each input is mapped into the *waitReceive* atomic model. The third minimal pair has only two outputs, and the outputs are mapped into two *holdSend* atomic models.

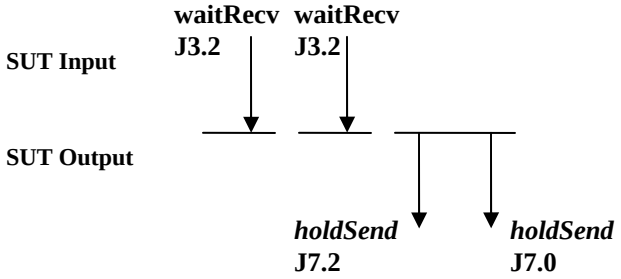


Figure 18: DEVS atomic models extracted from SUT model for Auto Correlation Example

During the verification process, the SUT and Test Model are executed by two different Test Drivers. Each DEVS model is associated with a processing time. If the models are transformed directly from the minimal testable pairs, both SUT and Test models will have the same processing times. Even if both SUT and Test Model Drivers are perfectly synchronized in time, there ought to be some delays from either the computer network or the computer itself. One of the Test Drivers will miss an incoming message because the resting time is expired in the atomic model, and the whole test scenario will fail. In order to avoid this problem, two assumptions are made. First, the SUT Test Driver always starts n/2 seconds before the Test Model Driver. Second, a time shift of n seconds is added to the SUT model's processing time. The rules of the time shift are defined as follows,.

1. If the current model is *holdSend* and the last model is not *holdSend*, the time shift is -n second.
2. If the current model is *waitReceive* and the last model is *holdSend*, the time shift is +n second.

This introduction of different initialization times of the SUT Test Driver and Test Model Driver is deliberately made so that the inputs and outputs are always out-of-sync and are detected accordingly. In our practical experience, when both the SUT Test Driver and Test Model Driver are in-sync 100% they do not acknowledge the messages flowing across each other.

Figure 19 illustrates the execution time of the SUT and test models of the auto correlation scenario by applying the time shift concept into the SUT. Since the SUT always starts n/2 seconds before the Test Model, we can assume that the SUT is always running a few hundred milliseconds faster than the Test Model. The whole objective is to make the SUT Test Driver and Test Model Driver out-of-sync. It is immaterial who is running faster.

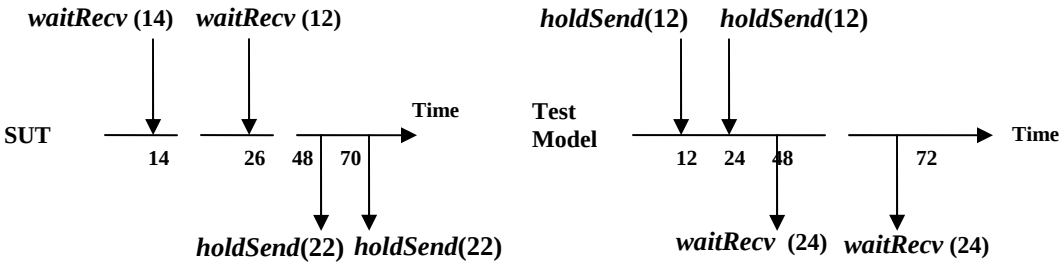


Figure 19: SUT and Test Model Timeline for Auto Correlation

4.1.8 DEVS Code Generation

The platform-dependent programming source codes are generated based on the information provided in the test model XML file. As shown in Figure 17 above, the CompositeTestModel is the test sequence, and the BasicTestModel is equivalent to a minimal testable pair. If multiple I/O pairs exist, multiple BasicTestModel scopes are cascaded together. Each atomic model has two input ports and two output ports, and the port usages are defined as follows:

- *holdSend* has the 'start', 'out_Jmsg', and 'pass' ports.

- *waitReceive* has the 'start', 'in_Jmsg', and 'pass' ports.
- *waitNotReceive* has the 'start' and 'pass' ports.

There are two levels of coupled models: CompositeTestModel and BasicTestModel. All the coupled models have the same input and output ports as the atomic model, and all the ports are used. BasicTestModel coupled models contain the atomic models as shown in Figure 15, and CompositeTestModel contains the BasicTestModel coupled models as illustrated in Figure 16. Since the hierarchical structure and the port coupling relationships of the atomic and coupled models are defined, the source codes can be generated easily.

A Java program was created to parse the test model XML file and traverse the SES structure to create the DEVS test model source codes. Five methods were created to generate the source codes:

1. CRSReader: the CRS file contains the trajectory of an aircraft. Each trajectory point is associated with a time index. This method provides the trajectory information required by the *holdSend* atomic model.
2. writeCompositeClassFile: It creates and prepares the hierSeqDigraph file and calls the writeCompositeTestModel method.
3. writeCompositeTestModel: This method creates the coupling information of all the basic test models, and initiates the writeBasicClassFile method.
4. writeBasicClassFile: This method creates the testSeqDigraph file and the coupling information of the atomic models, and initiates the writeBasicTestModel method.
5. writeBasicTestModel: Creates the atomic models information. During the creation of the *holdSend* atomic model, the CRSReader will be called to contain the trajectory required the model.

Figure 20 below illustrates the associations of these five methods and the test model SES diagram.

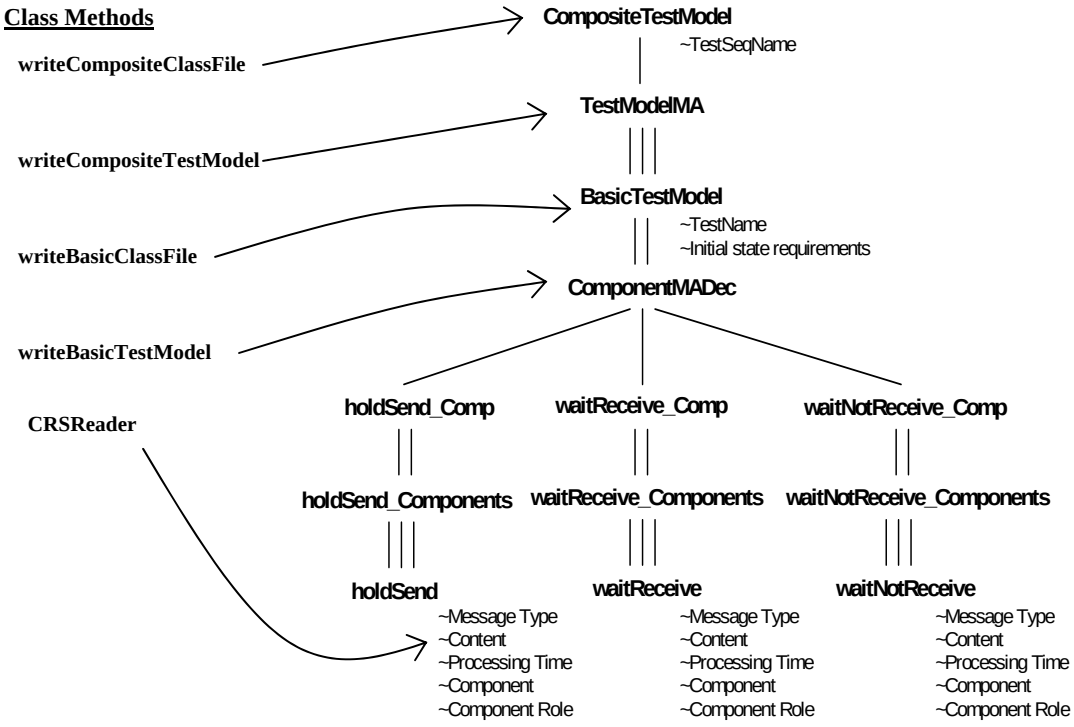


Figure 20: Associations between Java methods and SES nodes

4.2. Test Driver

The ATC-Gen Test Driver (TD) is an experimental frame designed to perform interoperability testing on TADIL-J systems. The objective of the Test Driver is to execute the DEVS test models generated by the Test Model Generator (TMG). TD emulates a tactical TADIL-J system by providing simulated TADIL-J messages over the

simulated tactical communication network, and accepts TADIL-J messages from the SUT to determine the condition of the test model, and is implemented via component-based design using the enhanced Model/Simulator/View/Controller (MSVC) design pattern. The DEVS model is generated by TMG. The TD simulator is a thread derived from the controller that schedules and receives Link-16 messages. The viewer extracts outputs from the simulator, and converts the outputs into a specific middleware format.

4.2.1 Enhanced MSVC Design Pattern

Jim Nutaro [11] demonstrated that the simulator was tuned to the behavior of certain network simulation protocols, and the controller could be rapidly modified to support other protocols. For example, the simulator is associated with HLA time management through the controller in order to pace the execution. The same simulator can be reused by implementing a new controller supporting other network simulation protocols to pace the execution using the wall clock. In this methodology, one controller is associated with one simulator due to the difficulties inherent in handling multiple control strategies and the differing characteristics of the middleware. The simulator is a child thread derived from the controller thread that contains the parameters to influence the simulator. Although Nutaro did not consider using the controller to manage the model operation, his work led to the enhanced MSVC framework, where a new controller is implemented to control the model as well as the simulator.

Figure 21 below provides a graphical representation of the enhanced MSVC paradigm [12]. The functions of the model, simulator, and view are the same as the original MSVC design. A basic controller is implemented to receive the messages via middleware. The specialized controllers are derived from the basic controller to handle message routing to either the simulator or the model. For example, as shown in Figure 21, Simple J controller handles the inputs from Simple protocol and controls the DEVS simulator. HLA Controller receives inputs from HLA middleware and controls the model's operations, such as start and stop operations.

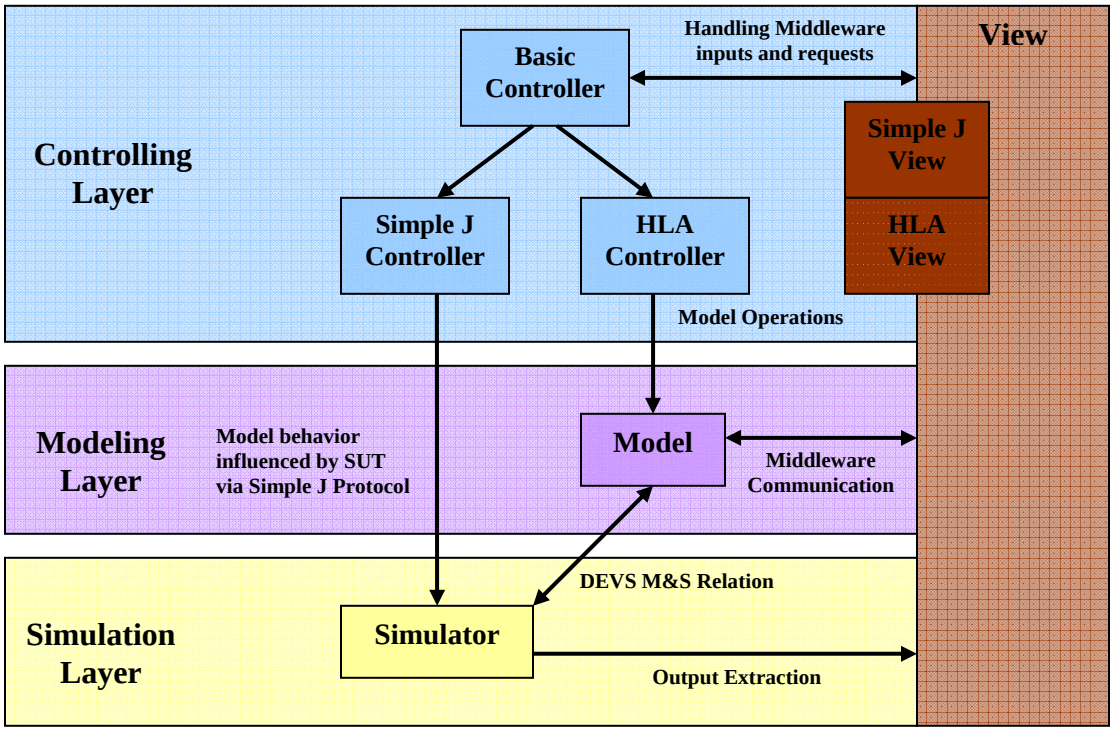


Figure 21: Enhanced MSVC paradigm with multiple controllers

It is common for simulation software to support multiple network simulation protocols. In a distributed testing environment, there are combinations of test components, such as simulation software, gateways, and hardware. Each of these components is associated with different network simulation protocols or middleware. A test control manager is often used to control the basic operations of all the test components or hardware, sending operation

commands to control the component via a particular middleware. For example, the test control manager synchronizes the time and start/stop of all the test components via HLA, and each component is considered as a federate in an HLA federation.

The Test Driver is implemented based on the enhanced MSVC pattern design. It supports HLA middleware and the Simple J network protocol. The test model is provided by the TMG, and the model behaviors are generated by three atomic models. The view is capable of extracting outputs from the simulator, and provides inputs the basic controller. Model operations are controlled by the HLA controller via HLA middleware, and the simulator is controlled by a Simple J controller.

4.2.4 MSVC Components

The Test Driver is an experimental frame that interacts with the SUT. It generates input stimuli and injects them into the SUT. It also receives outputs from the SUT and determines whether the desired experimental conditions are met. TD incorporates the generator, acceptor, and transducer concepts of the experimental frame, and implements these concepts using the MSVC design pattern. The *holdSend* atomic model handles the input stimuli generation, and the *waitReceive* atomic model determines the condition of the experiment.

Test Driver adopts two concepts: plug-and-play and rapid modification. The MSVC design allows the Test Driver to execute any DEVS test models generated by TMG. The DEVS test models are a set of instructions and the simulator provides the methods to generate the model behaviors. Any new network simulation protocols can be easily plugged into the Test Driver, and a new controller and view can be rapidly developed to support this new protocol based on the existing controller and view.

4.2.4.1 Model

The Test Driver (TD) model consists of DEVS test models and middleware. The model uses the middleware to communicate with SUT, and the DEVS test models are generated by the Test Model Generator.

4.2.4.2 Simulator

The simulator is an agent that is capable of obeying the model instructions and generating model behaviors. Typically, given the initial state values for the state variables and time segments for all input ports at the model, the simulator will generate the corresponding state and output trajectories. In a hierarchical model structure, the atomic models are the simulators and the coupled models are the coordinators. At the top of the hierarchy, a root coordinator is in charge to initiate the simulation cycles.

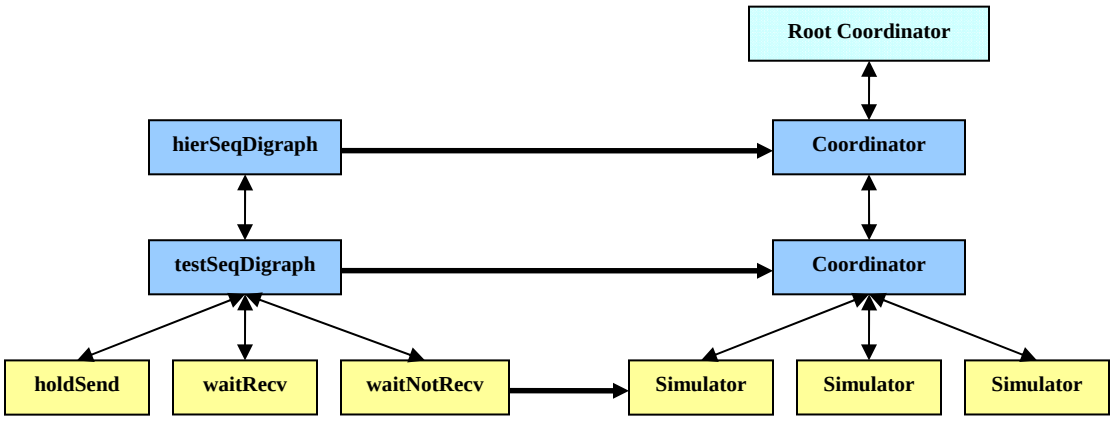


Figure 22: Mapping Test Driver model onto a hierarchical simulator

The hierarchy of the Test Driver model is shown in Figure 22. It illustrates the mapping from the test driver model onto the hierarchical model structure. The TD simulator is developed using ADEVS [13] simulation engine, and it consists of three atomic models and two coupled models. ADEVS is a simulator for models described using the Discrete Event System (DEVS) Specification modeling framework. It is a C++ library for constructing discrete event simulation based on the Parallel DEVS and Dynamic Structure DEVS formalisms. The Test Driver is

implemented using this simulation framework. All these models have two input ports and two output ports. The input ports are "start" and "in_Jmsg" and the output ports are "pass" and "out_Jmsg." The definitions of the ports are described as follows:

- "start:" All the models are passive at the beginning. When a model receives an external event through this port, the external transition function dictates a new system state s' with some new resting time ta(s').
- "in_Jmsg:" The Link-16 J message values are sent to the model through this port. A model must be in waiting state when receiving the in_Jmsg event; otherwise, the model will remain passive. When a model receives an external event through this port, the external transition function dictates a new system state s'' with some new resting time ta(s'').
- "pass:" This output port sends a start event to the next model. When the resting time (e = ta(s)) is expired, the system generates the start output, $\lambda(s)$, and changes to state $\delta_{int}(s)$.
- "out_Jmsg:" This output port sends the Link-16 J message generated by the *holdSend* model to the "out_Jmsg" port of the coupled model. When the resting time (e = ta(s)) is expired, the system generates the Link-16 J message output, $\lambda(s)$, and changes to state $\delta_{int}(s)$.

holdSend

holdSend is an atomic model that generates Link-16 J messages. The constructor of *holdSend* consists of five fields: model name, message type, wait time, message, and flag. The flag indicates the initial state of the atomic model. If the flag is true, *holdSend* will start at the "Send" state; otherwise, it will start at the "Passive" state. The elapsed time of the state is given by wait time. When the resting time expires, $\lambda(s)$ generates the two outputs: a Link-16 J message based on message type and message, and a start signal. Figure 23 shows the state transition of the *holdSend* model.

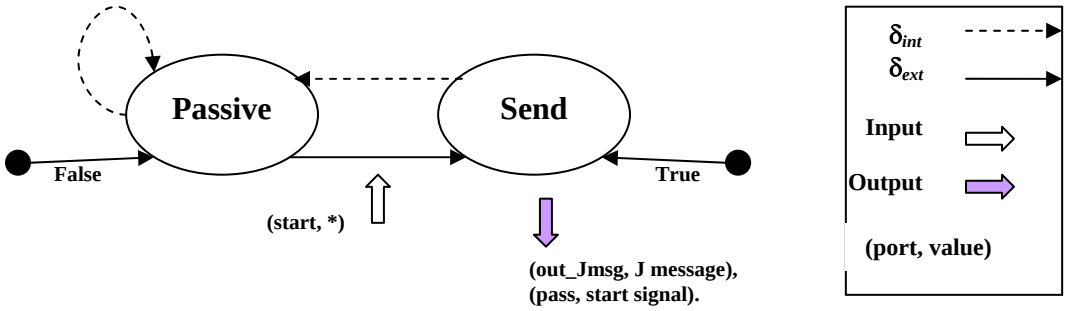


Figure 23: *holdSend* State diagram

Similarly, other atomic components like waitRecive and waitNoReceive are described and can be seen in the thesis [14]

testSeqDigraph

testSeqDigraph is a coupled model that contains a sequence of atomic models. It is equivalent to the BasicTestModel element described in the test model XML file. The rules to build the testSeqDigraph coupled model are:

- Multiple *holdSend* atomic models can be used, but *waitReceive* and *waitNotReceive* can only used once in each coupled model.
- If a coupled model starts with the *holdSend* model(s), it must end with either the *waitReceive* or *waitNotReceive* atomic model.
- If a coupled model starts with either *waitReceive* or *waitNotReceive*, no other atomic model can be added to this model.
- Since the atomic and coupled models execute sequentially, the first atomic model in the first coupled model must be *holdSend*, and the start flag should be set to true.

hierSeqDigraph

hierSeqDigraph is a coupled model that contains a sequence of testSeqDigraph coupled models. It is equivalent to the CompositeTestModel element described in the test model XML file. This coupled model is also known as the DEVS test model. It describes the procedures of a particular test scenario in a hierarchical structure.

4.2.4.3 View

The viewer extracts the output from the simulator and provides inputs to the controller. In the Test Driver, we have the Simple J viewer and the HLA viewer. The Simple J viewer extracts output from the simulator and provides inputs to the Simple J controller when the model receives J messages. The HLA viewer provides inputs to the HLA controller when the model receives HLA messages. In return, the HLA controller controls the model's operations.

When the Test Driver adopts a new network simulation protocol, a new viewer can be developed by copying and modifying the existing viewer. The new protocol usually will provide a set of methods to handle the operations, and the software engineer will replace the old operation methods with the new one.

4.2.4.4 Controller

The Basic Controller translates the information received from the network simulation protocols and routes it to the correct controller. There are two controllers derived from the basic controller: the Simple J controller and the HLA controller. The Simple J controller routes the messages to the DEVS simulator and the HLA controller routes the HLA messages to the model.

In the original MSVC design, the simulator is derived from the controller and each controller can only have one simulator. This one-to-one relationship is still valid in the enhanced MSVC design with a minor modification: the model can be controlled by the controller. If the Test Driver supports multiple models, there will be multiple controllers to control multiple simulators and models.

4.3 Activity Coordinator

An activity coordinator is implemented to manage the communications between MSVC components using the Event Notification and Mediator pattern [15,16]. An activity coordinator object acts as the mediator that links the coordinator to the simulator, views, and controller. The UML diagram shown in Figure 24 describes the primary objects and their relationships.

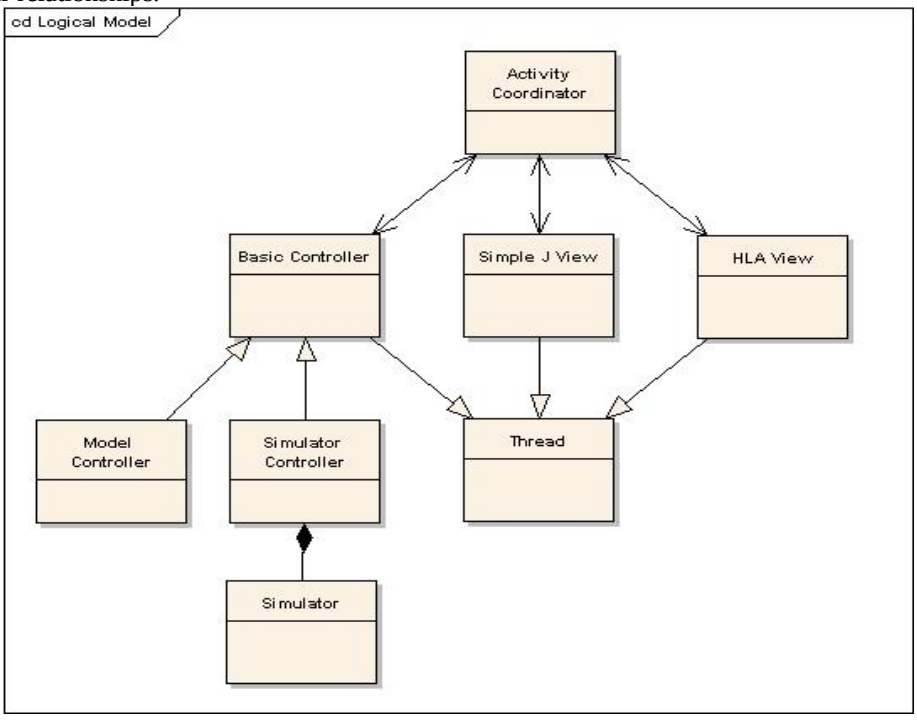


Figure 24: Test Driver's primary objects and their relationships

The viewer objects and controller objects are the specialization of a thread class. The thread class supports independent execution of each object and coordination via inter-thread events. The simulator object is derived from the controller. The inter-thread events are exchanged through the activity coordinator. There are two operations running in the Test Driver:

1. The HLA viewer receives instructions from the test control federate via the HLA middleware. It communicates with the HLA Controller via an inter-thread event, and the controller handles the operations of the model.
2. The Simulator generates Simple J messages. The J messages are passed to the Simple J viewer through the activity coordinator via an inter-thread event. When the viewer receives an incoming message from the Simple J network, the message is passed to the Simulator through the activity coordinator via inter-thread event.

5. Experiment and Results

In this section, an auto correlation experiment and the current testing status will be shown. The auto correlation experiment is conducted using the automated test generation processes described in section 4. The scenario was performed against the Integrated Architecture Behavior Model (IABM) developed by the Joint SIAP System Engineering Organization (JSSEO). The result of this scenario was verified by the ATC-Gen Test Driver and validated using JITC's Simple J network packet monitoring tool. Due to the classification of this system, the experimental results can not be shown. Thus, the System under Test (SUT) test models are developed to allow the test driver to act as the SUT and allow the experiment to be conducted.

5.1 Auto Correlation Scenario

As MIL-STD 6016C stated, when a system receives a remote track from a remote system that is within the correlation window of the local track, it initiates the tentative correlation process. If a second track arrives within the local track correlation window, it shall be correlated and held as common local track by transmitting a correlation request to the remote system. If the local track number is greater than the remote track number, the local system drops its own track and sends out a drop track notification; otherwise, the remote system drops its track and sends out the notification. Figure 25 illustrates the auto correlation process in the sequential diagram.

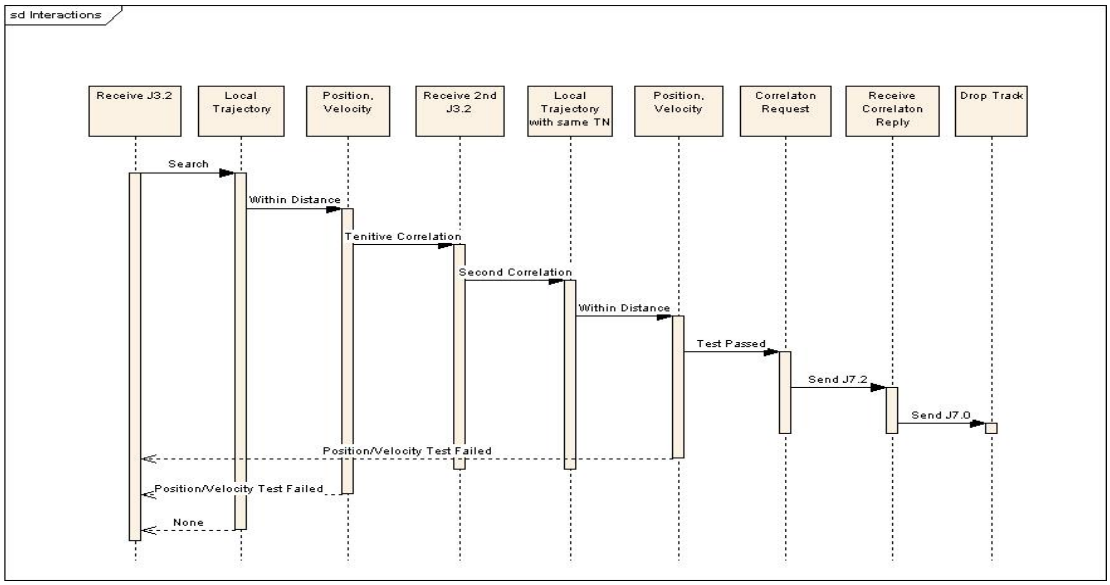


Figure 25: Auto Correlation Sequential Diagram

The test engineer follows the sequential diagram to construct the minimal testable pairs. Furthermore, the test models are generated using the Test Model Generator. Figure 26 illustrates the minimal testable pairs for SUT and Test Driver.

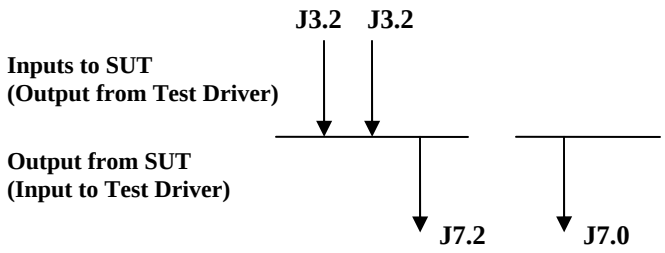


Figure 26: Minimal Testable I/O pairs for Auto Correlation

5.2 Auto Correlation Experiment Setup & Results

The auto correlation scenario is created to demonstrate the correctness of the models generated by the Test Model Generator. The models are implemented into the SUT and Test Model Test Drivers and communicate via Simple J protocol as illustrated in Figure 27. The transmissions and the receipt of the Simple J messages of the scenario are captured by a Simple J network packet monitoring tool. The packet monitor captures and decodes the Simple J messages, and the messages are saved into a log file. The log file is analyzed and the data is verified to ensure that the scenario data is the intended behavior of the Test Driver.

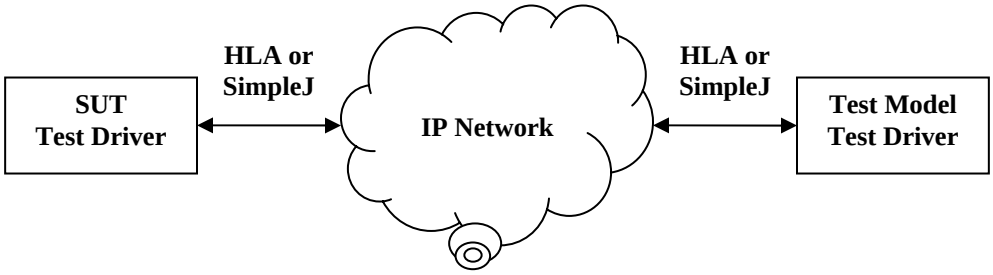


Figure 27: Test Drivers Setup Diagram

5.2.1 Successful Auto Correlation

In this scenario, the Test Drivers are communicated via Simple J protocol. The messages setup in the correct sequence and auto correlation is induced. The SUT models and Test Models are generated by the Test Model Generator, and implemented into the Test Driver. The SUT TD has the track number of 03000, and the Test Model TD has the track number of 00500. The two J3.2 track positions of the Test Model TD are exactly the same as the SUT J3.2 track position. This causes the tracks to correlate and creates a common local track with the track number of 00500. The SUT TD sends a correlation request and drops the local track with the track number of 03000. Figure 28 illustrates the outputs from the Test Model TD, and Figure 29 illustrates the results for the SUT TD.

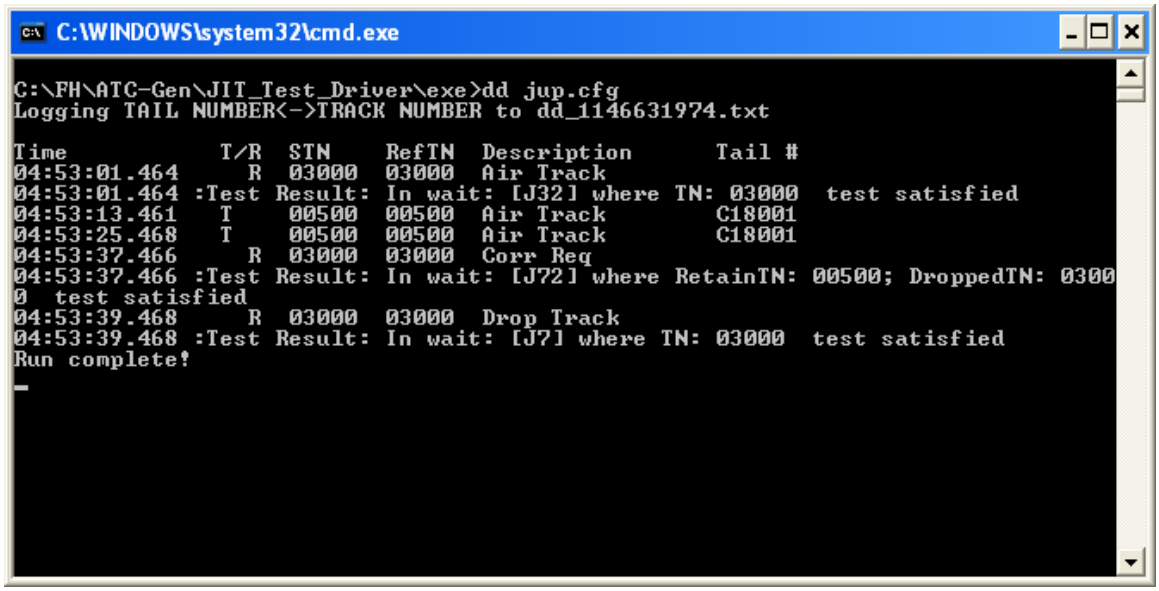


Figure 28: Test Model Test Driver successful Auto Correlation scenario

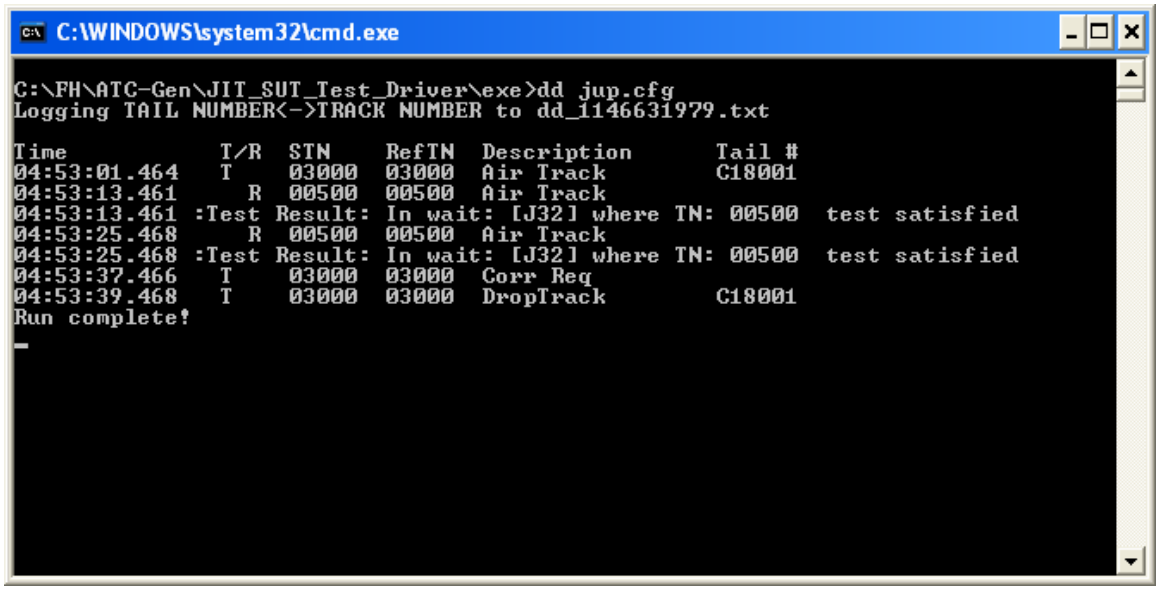


Figure 29: SUT Test Driver successful Auto Correlation scenario

5.3 Testing Status

ATC-Gen Test Driver was tested in both standalone and distributed environments. In the standalone environment, it performed Link 16 testing against two Link-16 systems: IABM and Air Defense System Integrator (ADSI). Recently, ATC-Gen Test Driver was participated in a distributed live testing environment in JITC. Table 2 summaries the results of the Link 16 functionalities against the systems.

Link 16 Systems MIL-STD 6016C Functions	IABM	ADSI	Distributed Environment
AutoCorrelation	Y	Y	Y
Correlation Window Size	N	Y	Y
Decorrelation	Y	Y	Y
Track Management	Y	N	N
Report Responsibility	Y	Y	Y
Track Quality	Y	Y	Y
Identity Different Resolution	N	Y	Y

Table 2: Link 16 functionalities vs. Systems

6. Discussion

This section provides a discussion about the existing multi-level architecture that supports interoperability testing and their comparison with our approach of using DEVS.

The M&S community has developed the SISO-STD-002-2006 for Link 16 simulation [17]. This standard is not applicable to ATC-Gen, because the ATC-Gen core software was built before the development of the standard.

The Levels of Information Systems Interoperability (LISI) was introduced in 1998. It is a set of models and associated process based on five levels of interoperability described in Table 3. Although the LISI models are used successfully to determine the degree of interoperability between information technology systems, they do not provide a systematic formulation of the underlying properties of information exchange [18].

Level of Information System Interoperability	**Characteristic**	**Information Exchanges at the Level**
Enterprise	Shared data and applications	Global Information Grid (GIG) web–services, service-oriented architecture, advanced collaboration
Domain	Shared data, separate applications	Access to common databases, sophisticated collaboration
Functional	Minimal common functions, separate data and applications	Annotated images, maps with overlays
Connected	Electronic connection	Tactical data links, email, file transfer

Table 3: Levels of Information Systems Interoperability

To remedy this situation, Level of Conceptual Interoperability (LCIM) [19] was developed to become a bridge between conceptual and technical design for implementation, integration or federation [20, 21]. LCIM consists of these layers: Conceptual, Dynamic, Pragmatic, Semantic, Syntactic, Technical and Standalone. These layers provide various levels of interoperability that could be considered between different systems.

Zeigler [18] defines a three layered approach that provides the intended interoperability instead of 7 layers in LCIM, viz., Pragmatics, Semantics and Syntactic levels. They are defined as:

- Pragmatics: Data use in relation to data structure and context of application
- Semantics: Low Level semantics focuses on definitions and attributes of terms; high level semantics focuses on the combined meaning of multiple terms
- Syntactic: This focuses on a structure and adherence to the rules that govern that structure

The authors of LCIM associate the lower layers with the problems of simulation interoperation while the upper layers relate to the problems of reuse and composition models [22,23]. Our MSVC architecture that provides a layered framework for models and simulators provides the needed solution. Further, our earlier work [12] put this

layered architecture in perspective of Department of Defense Architecture Framework (DoDAF), which is mandated for all governmental information technology systems and a common denominator of understanding between various DoD agencies. The MSVC architecture, as shown in Figure 21, is one more implementation of this layered architecture. In our recent work, this layered architecture has been extended to work on Service Oriented Architecture using the DEVS Modeling Language (DEVSML) [24] where the test federations can be deployed using GIG. The DEVSML has the capability to inter-convert between DEVS and XML. It allows distributing DEVS models in the form of XML documents to remote nodes where they can be coupled with local services components to compose a federation [25].

7. Conclusion

A new automated testing approach has been successfully developed using System Entity Structure, the Extensible Markup Language, the Discrete Event System Specification, and the Model/Simulator/View/Controller design pattern. The hierarchical structures of the SUT scenarios and Test models are represented by SES and written in XML format. XML DTDs are developed based on the SES to verify the correctness of the XML files. The processes of automated testing approach are defined as follows:

1. The SUT scenario is constructed by the test engineer based on the system and the test requirement using the Minimal Testable Input/Output concept.
2. DEVS test models are developed using the model mirroring by reversing the minimal testable pairs of the SUT.
3. DEVS programming source codes are generated based on the test models.
4. The DEVS source codes are implemented into the Test Driver.
5. Test Driver executes the models and experiments against a real or simulated system.

The automated testing approach is developed to perform conformance testing on the military TADIL-J systems. This approach combines the system theory, the DEVS modeling and simulation framework, and the model continuity concepts to formulate and develop DEVS models. It promotes the separations of models and simulator, which allows model reuse and develops models independently of the simulation engine. The Test models are developed using the system specifications and DEVS framework by collecting the input/output pairs with the initial states and describing the I/O behaviors in DEVS. The simulators are well-defined for reusability and implemented according to the system behavior.

MSVC design pattern used in the Test Driver provides a model for building distribution simulation for the automated testing. MSVC promotes the component-based design and the reusability of the simulation software. By applying this design pattern in conjunction with DEVS modeling and simulation framework, Test models and the simulators are developed separately, and we can attach any network simulation protocols to the simulation. The models are expressed in the DEVS formalism, and the simulators are associated with ADEVS simulation engine to execute the models. The well defined semantics of the DEVS modeling and simulation formalism allows the simulator to be encapsulated and reused. The Test models developed under the automated testing guidelines are able to be executed by the Test Driver.

MSVC design architecture is extensible and is positioned correctly to provide dynamic collaborative DEVS model composition and execution over Net-centric infrastructure such as Global Information Grid and Service Oriented Architectures. MSVC architecture in conjunction with the DEVSML and DEVS/SOA capabilities allows the multi-level interoperability requirements to be dealt in a methodical and layered manner.

The automated testing approach was used to verify the conformance of the Integrated Architecture Behavior Model (IABM) to the MIL-STD 6016C, and the results of the test scenarios were validated using the Simple J network packet monitoring tool. The SUT/Test model method was introduced in this thesis to verify the correctness of the DEVS models. The transmissions and the receipts of the Simple J messages were captured by the packet monitoring tool. The system analyst interpreted and verified the messages, and determined whether these messages were the intended behavior of the Test Driver.

7. Future Work

Currently, the DEVS Test models are written in C++ source code. When the models are changed, the Test Driver requires to re-compile the source code. An XML Test models shall be developed to eliminate the recompilation and

achieve testing automation. The simulators are well-defined based on the system behavior. The Test Driver parses the XML models, generates the coupling and model behaviors on the fly, and executes the models to experiment with SUT.

The Test Driver is expanding into two testing modes: Reactive and Passive modes. In reactive mode, the Test Driver will receive and copy the incoming J3.2 message, and re-transmit this message to the link with its own track number. The reason for the reactive mode is because the track positions from the Common Reference Scenario (CRS) file are different when we are testing different military systems. In order to test the auto correlation function, we need to control the track location to induce correlation. In passive mode, the Test Driver will monitor a specific system by listening and receiving TADIL-J messages from all the systems in the testing network. It will use the test detector concept to determine whether the monitored system passes the certain Link 16 conformance tests.

Ultimately, the Test Driver will be expanded into the distributed environment. All the testing introduced so far are in the one-to-one environment, and the Test Driver is always being tested against a particular military system. In the future, the TD will expand into the one-to-many environment and will be able to test multiple military systems simultaneously.

References

[1] Zeigler, B. P., Fulton, D., Nutaro, J., Hammonds, P., "M&S Enalbed Testing of Distributed Systems: Beyond Interoperability to Combat Effectiveness Assessment", 9th Annual Modeling and Simulation Workshop, Dec. 8-11, 2003, ITEA White Sands Chapter

[2] Technology for the United States Navy and Marine Corps, 2000-2035 Becoming a 21st-Century Force: Volume 9: Modeling and Simulation (1997), National Academy Press.

[3] Modeling and Simulation in Manufacturing and Defense Acquisition: Pathways to Success (2002), National Academy Press.

[4] B.P. Zeigler, D. Fulton, P. Hammonds, J. Nutaro, "Framework for M&S-Based System Development and Testing in Net-centric Environment," ITEA Journal of Test and Evaluation, Vol. 26, No. 3, 2005.

[5] B.P. Zeigler, H. Praehofer, T.G. Kim, "Theory of Modeling and Simulation," Academic Press, 2000.

[6] Zeigler, B.P., Ball, G., Cho, H., Lee, J.S., Sarjoughian, H., "Implementation of the DEVS Formalism over the HLA/RTI: Problems and Solutions," Spring Simulation Interoperability Workshop, 1999.

[7] Extensible Markup Language, http://www.w3.org/XML, last accessed April 15, 2006.

[8] J.W. Rozenblit and Y.M. Huang, "Rule-Based Generation of Model Structures in Multifaceted Modeling and System Design," ORSA Journal on Computing, Vol. 3, No. 4, Fall 1991

[9] Chairman, JCS Instruction 6212.01C "Interoperability and Supportability of Information Technology and National Security Systems," 20 November 2003.

[10] H.S. Song, and T.G. Kim, "The DEVS framework for Discrete Event Systems Control," In Proceeding of the fifth Annual Conference on Al, Simulation, and Planning in High Autonomy Systems, pp. 228-234, 1994

[11] James Nutaro, Phil Hammonds, "Combining the Model/View/Control Design Pattern with the DEVS Formalism to Achieve Rigor and Reusability in Distributed Simulation," Journal of Defense Modeling and Simulation: Applications, Methodology, Technology, pp. 19-28, Vol. 1, No. 1, 2004

[12] Saurabh Mittal, Eddie Mak, and James Nutaro, "DEVS-Based Dynamic Model Reconfiguration and Simulation Control in the Enhanced DoDAF Design Process," submitted to Journal of Defense Modeling and Simulation, 2005.

[13] Nutaro, J., A Discrete Event Simulator, http://www.ece.arizona.edu/~nutaro, last accessed April 15, 2006

[14] Eddie Mak, "Automated Testing using XML and DEVS," http://www.acims.arizona.edu, last accessed June 30, 2006

[15] Riehle, D., "The Event Notification Pattern – Integrating Implicit Invocation with Object Orientation," Theory and Practice of Object Systems, Vol. 2, No. 1, pp. 43-52. 1996.

[16] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns: Elements of Reusable Design," Addison-Wesley. 1995.

[17] SISO (2006), "Standard for Link 16 simulation," SISO-STD-002-2006, 8 May 2006.

[18] Bernard Zeigler, Phil Hammonds, "Modeling and Simulation-Based Data Engineering", Academic Press, 2007

[19] Andreas Tolk, J.A. Muguira, “The Levels of Conceptual Interoperability Model (LCIM)”, Proceedings of the Fall Simulation Interoperability Workshop, 2003
[20] C. Turnista, “Extending the Levels of Conceptual Interoperability Model”, Proceedings IEEE Summer Computer Simulation Conference, 2005
[21] J. Muguira, A. Tolk, “Applying Methodology to Identify Structural Variances in Interoperations”, JDMS: The Journal of Defense Modeling and Simulation, Vol.3, No.2, 2006
[22] M. Hoffman, “Challenges of Model Interoperation in Military Simulations”, SIMULATION, Vol.80, pp. 659-667, 2004
[23] E. Chaum, M.R. Hieb, A. Tolk, “M&S and the Global Information Grid”, Proceedings Interservice/Industry Training, Simulation and Education Conference (I/ITSEC), 2005
[24] Saurabh Mittal, José Luis Risco Martín, Bernard P. Zeigler, “DEVS-Based Simulation Web Services for Net-centric T&E”, Summer Computer Simulation Conference, SCSC'07, San Diego, July 2007
[25] Saurabh Mittal, “Extending DoDAF to Allow DEVS-Based Modeling and Simulation”, JDMS: Journal of Defense Modeling and Simulation, Vol.3, No.2.