AUTOMATED TESTING USING XML AND DEVS

By

Eddie Chee-Hung Mak

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE
WITH A MAJOR IN ELECTRICAL AND COMPUTING ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

2006

STATEMENT BY AUTHOR

SIGNED: ___________________________

APPROVED BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

_________________________________ ______________________

Bernard P. Zeigler Date

Professor of Electrical and Computer Engineering

ACKNOWLEDGEMENTS

I would like to thank my advisor, Professor Bernard Zeigler, for the opportunity and years of mentoring and encouragement. I would also like to thank Dr. Moon-Ho Hwang for correcting and improving my thesis content and Dr. Salim Hariri for the time taken to serve on the thesis committee.

Thanks to my family, Angelina, Natalie, and Emilie for their patience and support.

Special thanks to Dr. Jerry Couretas, who encouraged me to continue my education from the beginning.

My sincere gratitude to Dr. James Nutaro, John Lee, Saurabh Mittal, and Fahad Bait-Shiginah, who helped me in my research, thesis correction, and presentation preparation.

Finally, I would like to thank Dr. Philip Hammonds and everyone in the ACIMS lab and the JITC ATC-Gen team for the support.

Table of Contents

List of Figures

List of Tables

Abstract

With the modernization of Department of Defense (DoD) systems and the growing complexity of communication equipment, traditional test methods and processes have no choice but 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 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 thesis, we present a new methodology to generate the test models and perform conformance testing using system theory, the DEVS modeling and simulation 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 the Theater Air and Missile Defense (TAMD) Interoperability Assessment Capability (TIAC) tool. The TIAC tool 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

In the 1990s, the military began using simulation to enhance its training exercises. The extensive use of simulation in military exercises resulted in significantly improved training progress, 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) [21]. Since then, system development has increasingly made use of modeling and simulation, and the performance of testing on military systems has become a greater challenge.

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 transition gateways. With the increased system complexity, testing methodology has to become more rigorous, in-depth, and thorough.

As detailed in recent DoD reports [1,2], 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. In this thesis, 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 military simulation training and developments described above was successful, which led directly to more intensive use of modeling and simulation in system development and a revised policy for acquiring new systems known as Simulation-Based Acquisition. This required the using of modeling and simulation in all phases of system development life-cycles. JITC took the initiative to employ modeling and simulation (M&S) to increase its simulation-based testing capabilities and automation of the testing processes, increasing the effectiveness and responsiveness of the testing [3]. 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.

The automated testing framework introduced in this thesis 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 advent of simulation-based acquisition, the test requirements in the simulated environment becomes how to automate and more precisely the scope, the completeness, and the methodology for updating conformance testing. The incorporation of the Systems Theory, the modeling and simulation concept, DEVS, XML, SES, and the MSVC design pattern all contribute to the development of ATC-Gen to automate the TADIL-J conformance testing, increasing the productivity and effectiveness of the software test tools at JITC.

Many testing methodologies were developed to assist the test engineers to perform conformance testing over the past decades [21]. 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 System Modeling Concepts

In this section, we review the system theory [4], the M&S framework [4], and model continuity [4]. The automated testing approach is developed according to these concepts.

1.2.1 System Specifications

The hierarchy of System Specifications employs a general concept of dynamical system and defines levels at which a system may be known or specified and provides a mathematical underpinning to define a framework for modeling and simulation. Table 1 shows the Hierarchy of System Specifications [3].

1. 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

1.2.2 Framework for Modeling and Simulation

The modeling and simulation framework defines entities and their relationships that are central to the M&S enterprise [4]. The basic entities of the framework are source system, model, simulator, and experimental frame as illustrated in Figure 1, and they are linked to the modeling and simulation relationships.

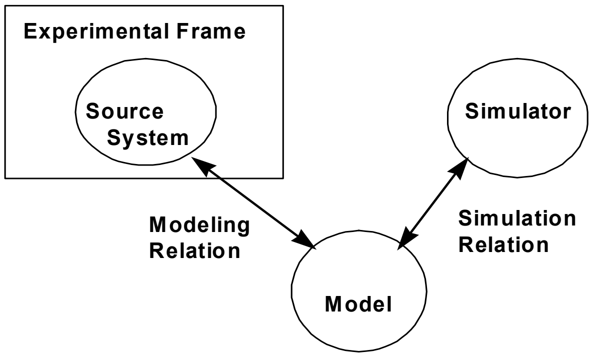


Figure 1: Basic M&S Entities and their relationships

1.2.3 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.

1.3 Plan of Thesis

The next chapter of the thesis discusses the background of ATC-Gen, High Level Architecture, and distributed simulation. In Chapter 3, the system theory used to create the input/output pairs is discussed, and the transformation from I/O pairs to test models is illustrated. Chapter 4 presents the generic HLA wrapper, which allows rapid modification of the simulation software to reduce the development time in HLA simulations. Chapter 5 discusses the use of MSVC design patterns to develop simulation software. The MSVC pattern enhances the reusability of the simulation software and allows for the test models, the simulators, and the generic HLA wrapper to be developed separately. Four real test scenarios are illustrated; the method of SUT test models development is shown in Chapter 6. These scenarios were tested by the Joint SIAP System Engineering Organization (JSSEO) using the Integrated Architecture Behavior

Model (IABM) and verified by the ATC-Gen Test Driver. The experimental results in this thesis were run against the SUT Test Driver due to the fact that the IABM is classified by the Department of Defense. Finally, Chapter 7 concludes the thesis with the discussion of future work.

2. Backgrounds

2.1 Automated Test Case Generation (ATC-Gen)

ATC-Gen is composed of several stages that are developed in conjunction with DEVS formalism. 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.

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 as Figure 2, and these rules are not associated with time.

If X is true,

Then do action Y later

Figure 2: IF-THEN rule format

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 customer present. 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 no customer.
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 two state variables in the above example: money and item. 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. XML uses a generic syntax to markup the document with tags, and enhances the structure of the information and identifies the relationship in the document. An XML document can be used in many different applications and allows for the document to be queried a meaningful way. But first, a XML Document Type Definition (DTD) or schema must be created to validate and provide the correct syntax to the XML document. Tags are the legal building blocks of the XML document as shown in Figure 3. Each statement in the below figure 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 XML format as follows:

```
<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="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 is a DEVS tool and provides a visual display of dependencies, allowing selection of test sequences by the test engineer. 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 and Test Model Generation, 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 consists of running the DEVS test model against a real hardware/software system. 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.

2.2 High Level Architecture (HLA)

2.2.1 HLA Overview

Traditional simulation models have often lacked reusability and interoperability. The High Level Architecture (HLA) [6] is a standard framework developed by the Defense Modeling and Simulation (DMSO) of the Department of Defense (DoD) to support component simulation models, in which the models can be reused and combined in distributed simulation. HLA consists of three components: Federation Rules, the Interface Specification, and the Object Model Template. There are ten HLA rules which must be obeyed in order to be regarded as HLA compliant, consisting of five federation rules and five federate rules. The federation rules are the ground rules for creating a federation, and federates are governed by the federate rules. The HLA Interface specification is implemented by the Run-Time Infrastructure (RTI). RTI is a software application which provides common services to support an HLA-compliant simulation.

The RTI services separate simulations and communication, and provide communications and management between federates. The Object Model Template (OMT) provides a standard for documenting HLA Object Model information. All objects and interactions managed by a federate and visible out the federate should be specified in the OMT format. It provides a common mechanism for specifying the data exchange and coordination between federates and describing the capabilities of potential federates. The OMT defines the Federate Object Model (FOM), the Simulation Object Model (SOM), and the Management Object Model (MOM). Each federation is implemented according to the FOM, and federates are implemented according to the SOM. FOM specifies the data exchange among federates, and SOM specifies the capabilities of the federates provided to the federation.

The HLA Interface specification is divided into 6 management areas. Each area plays an important role between federates and their associated federations. The basic functions of each management area are:

1. Federation Management – coordinates activity and manages federation execution.
2. Declaration Management – coordinates data execution and specifies data type send and receive.
3. Object Management – creates, manages, modifies, and deletes objects. It also coordinates attribute updates.
4. Ownership Management – supports transfer of ownership for object attributes.
5. Data Distribution Management – coordinates information routing.
6. Time Management – coordinates federate time advances.

Figure 4 [6] shows the interactions of RTI, federations, and federates in the simulation world. Each federation has its own OMT and is described in the FED file format. Furthermore, each federate within a federation is created according to the OMT. RTIExec and FedExec are RTI components. RTIExec manages the creation and deletion of federations, while FedExec manages federates joining and resigning from the federation, and manages data exchanges between federates. RID stands for RTI Initialization Data. It provides information to run an RTI.

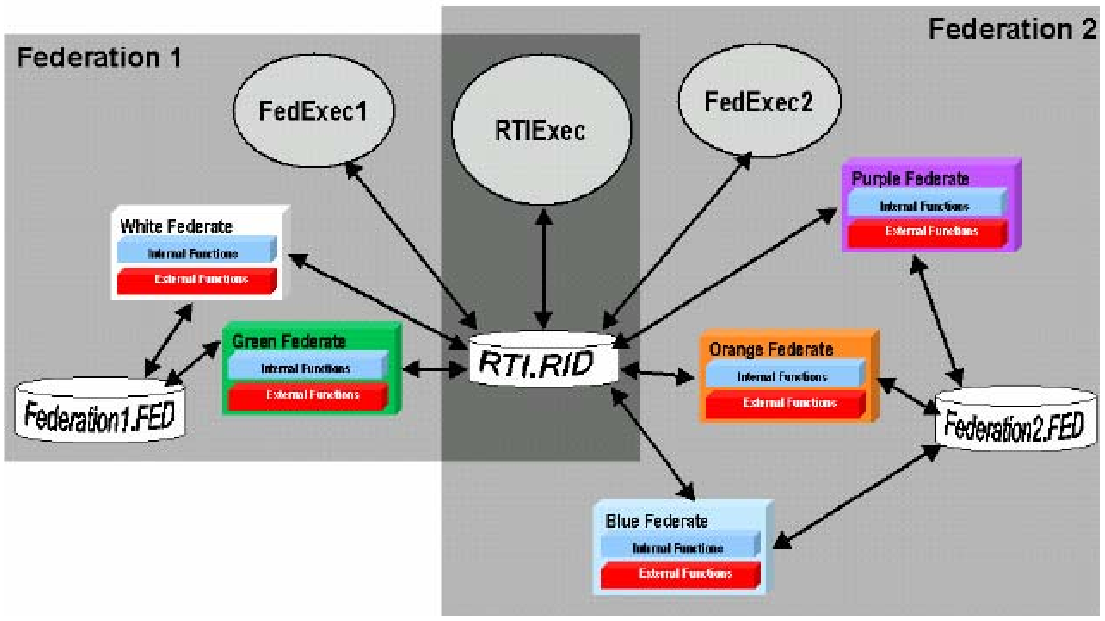


Figure 4: HLA Simulation

There are several processes which can initiate and run the HLA simulation. First, the software programmer develops a federate based on the OMT specification. Second, the user starts "RTIExec" before executing the federate. If the federation already exists, the federate will join the federation automatically. Otherwise, the federate acts as a manager and creates a federation execution, and then joins the federation. The messages distributions are managed by Publish and Subscribe methods. Each federate performs a service, and it uses Publish and Subscribe to exchange data between federates. The Publish method is used to publish the object classes, object attributes, and interaction classes to the RTI; each federate indicates an interest in certain object classes/attributes and interaction classes it wishes to receive by invoking the Subscribe method in RTIambassador. Third, the registration, updates, discovery, and reflection of the object classes are handled by object management, which also handles sending and receiving for interaction classes. In order for the federation to be aware the existence of a federate, it is required to register the object instance to the RTI. Once the object exists, RTI informs other federates that are interested in this particular object by using the Discover method. When the federate updates the attributes associated with a registered object, the federate encodes the data and calls the Update method to inform RTI, which in turn uses the Reflect method to update the data to the interested parties. Interactions use the Send and Receive methods to handle the parameters instead of Update and Reflect, the difference being that in the latter the object persists, but the interaction does not. An object can specify which attributes should be published, while an interaction requires publishing all parameters. Last, before terminating the federate, the objects/interactions are removed and the federate is resigned from the federation.

2.2.2 Object Oriented HLA Interface

HLA simulation development is quite different from developing DEVS-based simulations, and software engineers are often subjected to a learning curve in order to develop HLA simulations. There is the need for a core HLA design philosophy to guide the development of a robust and interoperable interface that goes beyond current HLA development. Lockheed Martin Advanced Simulation Center [7] had developed an object-oriented HLA interface that provides a clean conceptual and architectural separation of the simulation from distributed computing components. The HLA interface allows software engineers without extensive simulation experience to develop HLA compliance simulation, thereby accelerating the simulation development without forcing the engineers through the HLA learning curve.

2.3 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. 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.

2.3.1 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. Components are like building blocks, representing complete pieces of functionally that are ready to be used. When these blocks are combined together, a useful software application is created. 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 modeling and simulation, 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 [8]. This approach seems restrictive, but it provides a framework to validate the models and verify the simulators.

2.3.2 Supporting multiple network protocols

It is common to require a single simulation software application to support multiple network protocols in an interoperability testing environment, such as training exercises and systems evaluation. Interoperability testing often links multiple platforms or systems together in a heterogeneous environment, such as hardware in the loop systems, live systems, and other simulation software systems. These systems often support only selected network simulation protocols. To permit the simulation software collaboration in such an environment, the software must be able to support multiple network protocols 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.

3. Test Model Generator (TMG)

3.1 Building Blocks

3.1.1 Extensible Markup Language (XML)

Extensible Markup Language (XML) [9] 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 using human readable tags. 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, which is the most common object type of XML. They break up a document into a smaller structure and organize it into a hierarchical form. Elements can be empty, containers, or holding texts, etc. Tags are used to mark the boundaries of elements. An XML element consists of a start tag and an end tag. Additional data, such as comments and special instructions, can be specified in the tag. XML doesn't have a predefined set of tags and elements, and it works for all areas of interest. However, the XML standard specifies the format of tags and elements. It defines how tags should appear and where they may be placed. Also, it defines the element-naming scheme and where the attributes attach, and so forth. 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; the XML processor will reject the document if it is not well-formed.

Some programs may perform special operations and require the use of specific tags. These specific sets of tag are called applications of XML. An XML application is not a software program, but it uses specific tag sets in a particular operation. 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. An element declaration defines a new element type, the order, and what it can contain. Any element type not declared in the DTD but used in the document is illegal. The disadvantages are the complexity and the lack of restrictions on the actual data types in each element's content. An XML schema is an alternative to the DTD. It defines the building blocks of an XML document, and it is simpler, 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.

3.1.2 System Entity Structure (SES)

System Entity Structure (SES) formalism is a knowledge representation scheme for systematically organizing a family of 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 [10]:

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 5 illustrates the alternate modes of SES.

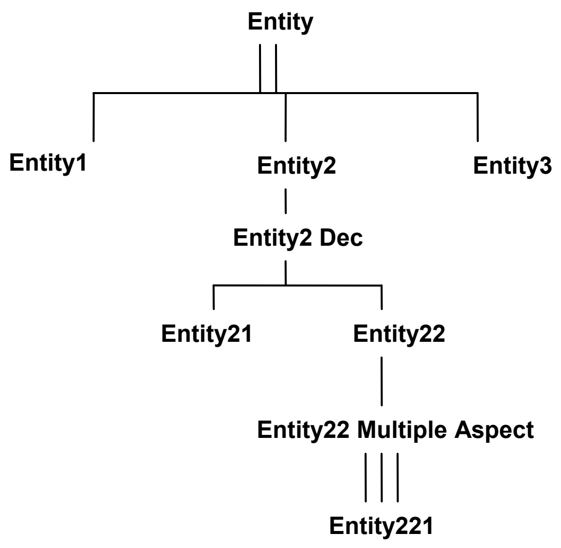


Figure 5: SES nodes structure

3.1.3 Minimal Testable Input/Output Pairs

In a basic operation, there are many possibilities to characterize the I/O behavior at Level 1 of the system specification, such as which input values are paired to produce an output value and the order of the inputs and output pairing [3]. Figure 6 shows the complexity involved in the Level 1. If the initial messages X(x) and Y(x) arrive, an output of Z(x+y) is produced. However, when the second message Y(y') arrives, the SUT can produce or not produce an output of Z(x+y'). If the second Y message does not produce an output, the SUT will produce an output Z(x'+y') if the second X(x') message arrives.

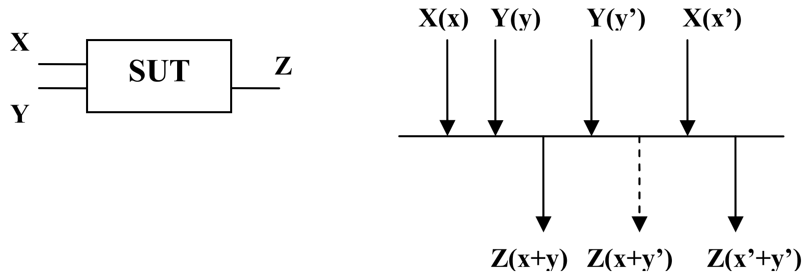


Figure 6: Variants of Input/Output Pairs

It is easier to handle I/O segments with limited complexity. The tests can be synthesized from the segments because each I/O pair has a finite number of input messages and output message in its interval. Thus, the concept of a minimal testable pair is introduced, in which the output segment has at most one event at the end of the segment in an I/O pair. The I/O pair can be easily extracted from the system specification at Level 2, because each input stimulus produces a unique output if the initial state is given. Figure 7 illustrates the minimal testable input/output pairs. The minimal testable I/O pairs are in sequential order, and the original I/O pair can be easily reconstructed from the minimal I/O pairs.

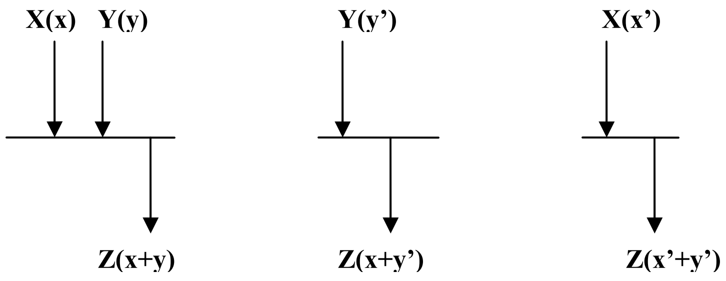


Figure 7: Minimal Testable Input/Output Pairs

3.2 Test Objective

The objective of the Test Model Generator is to a create DEVS test models based on minimal testable I/O pairs. In this thesis, we are performing a reachable states study 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 customers' test 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 8 below illustrates the process of automated test model generation.

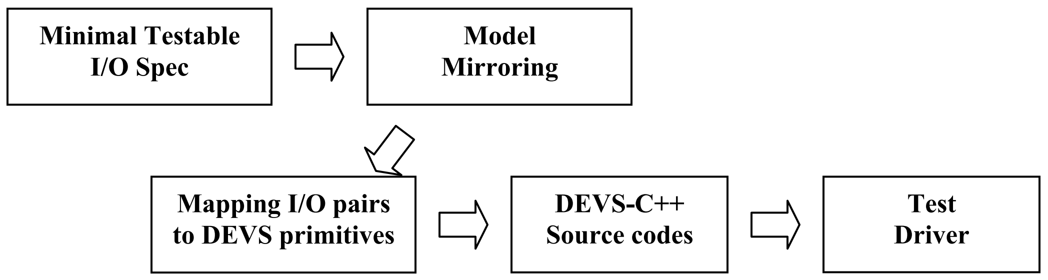
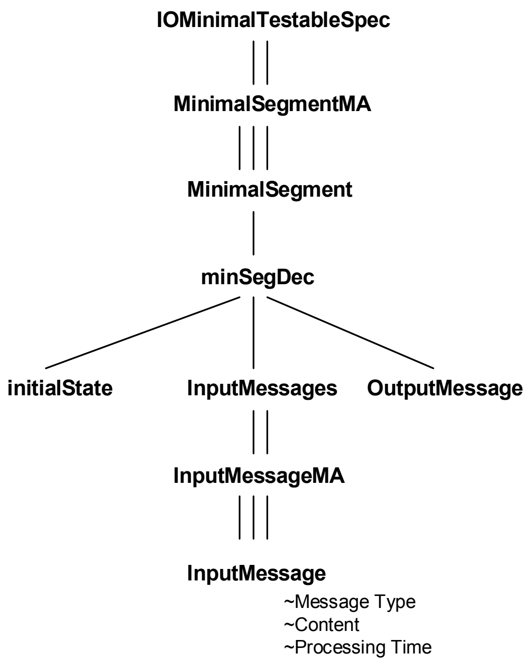

Minimal Testable I/O Spec
Model Mirroring
Mapping I/O pairs to DEVS primitives
DEVS-C++ Source codes
Test Driver

Figure 8: Test Model Generator

3.3 Test Model Generation

3.3.1 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 9 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 9: IO Minimal Testable SES Diagram & XML Tags

3.3.2 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 10. 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.

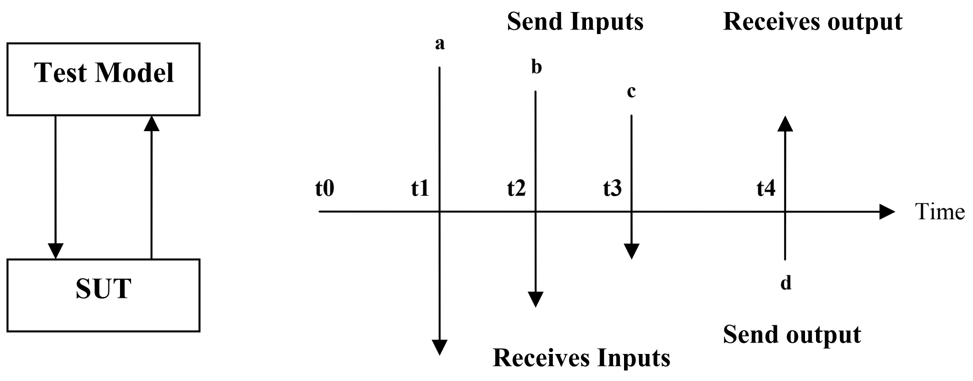


Figure 10: Mirror Image of Input/Output Pairs

3.3.3 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 11 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.

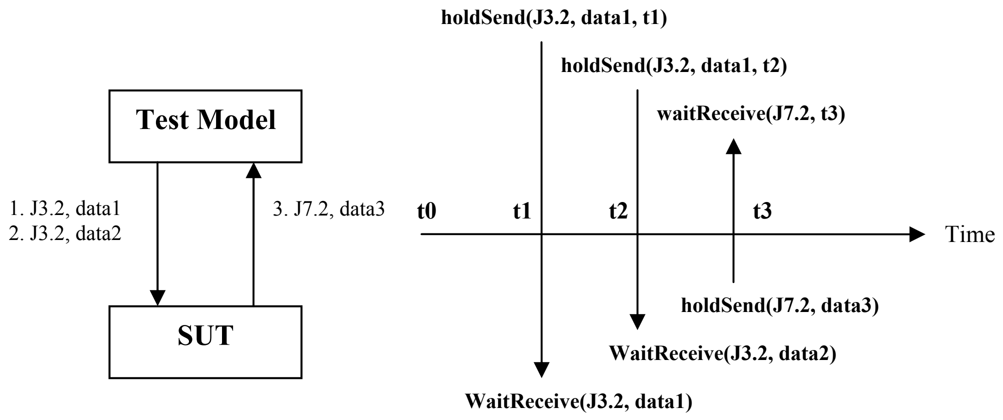


Figure 11: Using the primitives to construct test model

The test model is in the form of an experimental frame in which the holdSend model is a generator and the waitReceive is the acceptor. These atomic models are coupled together to form the basic test model. Each atomic model and coupled model has two input ports: start and in_Jmsg, and two output ports: out_Jmsg and pass. The inputs and output of the minimal testable I/O pair are in sequential order, and the basic test model is formed by coupling the atomic models together as shown in Figure 12.

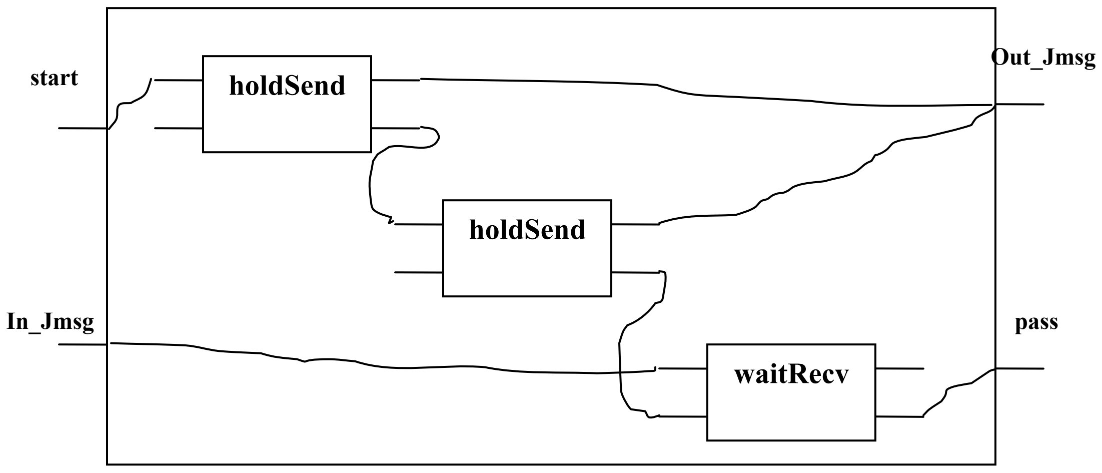


Figure 12: Basic Test model

3.3.4 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 13 illustrates the composite test model derived from cascading the basic test models. The initial correlation test model has three operations: transmit twice and receive, while the confirmation test model has two operations: transmit and receive. The atomic models of the basic test models are concatenated, and the basic test models of the composite model are cascaded together. 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.

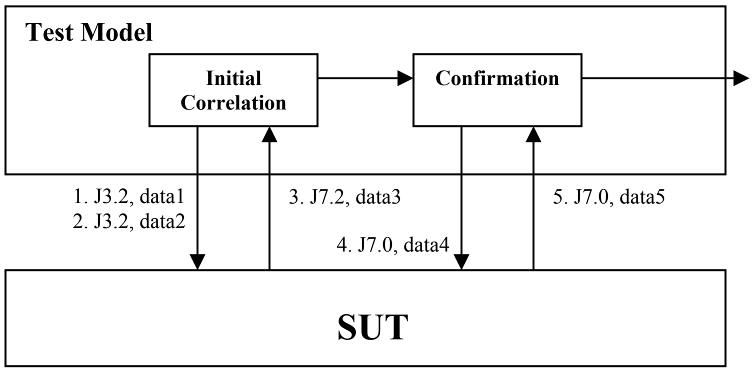


Figure 13: Composite Test model

Based on the composite model, we can concatenate 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.

3.3.5 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 14, 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.

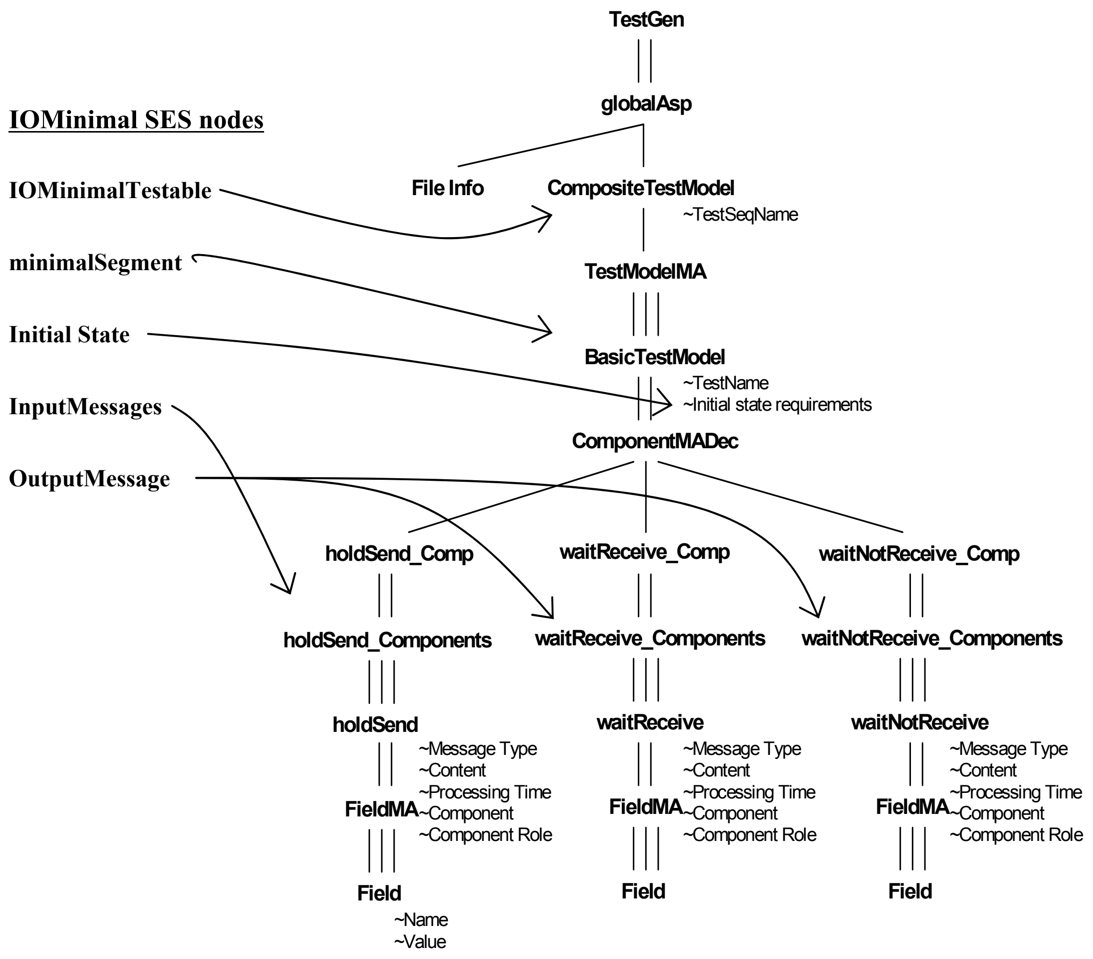


Figure 14: Mapping Minimal Testable pairs into Test Model SES

The above figure also illustrates the transformation 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.

3.3.6 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 14 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 uses the 'start', 'out_Jmsg', and 'pass' ports.
- waitReceive uses the 'start', 'in_Jmsg', and 'pass' ports.
- waitNotReceive uses 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 12, and CompositeTestModel contains the BasicTestModel coupled models as illustrated in Figure 13. 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 15 below illustrates the associations of these five methods and the test model SES diagram.

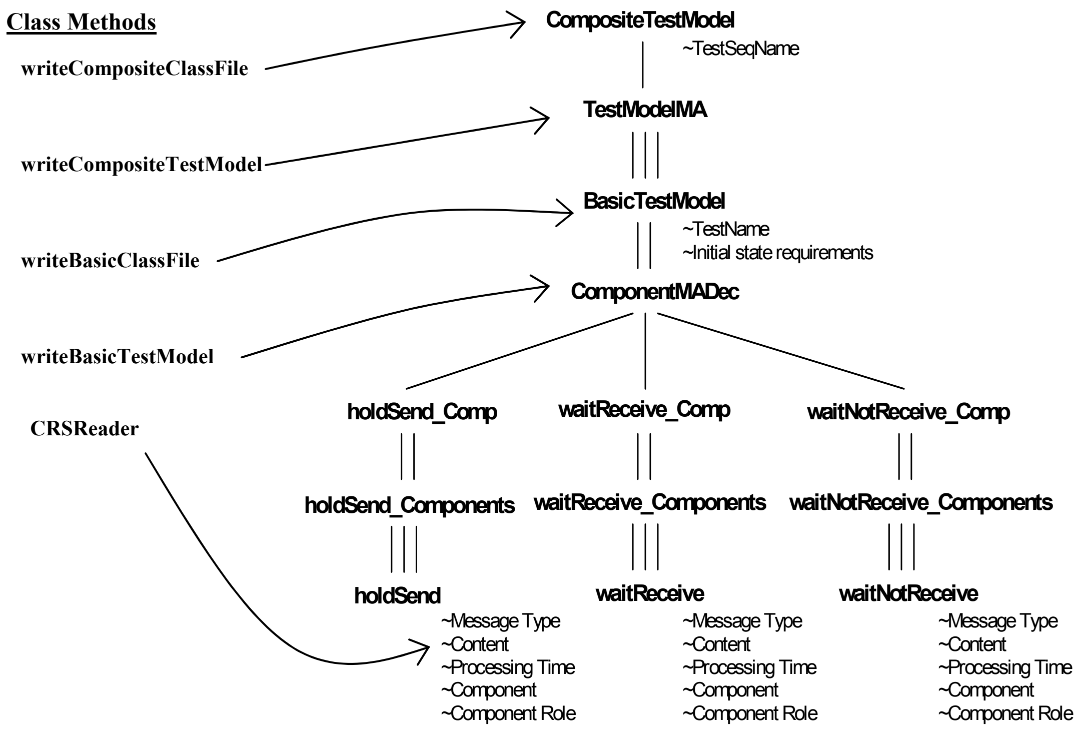


Figure 15: Associations between Java methods and SES nodes

4. Generic HLA Wrapper

The Generic HLA wrapper is developed to support the Test Driver (TD) in the ATC-Gen testing effort. The objectives of this HLA wrapper are

1. Allowing non-simulation engineers to develop their models and simulations without an in-depth knowledge of HLA.
2. Enabling rapid prototyping of HLA compliance simulation.
3. Providing software engineer interface modules to simplify the HLA set up processes and minimize redundant processes to modify the federate during the software development cycle.
4. Supporting the ADEVS simulation engine [11].

A simplified interface is necessary for rapid prototyping and reducing the development time during software development cycle. For example, the FOMs/SOMS in Object Model Template (OMT) might change numerous times during the software development cycle before the final release of the software. If the changes are minor, the software engineer may simply change the attributes or parameters in the HLA class. If the changes are major, it may require rewriting the HLA classes, and probably will require structural changes in the software design. This interface will reduce the burdens on the software engineers, and will simplify the HLA development to include non-HLA simulation engineers.

4.1. HLA Compliance Software Development

Federation and federates are the basic terminologies used in describing HLA simulation, and the OMT provides the necessary information to build the simulation. The Federation Object Models (FOMs)/Simulation Object Models (SOMs) Lexicon in OMT consists of definitions of object classes, interaction classes, attributes, and parameters. The HLA FOMs and SOMs are stored and transferred via Federation Execution Data (FED) files using OMT Data Interchange Format (DIF). The definitions are defined in table format, and the following are the necessary tables to implement a federate:

1. Object class structure table
2. Object interaction table
3. Attributes table
4. Parameters table

In HLA simulation development, the HLA classes are represented by C++ classes[1] and the instances of these C++ classes are HLA objects. The HLA classes can be either object classes or interaction classes as shown in Figure 16. Object classes have attributes and they persist. Interaction classes have parameters, but they are sent and forgotten. The following procedures are the requirements to implement an object class:

1. Publish the object class
2. Indicate which attributes are to be published and updated
3. Subscribe object classes and their instances from other federates
4. Encode and update the attribute values
5. Reflect and decode other federates’ attribute values

[1] HLA software are developed using Object Oriented languages, such as C++ and Java. In this discussion, we assume the federate is developed using C++.

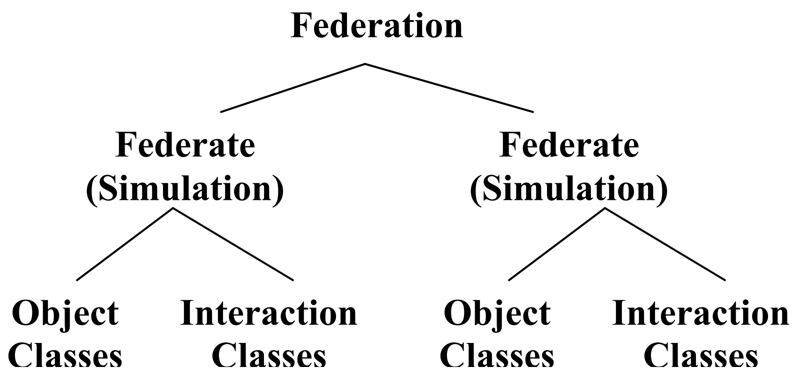


Figure 16: HLA Simulation program structure

The implementations of interaction classes are very similar to the object classes, except interaction cannot specify which parameters to publish. Interaction publishes either all parameters or nothing. Instead of using the Update and Reflect methods, interaction invokes the Send and Receive methods to update the parameters.

4.2 DEVS Simulation Support

The generic HLA wrapper is developed to support the ADEVS simulation engine based on the model-oriented approach in DEVS/HLA [12] distributed simulation and the Model/Simulator/View/Controller design pattern [13]. The HLA view object and the DEVS Simulator are communicated through the activity coordinator through a user defined class object as shown in Figure 17.

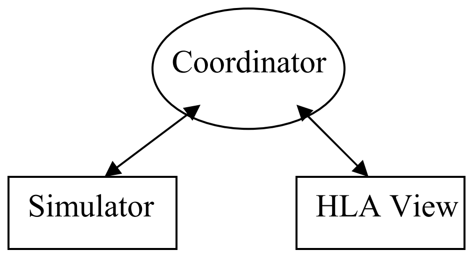


Figure 17: HLA and Simulator Communication

HLA provides two types of communications: attribute updating and interaction. Interaction updates all the parameters, and attribute updating can choose which attribute(s) to update. When the external transition function of DEVS processes input from the HLA view object, the input must contain all attributes or parameters. Because of this constraint, the Generic HLA wrapper preserves most of the basic functions in HLA management, except the behaviors of Attribute Publish/Subscribe methods in the declaration management and Attribute Update/Reflect methods in the object management. The Publish/Subscribe method is required when publishing or subscribing all attributes, and the Update/Reflect method is required when updating or reflecting all attributes. For example, when the controller receives data from HLA and injects the updates to the simulator, the updates must contain all the attributes in the object class. Although the attribute being updated behaves similarly to the interaction, attribute updating is still a persistent event while interaction is not.

4.3 Architecture

The Generic HLA wrapper has two components: generic data type and generic HLA interface. Generic data type provides a unified data coding scheme to the HLA attributes and parameters. The Generic HLA interface provides a simplified set of methods for federation management, declaration management, time management, and object management.

4.3.1 Generic Data Type

The UML diagram [14] in Figure 18 shows the standard data types in the Generic HLA wrapper. It provides the basic class structure to the Generic HLA interface in order to handle attributes/parameters in the HLA classes. The root class is Entity, which derives from DEVSJAVA [15]. HLA_Parameters and HLA_Attributes are subclasses of Entity, and they inherit all the methods and attributes from Entity class. Both classes are responsible for storing the attribute/parameter's information, such as handlers and data. The following are the descriptions for these classes:

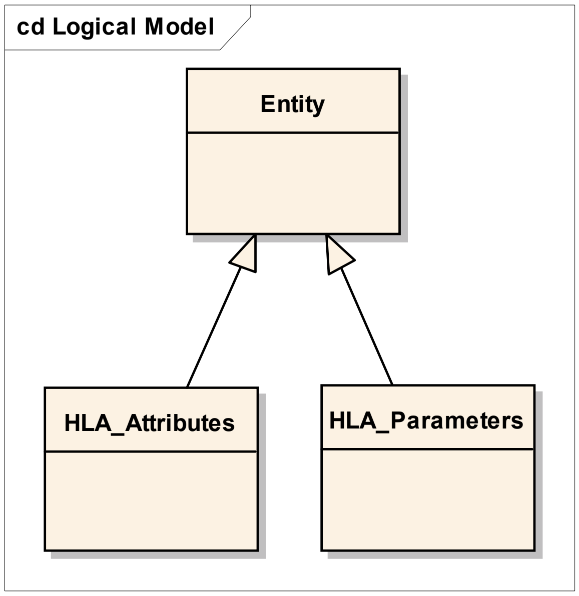


Figure 18: Generic data types

4.3.1.1 Entity

Entity is the most important element in the Generic HLA wrapper. It is a unified data type to simplify data marshalling and de-marshalling. Entity class is derived from DEVSJAVA's entity. The goal is to provide a generic data type to the software engineer, simplifying data conversion during function calls. It restricts the number of supported data types, which are integer, long, double, string, and boolean. The software engineers define the name of the variable and its data type in the initialization. By knowing the data type, the data variable can be easily converted between entity data type and specific data type. Methods, such as Get and Set, are provided to retrieve and set the name, value, and data type.

When the federate invokes the attributes' updating or interaction, the data is encoded/decoded based on its data type. If encoding occurs, the federate will convert the Entity to HLA data type and send the data to other federates. Otherwise, the federate decodes the data received from other federates and converts the HLA data type back to Entity.

4.3.1.2 HLA_Attributes/HLA_Parameters

HLA_Attributes/HLA_Parameters is a subclass of Entity. It is responsible for storing the attribute's/parameter's RTI information, such as object/interaction handler, attribute/parameter handler, data type, name, and value. The objective is encapsulating all the information about the attribute/parameter in a particular class, and the data can be easily identified by the handlers. The handlers are unique identifiers assigned by the RTI based FOM/SOM lexicon in the FED file. HLA_Attributes is identified by object handler and attribute handler, while HLA_Parameters is identified by interaction handler and parameter handler. The RTI associates the classes and attributes/parameters with handlers; the information routing is determined based on these identifiers. Standard methods, such as Get and Set, are provided to retrieve and set the handlers.

4.3.2 Generic HLA interface

The Generic HLA interface class diagram is shown Figure 19. It provides methods for HLA management and a generic data object to communicate with ADEVS components. The interface can be divided into three areas:

1. HLA_Object class is inherited from the ADEVS Object class, and it is the base class of the Generic HLA Wrapper. HLA_Object is wrapped within an ADEVS object that allows message passing between the Simulator and the HLA viewer without any data conversion.
2. HLA_ObjectClass and HLA_InteractionClass provide methods for declaration management and object management, such as Publish/Subscribe and Encoding/Decoding.
3. HLA_Federate class provides federation management and initiation of object or interaction class. The ParameterSetup method allows non-simulation engineers to develop HLA simulation without any additional development by declaring the attributes/parameters and their data types for object/interaction classes in this method.

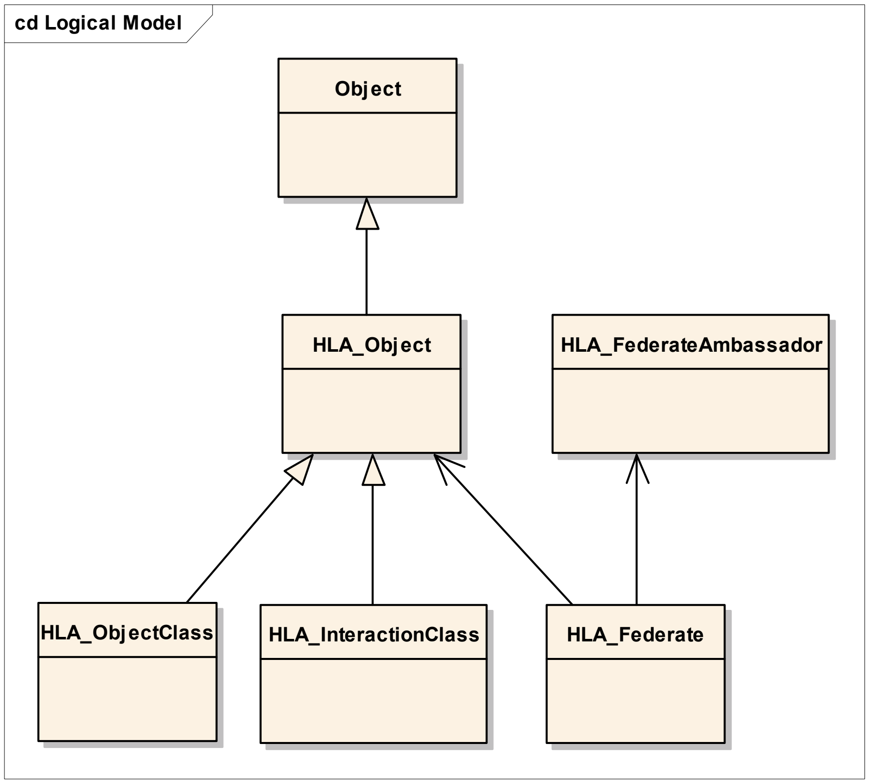


Figure 19: Generic HLA interface

4.3.2.1 HLA_Object

HLA_Object is the root class of Generic HLA interface, and it is a subclass of ADEVS object class. The ADEVS Object class is used as the basic I/O type, and allows other data types to be wrapped in it without data conversion. HLA_Object is responsible for storing the HLA_Parameters and HLA_Attributes class objects. It has two methods and two virtual methods: Print, Clone, addParameter, and addAttribute. Print() method prints all the entries in the linked lists. Clone() returns a new object of the HLA_Object. The "add" method inserts the attributes/parameters to the linked lists. There are four linked lists: recvParameterList, sendParamenterList, recvAttributeList, and sendAttributeList. They contain the attributes/parameters received from RTI. Each node of the linked list stores an Entity, and they are distinguished by class handler and object/parameter handler as shown in Figure 20. The primary objective of the HLA_Object is to simplify message passing between HLA components and ADEVS simulators without any data conversion. For example, when the federate receives a message from RTI's Interaction or Reflect callback routine, the parameters are decoded and contained in HLA_Object and passed to the simulator. When the Simulator generates and sends an event to the HLA view object, the parameters are encoded to the HLA format and the message is sent using SendInteraction.

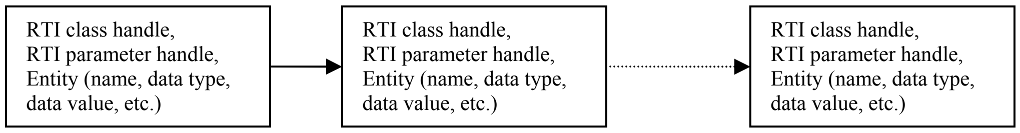

RTI class handle,
RTI parameter handle,
Entity (name, data type, data value, etc.)
RTI class handle,
RTI parameter handle,
Entity (name, data type, data value, etc.)
RTI class handle,
RTI parameter handle,
Entity (name, data type, data value, etc.)

Figure 20: Generic HLA Data List

4.3.2.2 HLA_ObjectClass

HLA_ObjectClass is a subclass of HLA_Object. It is responsible for providing methods to HLA object class for the declaration and object management. The following are the descriptions of the methods:

1. PublishandSubcribe – invokes the RTIambassador's Publish and Subscribe methods registering publication and subscription interests in object class.
2. addAttribute – adds a new HLA_Attributes class objects to the sendAttributeList in HLA_Object.

3. Update – encodes all the attributes in the sendAttributeList list, and invokes the RTIambassador's *UpdateAttributeValues* method updating all the attributes to other federates.
4. Reflect – when another federate updates its attributes, the RTI invokes the *ReflectAttributeValues* callback method to receive the attributes, then decodes the attributes to receiveAttributeList list.
5. Decode – this method decodes the attributes, and converts the data to HLA_Attributes type.

4.3.2.3 HLA_InteractionClass

HLA_InteractionClass is a subclass of HLA_Object. It is responsible for providing methods to the HLA interaction class for declaration and object management. The following is the description of the methods:

1. PublishandSubcribe – invokes the RTIambassador's Publish and Subscribe methods registering publication and subscription interests in interaction class.
2. Send – encodes all the parameters in the sendParameterList list, and invokes the RTIambassador's *SendInteraction* method, sending all the parameters to other federates.
3. Receive – when another federate updates its parameters, the RTI invokes the *ReceiveInteraction* callback method to receive the parameters and decode the parameters to receiveParameterList list.

4. Decode – this method decodes the parameters, and converts the data to HLA_Parameters type.

4.3.2.4 HLA_Federate

HLA_Federate class is responsible for federation management, time management, and the initiation of declaration management. The constructor of the HLA_Federate class initiates the declaration management, HLA_Object class object, and object/interaction classes. The following are the method description:

1. CreateandJoinFederation – allows a federate to create and join the federation. If the federation already exists, it will simply join the federation.
2. ResignandDestroyFederation – it allows a federate to resign from the federation. If this federate is the last one, it will simply destroy the federation.
3. tick – this is a time management function which yields time to RTI.
4. receiveInteraction – a callback routine for interaction classes to receive messages from other federates.
5. reflectAttributeValues – a callback routine for object classes to receive message from other federates.
6. ParametersSetup - the most important part of the HLA interface. It is responsible for the object/interaction classes and attributes/parameters declaration, in which the federate is interested to publish and subscribe.

4.4 HLA Simulation Setup

The constructor of HLA_Federate class contains all necessary components to run the HLA wrapper. In every HLA simulation, the federate must be interested to publish and subscribe certain classes described in the FED file, and the software engineer will define these class structures in the ParameterSetup method. The following steps setup the constructor:

1. Create and join federation using CreateAndJoinFederation method.
2. Set up the attributes/parameters and create new instances of the HLA_ObjectClass and HLA_InteractionClass classes in ParameterSetup method as shown in Figure 21.
3. Create an instance of HLA_Object class
4. Call PublishAndSubscribe methods in HLA_ObjectClass and/or HLA_InteractionClass classes.

```
void HLA_Federate::ParametersSetup()
{
		const int num = 6;
		parameters param[num];

		param[0].name = "ID";
		param[0].type = "LONG";
		param[1].name = "xpos";
		param[1].type = "DOUBLE";
		param[2].name = "ypos";
		param[2].type = "DOUBLE";
		param[3].name = "zpos";
		param[3].type = "DOUBLE";
		param[4].name = "xvel";
		param[4].type = "DOUBLE";
		param[5].name = "yvel";
		param[5].type = "DOUBLE";
		param[6].name = "zvel";
		param[6].type = "DOUBLE";
		char* className1 = "AirTrack";
		HLA_Interaction *iClass = new HLA_Interaction(className1, param, num, &rtiAmb);
}
```

Figure 21: ParameterSetup method

In order to incorporate the Generic HLA wrapper into a simulation, the software engineer is required to implement the interested object/interaction class structures in the ParameterSetup method. The simulation will create a new instance of the HLA_Federate class by calling the constructor, and publishes and subscribes the objects/interactions defined in the ParameterSetup. The HLA_ObjectClass and HLA_InteractionClass have methods to send and receive messages from other federates. The software engineer can specify when to use these methods as follows:

1. When the simulator generates an event, Update or SendInteraction method is called.
2. When the federate receives messages from other federates, the Reflect or ReceiveInteraction method is called.

5. 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 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.

5.1 Building Blocks

5.1.1 DEVS Specification

The DEVS formalism was introduced by Bernard Zeigler [4] 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 behavior in the physical 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 22.

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

where

X is the set of input values

S is the set of state

Y is the set of output values

δ_{int}: S -> S is the internal transition function

δ_{ext}: Q x X -> S is the external transition function, where

Q = {(s,e)|s εS, 0 ≦ e ≦ ta(s)} is the total state set,

and e is the time elapsed since last transition

λ: S->Y is the output function

ta: S->$R_{0,\infty}^{+}$ is the set of positive reals with 0 and infinity

Figure 22: 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 23.

$\boldsymbol{M = <X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta>}$

where

X is the set of input values

S is the set of state

Y is the set of output values

δ_{int}: S -> S is the internal transition function

δ_{ext}: Q x X^b -> S is the external transition function,

where X^b is a set of bags over elements in X, Q = {(s,e)|s εS, 0 ≦ e ≦ ta(s)} is the total state set, and e is the time elapsed since last transition

δ_{con}: S x X^b -> S is the confluent transition function,

subject to $\delta_{con}(s, \Phi) = \delta_{int}(s)$

λ: S->Y^b is the output function

Figure 23: 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 following structure:

$\boldsymbol{M = <X, Y, D, \{M_{ij}\},\{I_j\}, \{Z_{ij}\}>}$

Where

X is a set of inputs

Y is a set of outputs

D is a set of DEVS component names

For each i εD,

$\mathbf{M_i}$ is a DEVS component model

$\mathbf{I_i}$ is the set of influences for I

For each j ε I_i,

Figure 24: Coupled DEVS Specification

Two 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.

5.1.2 Experimental Frame

The Experimental Frame is a specification of the conditions under which the system is observed or experimental with [4]. It reflects the objectives of the experiment performed on a real system or simulation. 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. Another view is that a frame can interact with the system to obtain data under specified conditions. In the second view, the frame can be treated as an observer, and it has three components: generator, acceptor, and transducer, as illustrated in Figure 25. 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.

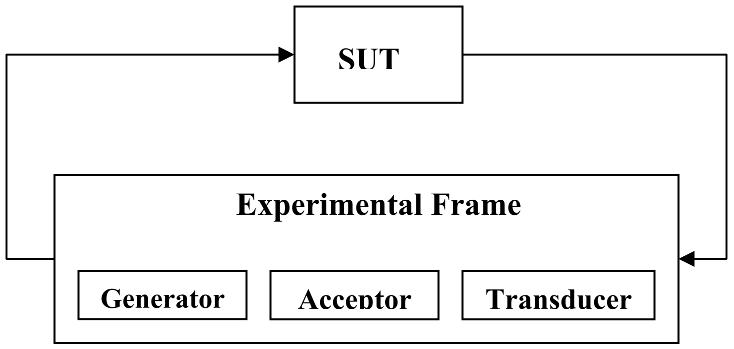


Figure 25: Experimental Frame

5.1.3 Model/Simulator/View/Controller (MSVC) Design Pattern

The Model/Simulator/View/Controller (MSVC) design pattern was proposed by Jim Nutaro [13] and provides a software framework to the developers that supports component-based software design, and incorporate the key concepts from DEVS modeling and simulation to promote a separation of model, simulator, and distributed computing. MSVC is an extension of the Model/View/Controller (MVC) design pattern commonly used in interactive programming design, and it can apply to building distributed simulation [16]. MVC is a component-based software architecture that separates the data model, data representation, and control logic into three components. These components are reusable, and the modification of these components will have minimal impact on the others. In order to extend the MVC approach to the support of complex distributed modeling and simulation application, a simulator is added to the existing MVC design. As a result, the MSVC design provides a software framework to support distributed modeling and simulation.

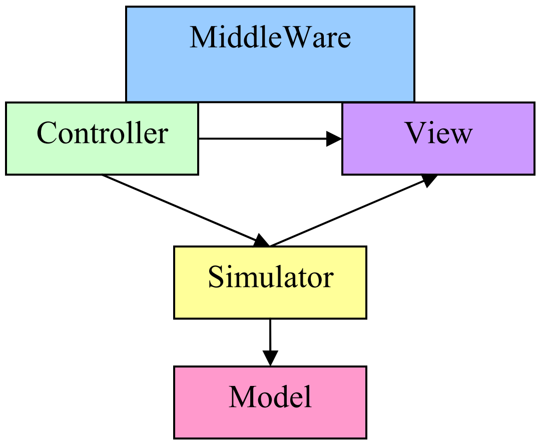


Figure 26: MSVC components and their relationships

The MSVC components and their relationships are shown in Figure 26 [14]. Model is a physical or logical representation of a proposed or real system, and it can be a set of instructions or constraints for generating I/O behavior. It is a system specification that has state transitions and output generation mechanisms to accept input and generate outputs depending on the initial state. The simulator is capable of obeying the instructions and executing the model to generate behavior. It provides an interface allowing the injection of inputs, computation of states changes, and the viewing of model outputs. The controller interacts with the simulator to allow the model behavior to be influenced by other large systems. The viewer extracts output data from the simulator, and acts on simulation output at the request of the controller. In a distributed simulation environment, the controller and the viewer can interact with the middleware as in Figure 26. The controller handles the inputs and time management scheme sent from the middleware and the viewer extracts the data outputs from the simulator and sends it via middleware. For example, the controller utilizes the time management and object management features of the HLA middleware to control the simulator execution rate and generate model inputs. The viewer is not involved in the any controller scheme, although the controller can signal the viewer to generate data outputs from the model outputs.

5.1.4 A Discrete Event System Simulator (ADEVS)

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.

5.2 Enhanced MSVC Design Pattern

Jim Nutaro 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 27 below provides a graphical representation of the enhanced MSVC paradigm [17]. 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 27, 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.

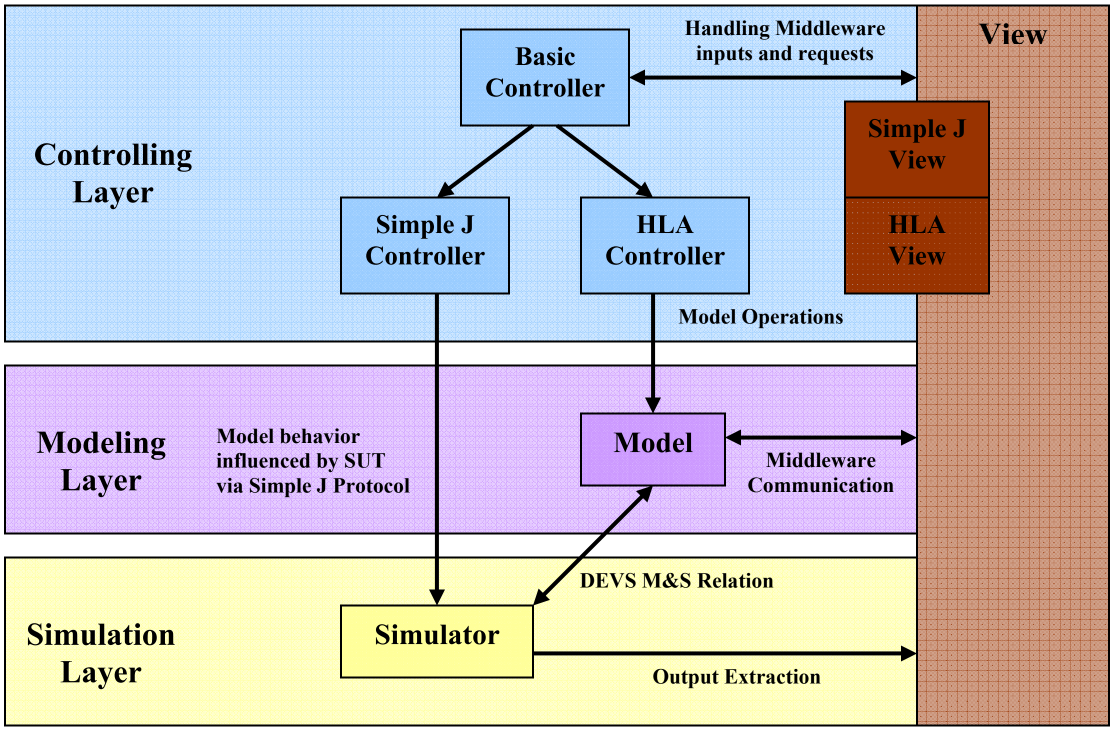


Figure 27: Enhanced MSVC paradigm with multiple controllers

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.

5.3 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.

5.3.1 Model

The model is a physical or logical representation of a system, and it represents the system specification at level 3 and level 4 of the system specification hierarchy. The most common concept of simulation model is a set of instructions or constraints for generating I/O behavior. A model is written with state transition and coupled component system specifications. In state transition specification, we can specify the initial state setting and how the state changes as the system responds to the input trajectory, and also what output trajectory is generated by the state. A model is composed of interacting components, and these components can be coupled together leading to a hierarchical structure.

The Test Driver 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 described in Section 3.

5.3.2 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.

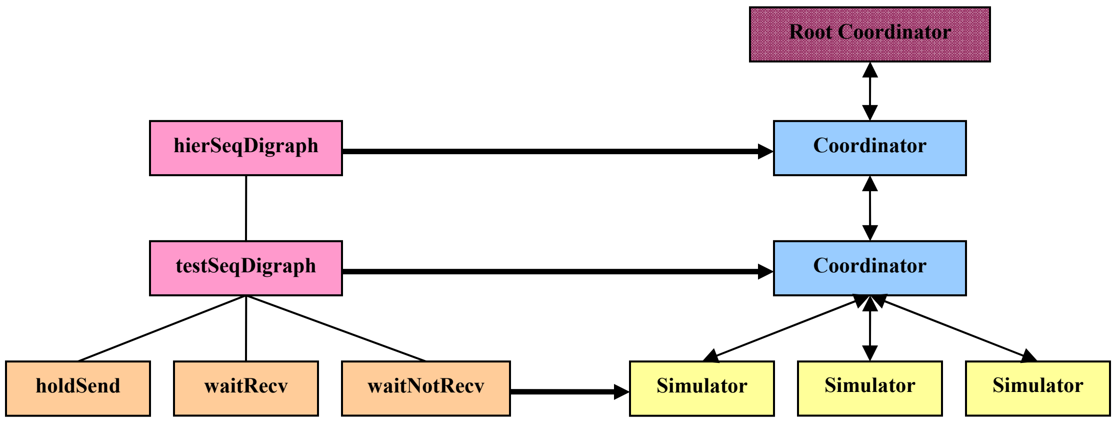


Figure 28: Mapping Test Driver model onto a hierarchical simulator

The hierarchy of the Test Driver model is shown in Figure 28 in above. The test driver simulator is developed using ADEVS simulation engine, and it consists of three atomic models and two coupled models. 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 Simple 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 Simple 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 Simple J message output, $\lambda(s)$, and changes to state $\delta_{int}(s)$.

5.3.2.1 holdSend

holdSend is an atomic model that generates 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 Simple J message based on message type and message, and a start signal. Figure 29 shows the state transition of the holdSend model.

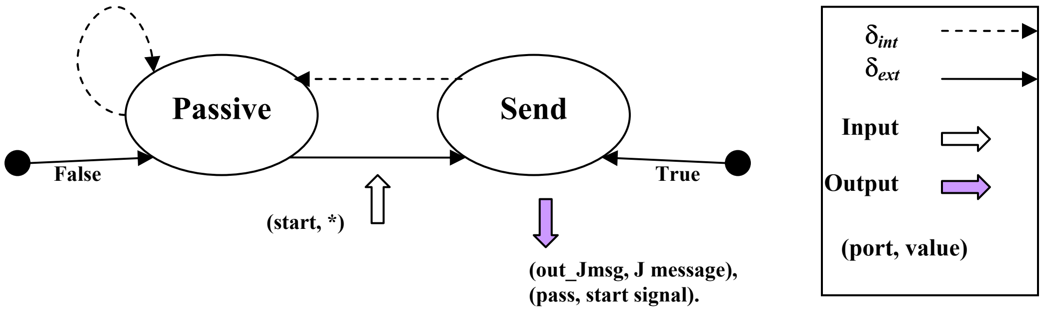


Figure 29: holdSend State diagram

5.3.2.2 waitReceive

waitReceive is an atomic model that waits for an incoming J message and determines the experimental condition based on the received message and the pre-defined message. The constructor of waitReceive consists of three fields: model name, message, and wait time. The model begins at the "Passive" state. When it receives a start event, the external transition function dictates a new state, "Wait," with a new wait time. If the model receives a J message event before resting time expires, it compares the received J message with the given J message. If they are identical, it will print out a message indicated that the test is passed, and the state will change to "Success." λ(s) will generate a start signal output, and the internal transition function will change to the "Passive" state. If the messages are different, it will print out a failed message, and the state will change to "Passive." If the J message doesn't arrive before the resting time expires, the test will be failed, and the state will change to "Passive." Figure 30 shows the state transition of the waitReceive model.

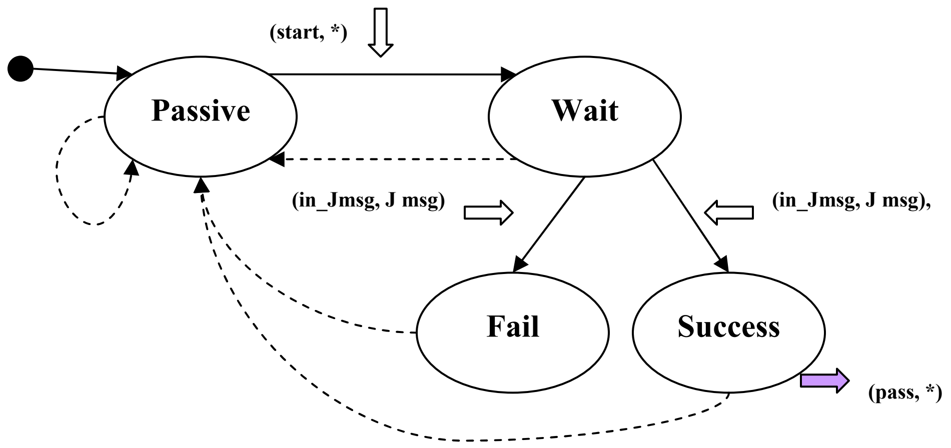


Figure 30: waitReceive State diagram

5.3.2.3 waitNotReceive

waitNotReceive is an atomic model that idles the model for a given time if it is not expecting to receive any incoming message. The constructor of this model has two fields: model name and wait time. The model begins at the "Passive" state. When an external event occurs, the external transition function dictates a new "Wait" state. When the resting time expires, it generates a start output signal, and the internal transition function will change to the "Passive" state. Figure 31 shows the state transition of waitNotReceive model.

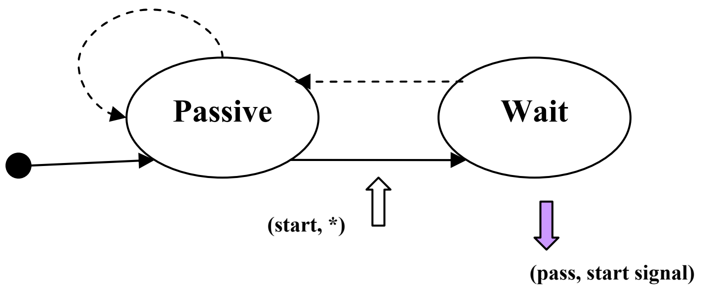


Figure 31: waitNotReceive State diagram

5.3.2.4 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.

5.3.2.5 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.

5.3.3 View

The viewer extracts the output from the simulator and provides inputs to the controller. In the Test Driver, we find 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.

5.3.4 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.

5.4 Activity Coordinator

An activity coordinator is implemented to manage the communications between MSVC components using the Event Notification and Mediator pattern [18, 19]. An activity coordinator object acts as the mediator that links the coordinator to the simulator, views, and controller. The UML diagram shown in Figure 32 describes the primary objects and their relationships.

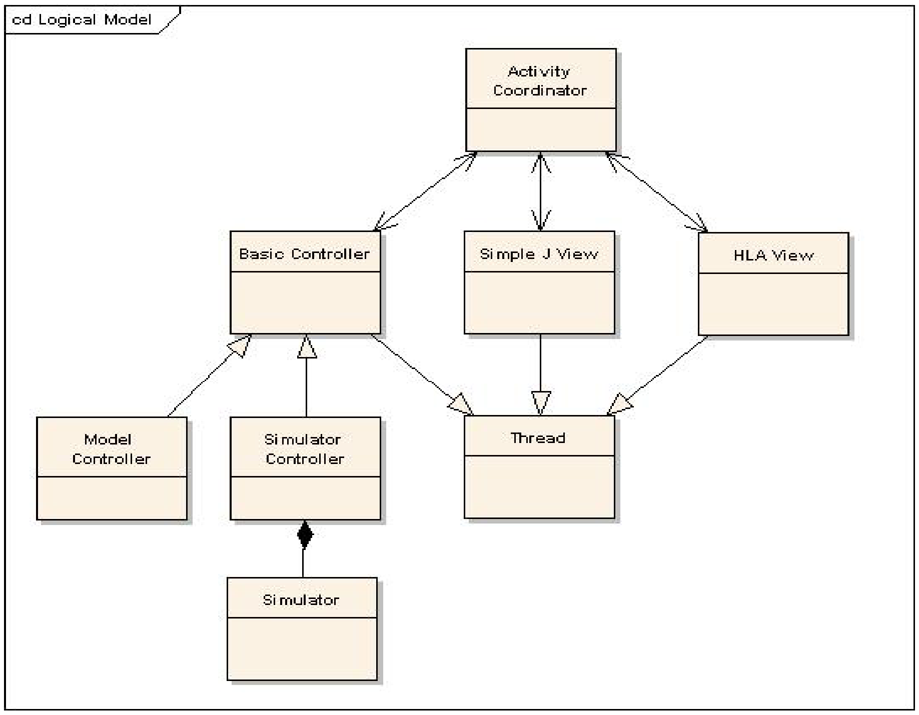


Figure 32: Test Driver's primary objects and their relationships

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.

6. Experiment and Results

In this chapter, several experiments are conducted using the automated test generation processes shown in Figure 33. The scenarios used in these experiments are auto correlation, decorrelation, report and responsibility shift, and drop track. Each of these scenarios were performed against the Integrated Architecture Behavior Model (IABM) developed by the Joint SIAP System Engineering Organization (JSSEO). The results of the scenarios were verified by the ATC-Gen Test Driver and validated using JITC's TAMD Interoperability Assessment Capability (TIAC) tool. Due to the classification levels 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 experiments to be conducted.

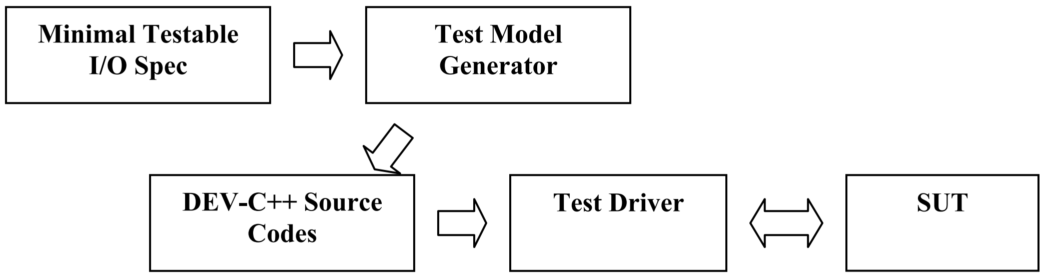


Figure 33: Automated Testing

6.1 Scenarios

6.1.1 Auto Correlation

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 34 illustrates the auto correlation process in the sequential diagram.

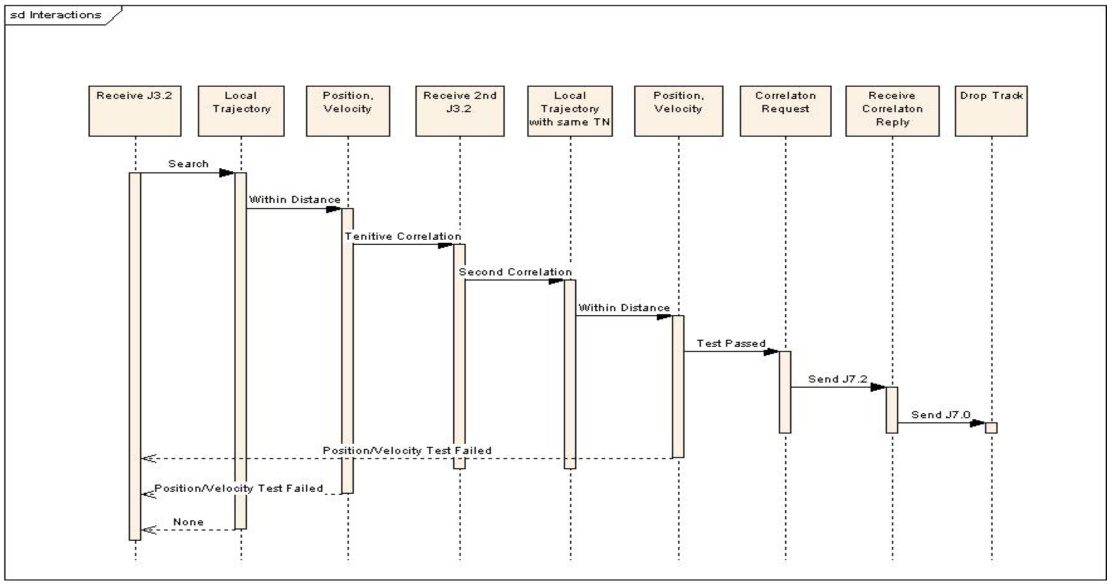


Figure 34: Auto Correlation Sequential Diagram

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 35 illustrates the minimal testable pairs for SUT and Test Driver.

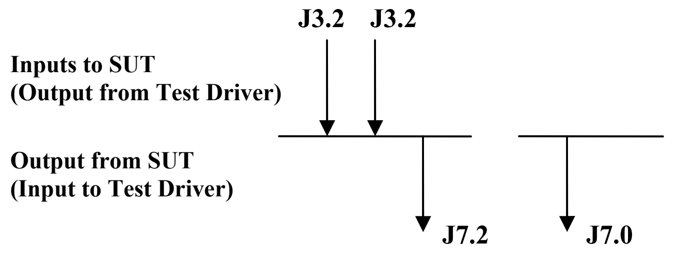


Figure 35: Minimal Testable I/O pairs for Auto Correlation

6.1.2 Report and Responsibility (R^2) Shift

As MIL-STD 6016C stated, the Joint Tactical Information Distribution System (JTIDS) unit (JU) reports the tracks on the interface, because it has the best positional data available. A JU assumes R^2 on a common local track if its track quality (TQ) at the time of transmission exceeds the received TQ by 2 or higher. Figure 36 illustrates the R^2 process in the UML sequential diagram.

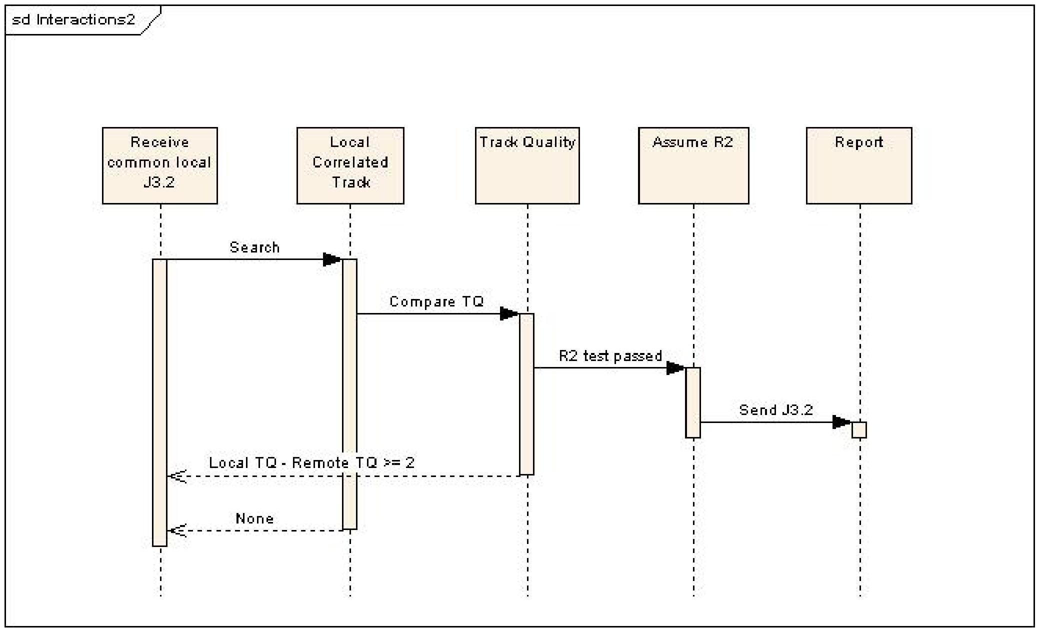


Figure 36: R^2 Shift Sequential Diagram

In order to create the R^2 minimal testable pairs, a common local track is created by the auto correlation process described in Section 6.1.1. After the common local track is created, the SUT receives a J3.2 message from the remote system. The SUT compares its own track quality to the received track quality. If the SUT TQ is higher than the received TQ by 2, the SUT will send a J3.2 message back to the remote system and assumes reporting responsibility; otherwise, the remote system retains the reporting responsibility and continues to transmit J3.2 messages. Figure 37 illustrates the minimal testable pairs for the SUT and the Test Driver.

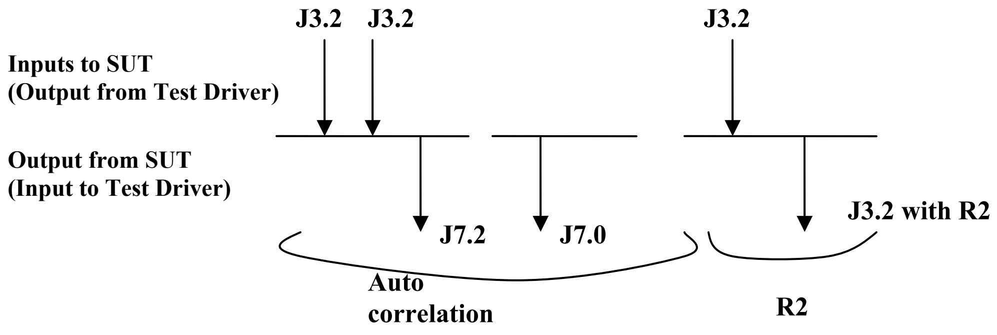


Figure 37: Minimal Testable I/O pairs for R^2 Shift

6.1.3 Decorrelation

As MIL-STD 6016C stated, the common local tracks shall be decorrelated if two consecutive remote track reports are received and the remote tracks falls outside a distance of 1.5 times the maximum correlation distance. Figure 38 illustrates the decorrelation process in the UML sequential diagram.

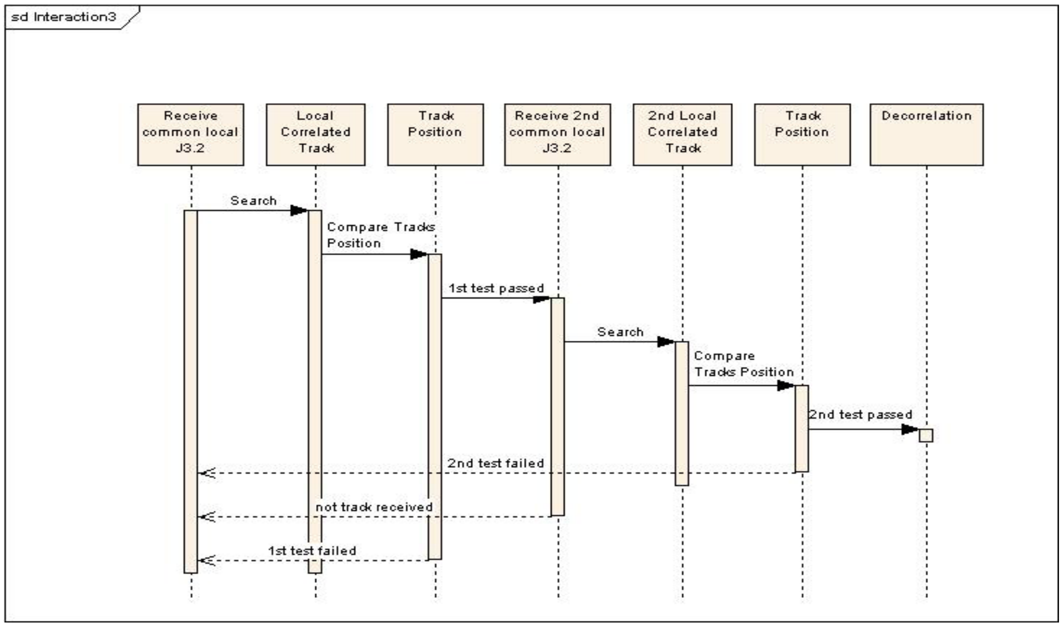


Figure 38: Decorrelation Sequential Diagram

At the beginning, the common local track is assumed to be created by the auto correlation process described in Section 6.1.1. The SUT receives a remote J3.2 air track message and compares remote air track position to the SUT air track position. If their distance is 1.5 times greater than the maximum correlation window, the first decorrelation condition is met. The SUT waits for a second J3.2 air track message and compares it to the SUT air track. If the distance is also 1.5 times greater, the second condition is met. The common local track will decorrelate and a new track number will be assigned to the local track. Figure 39 illustrates the minimal testable pairs for the SUT and the Test Driver.

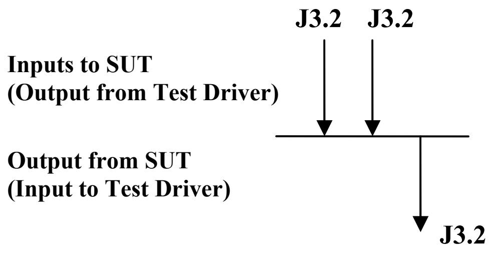


Figure 39: Minimal Testable I/O pairs for Decorrelation

6.1.4 Drop Track

In MIL-STD 6016C, a drop track is used in many different situations, such as R^2 and auto correlation. A simple scenario is chosen in this section to demonstrate the condition and the result of the drop track. As MIL-STD stated, if for any reason, the simulation status of a track is changed, the track shall be dropped using J7.0 track management, ACT = 0 (Drop track report). Figure 40 illustrates the drop track process in the UML sequential diagram.

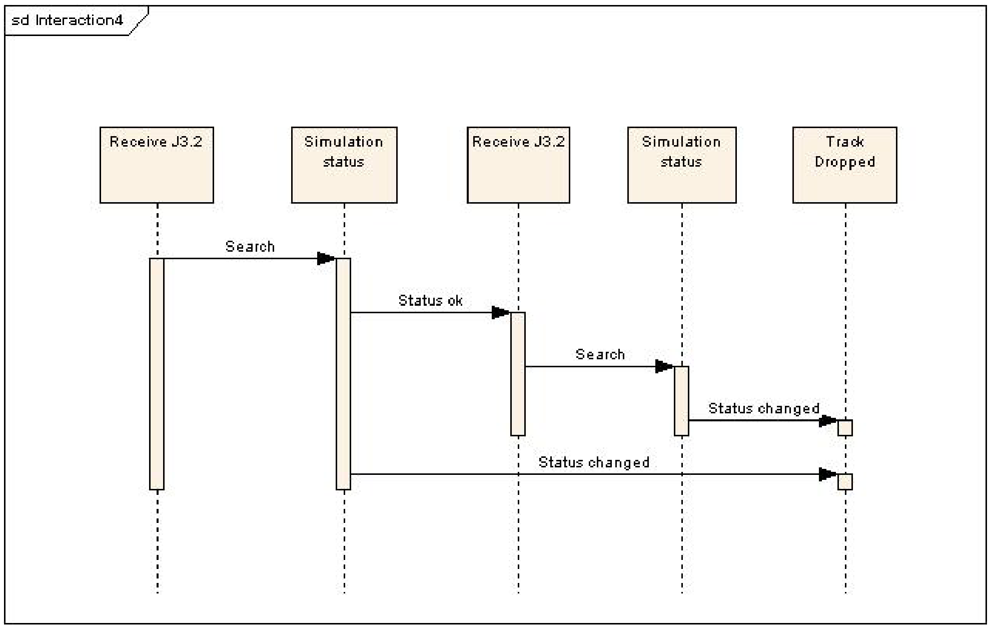


Figure 40: Drop track Sequential Diagram

When the SUT sends J3.2 air track message, it checks the status of the environment message field. If the SUT changes the status of the track for any reason, a J7.0 drop track message will be sent after the J3.2 air track message. Figure 41 illustrates the minimal testable pairs for the SUT and the Test Driver.

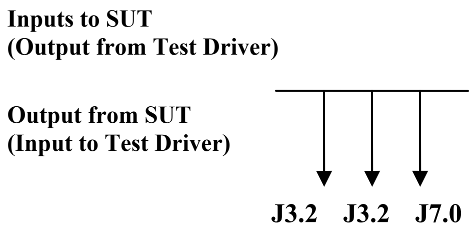


Figure 41: Minimal Testable I/O pairs for Drop Track

6.2 SUT Model Creation

Due to the fact that the military hardware system is classified by DoD, the SUT models are created to verify the correctness of the test models. The SUT models are implemented into the Test Driver and verify against the test model version via HLA middleware or Simple J protocol. This DEVS inversion concept was first introduced by Song and Kim [20].

6.2.1 SUT Model

The SUT model is generated based on the test model generation concept described in Section 3. 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 minimal testable pair concept in which an I/O pair has either a set or an empty set of inputs followed by an output. Figure 42 illustrates the transformation from minimal testable pairs to the DEVS atomic models.

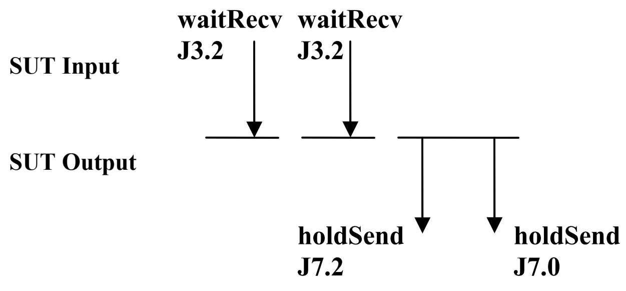


Figure 42: SUT model for Auto Correlation

6.2.2 Processing time

During the verification process, both Test Drivers are assumed started at the same time by a batch file. Each DEVS atomic 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 Test 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 before the Test Model Test Driver. Second, a time shift is added to the SUT model's processing time. The rules of the time shift are defined as follows,

1. The first atomic model must be holdSend with processing time equaled to zero due to the behavior of the ADEVS simulation engine.
2. If the current model is holdSend and the last model is not holdSend, the time shift is -2 second.
3. If the current model is waitReceive and the last model is holdSend, the time shift is +2 second.

Figure 43 below 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 before the Test Model, we can safely assume that the SUT is always running a few hundred milliseconds faster than the Test Model.

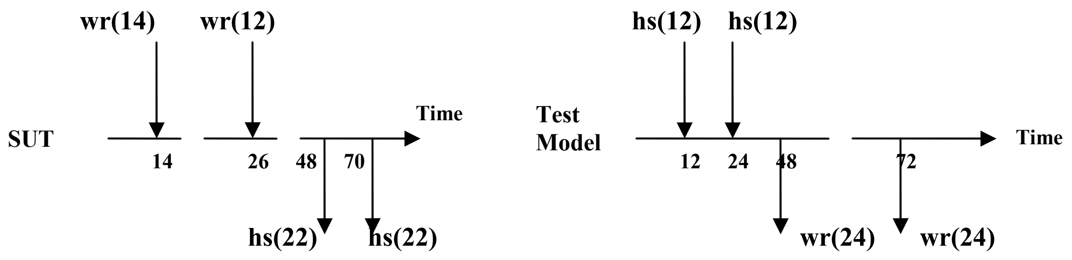


Figure 43: SUT and Test Model Timeline for Auto Correlation

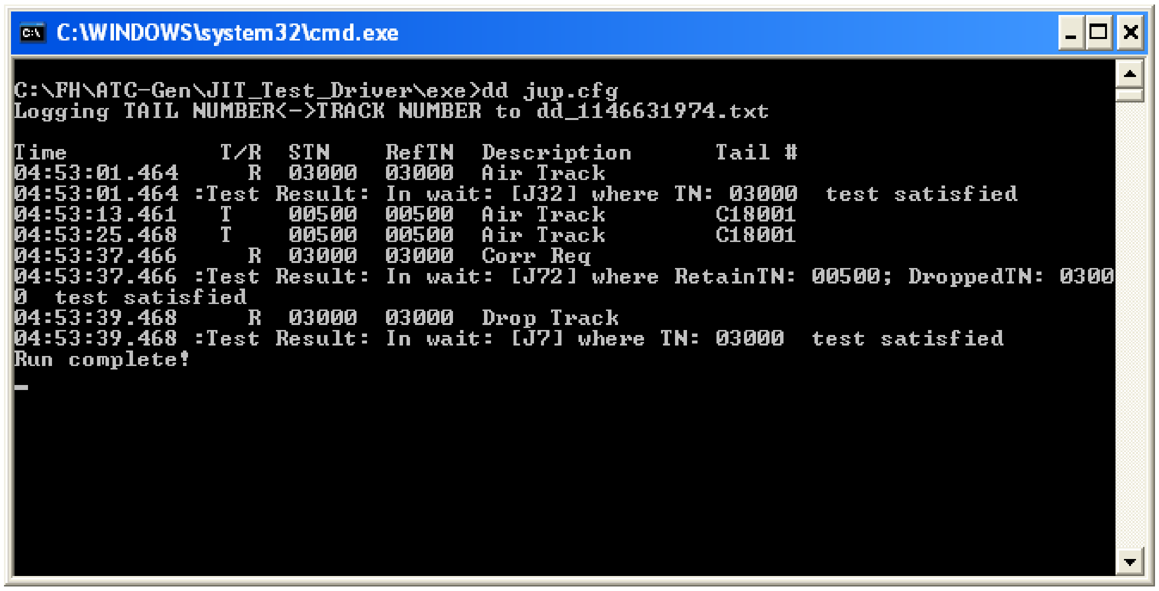
6.3 Experiment Setup & Results

The auto correlation, decorrelation, report and responsibility shift, and drop track scenarios are created to demonstrate the correctness of the models generated by the Test Model Generator. These models are implemented into the SUT and Test Model Test Drivers and communicate via Simple J protocol as illustrated in Figure 44. The transmissions and the receipt of the Simple J messages of each scenario are captured by a Simple J network packet sniffer called TIAC Simple J parser. The TIAC parser 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 44: Test Drivers Setup Diagram

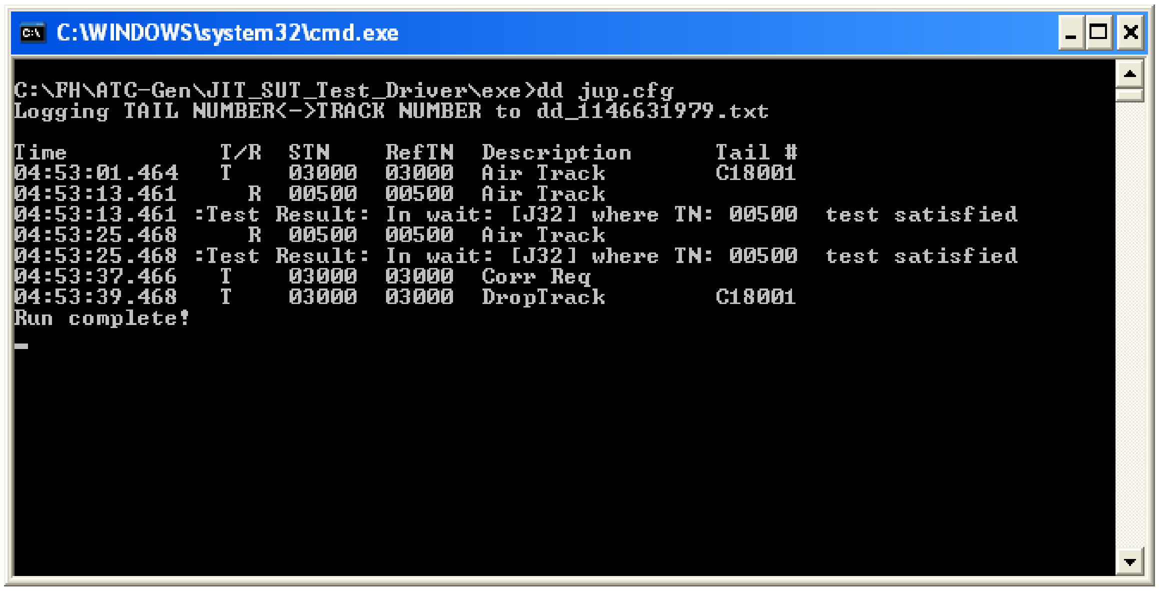
6.3.1 Successful Auto Correlation

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 45 illustrates the outputs from the Test Model TD, and Figure 46 illustrates the results for the SUT TD.



```
C:\FH\ATC-Gen\JIT_Test_Driver\exe>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1146631974.txt

Time            T/R  STN    RefTN  Description      Tail #
04:53:01.464      R  03000  03000  Air Track
04:53:01.464 :Test Result: In wait: [J32] where TN: 03000  test satisfied
04:53:13.461    T    00500  00500  Air Track        C18001
04:53:25.468    T    00500  00500  Air Track        C18001
04:53:37.466      R  03000  03000  Corr Req
04:53:37.466 :Test Result: In wait: [J72] where RetainTN: 00500; DroppedTN: 0300
0  test satisfied
04:53:39.468      R  03000  03000  Drop Track
04:53:39.468 :Test Result: In wait: [J7] where TN: 03000  test satisfied
Run complete!
_
```

Figure 45: Test Model Test Driver successful Auto Correlation scenario



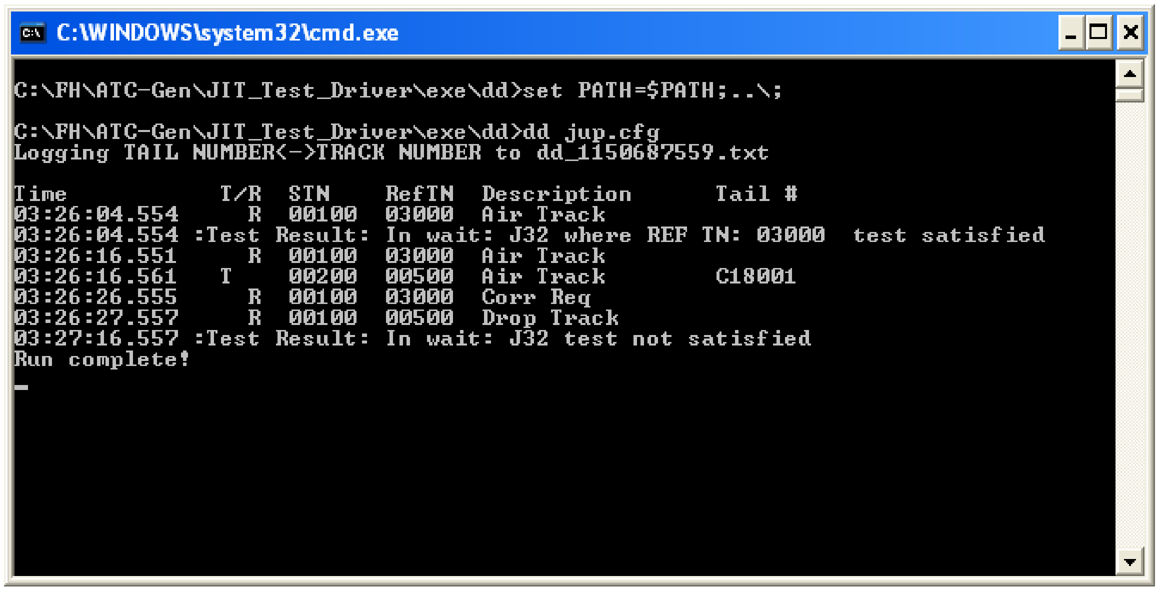
C:\WINDOWS\system32\cmd.exe

```
C:\FH\ATC-Gen\JIT_SUT_Test_Driver\exe>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1146631979.txt

Time             T/R  STN    RefTN  Description      Tail #
04:53:01.464     T    03000  03000  Air Track        C18001
04:53:13.461       R  00500  00500  Air Track
04:53:13.461 :Test Result: In wait: [J32] where TN: 00500  test satisfied
04:53:25.468       R  00500  00500  Air Track
04:53:25.468 :Test Result: In wait: [J32] where TN: 00500  test satisfied
04:53:37.466     T    03000  03000  Corr Req
04:53:39.468     T    03000  03000  DropTrack        C18001
Run complete!
_
```

Figure 46: SUT Test Driver successful Auto Correlation scenario

6.3.2 Failed Auto Correlation

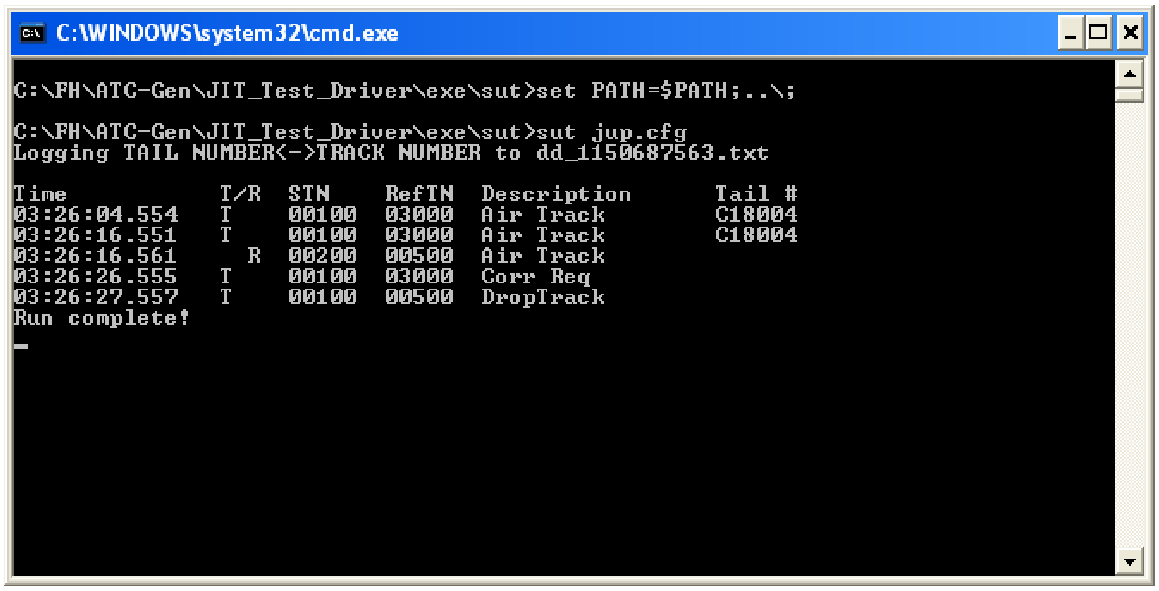
This scenario is similar to the last one except the correlation process is failed. The Test Model TD sends out the two J3.2 track messages and waits for a J7.2 correlation request message and a J7.0 Drop Track message. But, the SUT Test Driver is modified to receive only one J3.2 track message and initiate the correlation request. This causes the Test Model TD to receive a J7.2 message prematurely while it is waiting for a J3.2 message. Figure 43 shows the outputs from the Test Model TD, and Figure 44 illustrates the results from the SUT TD.



C:\WINDOWS\system32\cmd.exe

```
C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150687559.txt

Time           T/R  STN    RefTN  Description      Tail #
03:26:04.554     R  00100  03000  Air Track
03:26:04.554 :Test Result: In wait: J32 where REF TN: 03000  test satisfied
03:26:16.551     R  00100  03000  Air Track
03:26:16.561   T    00200  00500  Air Track        C18001
03:26:26.555     R  00100  03000  Corr Req
03:26:27.557     R  00100  00500  Drop Track
03:27:16.557 :Test Result: In wait: J32 test not satisfied
Run complete!
_
```

Figure 47: Test Model Test Driver failed Auto Correlation scenario



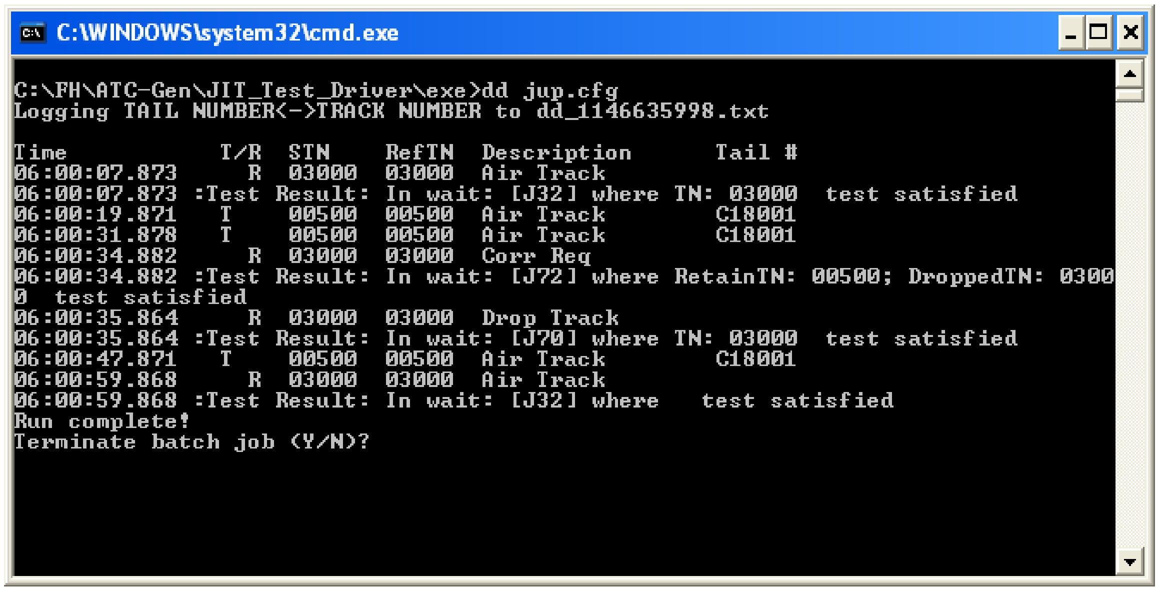
C:\WINDOWS\system32\cmd.exe

```
C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>sut jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150687563.txt

Time           T/R  STN    RefTN  Description      Tail #
03:26:04.554   T    00100  03000  Air Track        C18004
03:26:16.551   T    00100  03000  Air Track        C18004
03:26:16.561     R  00200  00500  Air Track
03:26:26.555   T    00100  03000  Corr Req
03:26:27.557   T    00100  00500  DropTrack
Run complete!
_
```

Figure 48: SUT Test Driver failed Auto Correlation scenario

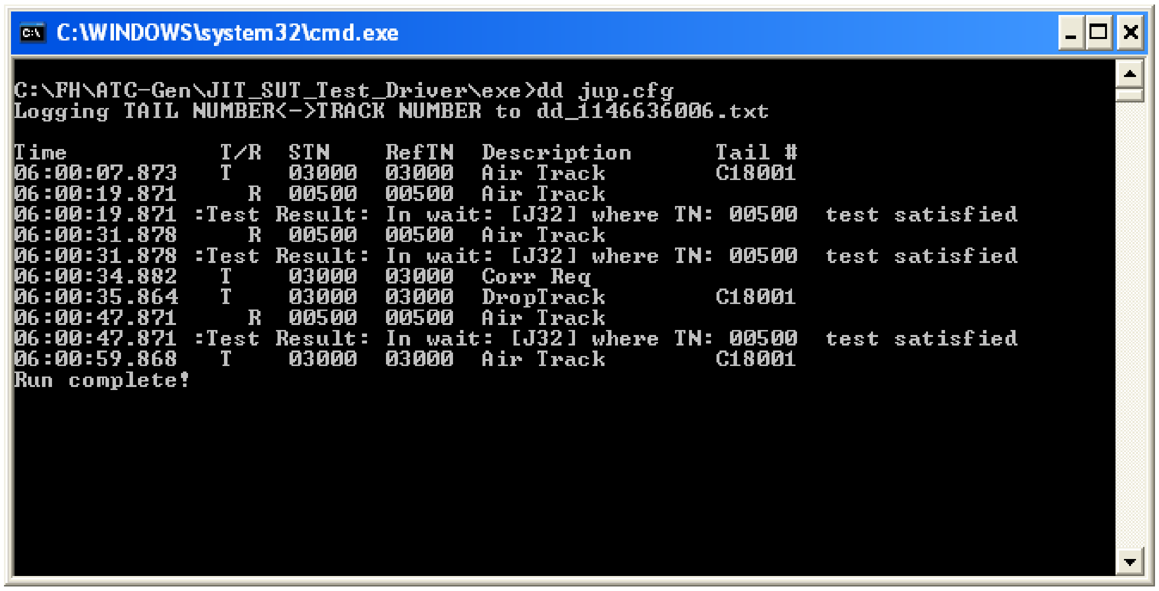
6.3.3 R^2 Shift

In this scenario, the Test Drivers are communicated via Simple J protocol. The messages setup in the correct sequence and the R^2 shift 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. After the correlation process, a common local track with track number of 00500 is created. The Test Model TD has the reporting responsibility and sends out a J3.2 track message to the SUT Test Driver. When the SUT TD receives the remote J3.2 message and compares its own TQ to the received TQ. The SUT TD has the higher TQ than the Test Model TD, and it assumes the reporting responsibility. Figure 45 illustrates the Test Model TD outputs, and Figure 46 shows the results from the SUT TD.



```
C:\FH\ATC-Gen\JIT_Test_Driver\exe>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1146635998.txt

Time            T/R  STN    RefTN  Description      Tail #
06:00:07.873      R  03000  03000  Air Track
06:00:07.873 :Test Result: In wait: [J32] where TN: 03000  test satisfied
06:00:19.871    T    00500  00500  Air Track        C18001
06:00:31.878    T    00500  00500  Air Track        C18001
06:00:34.882      R  03000  03000  Corr Req
06:00:34.882 :Test Result: In wait: [J72] where RetainTN: 00500; DroppedTN: 0300
0  test satisfied
06:00:35.864      R  03000  03000  Drop Track
06:00:35.864 :Test Result: In wait: [J70] where TN: 03000  test satisfied
06:00:47.871    T    00500  00500  Air Track        C18001
06:00:59.868      R  03000  03000  Air Track
06:00:59.868 :Test Result: In wait: [J32] where   test satisfied
Run complete!
Terminate batch job (Y/N)?
```

Figure 49: Test Model Test Driver successful R^2 Shift scenario



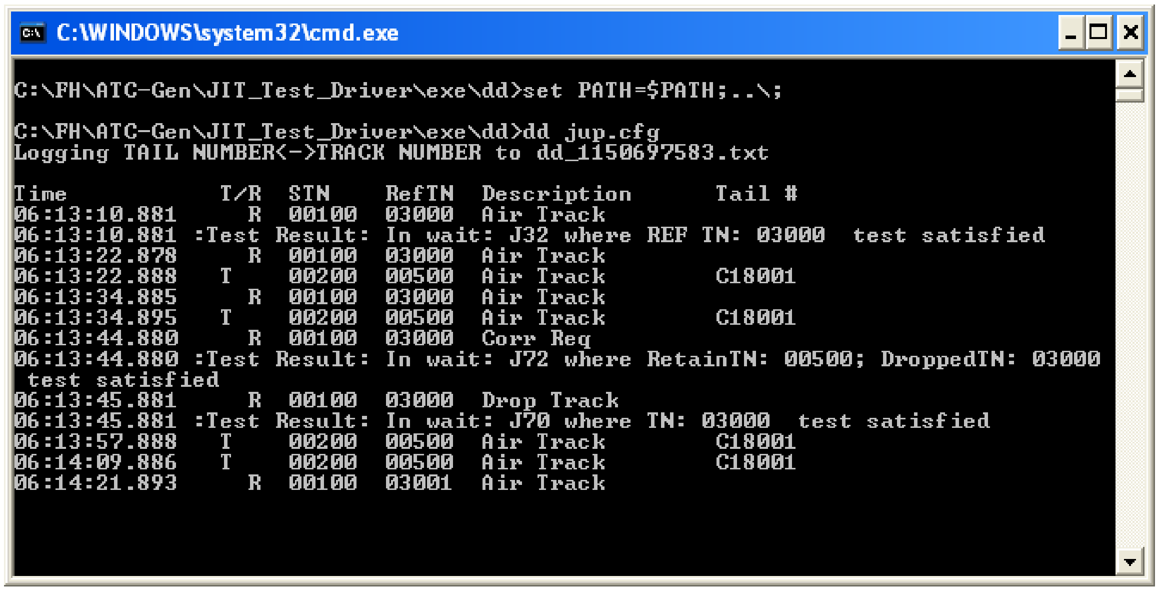
```
C:\FH\ATC-Gen\JIT_SUT_Test_Driver\exe>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1146636006.txt

Time           T/R   STN    RefTN  Description      Tail #
06:00:07.873   T     03000  03000  Air Track        C18001
06:00:19.871     R   00500  00500  Air Track
06:00:19.871 :Test Result: In wait: [J32] where TN: 00500  test satisfied
06:00:31.878     R   00500  00500  Air Track
06:00:31.878 :Test Result: In wait: [J32] where TN: 00500  test satisfied
06:00:34.882   T     03000  03000  Corr Req
06:00:35.864   T     03000  03000  DropTrack        C18001
06:00:47.871     R   00500  00500  Air Track
06:00:47.871 :Test Result: In wait: [J32] where TN: 00500  test satisfied
06:00:59.868   T     03000  03000  Air Track        C18001
Run complete!
```

Figure 50: SUT Test Driver successful R^2 Shift scenario

6.3.4 Decorrelation

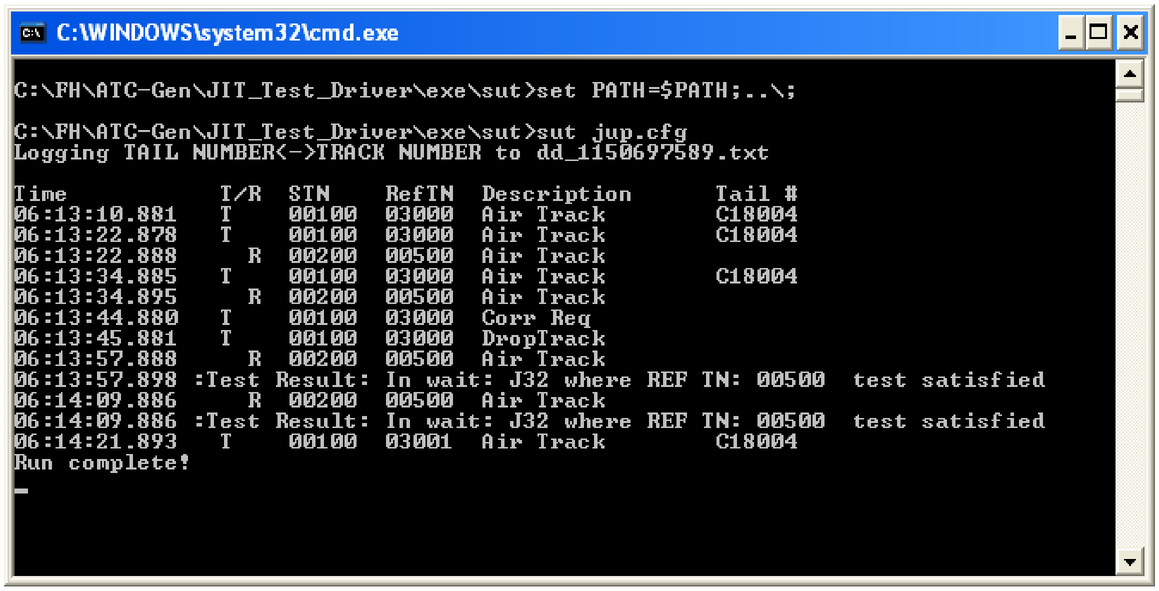
In this scenario, the Test Drivers are communicated via Simple J protocol. The messages setup in the correct sequence and decorrelation is induced. The SUT models and Test Models are generated by the Test Model Generator, and implemented into the Test Driver. The SUT TD has the track number of 03000, and the Test Model TD has the track number of 00500. After the auto correlation process, a common local track with track number of 00500 is created. The Test Model TD has the reporting responsibility and sends out a J3.2 track message to the SUT Test Driver. When the SUT TD receives the remote J3.2 message and compares its own track's position. The remote track will start deviating from the local track. At the end, the distance of the remote track will be 1.5 times of the local track. The SUT decorrelates and assigns a new track number of

3001 to the track. Figure 51 shows the outputs of the remote system, and Figure 52 illustrates the results of the local system.



```
C:\WINDOWS\system32\cmd.exe

C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150697583.txt

Time          T/R  STN    RefTN  Description      Tail #
06:13:10.881     R  00100  03000  Air Track
06:13:10.881 :Test Result: In wait: J32 where REF TN: 03000  test satisfied
06:13:22.878     R  00100  03000  Air Track
06:13:22.888   T    00200  00500  Air Track        C18001
06:13:34.885     R  00100  03000  Air Track
06:13:34.895   T    00200  00500  Air Track        C18001
06:13:44.880     R  00100  03000  Corr Req
06:13:44.880 :Test Result: In wait: J72 where RetainTN: 00500; DroppedTN: 03000
 test satisfied
06:13:45.881     R  00100  03000  Drop Track
06:13:45.881 :Test Result: In wait: J70 where TN: 03000  test satisfied
06:13:57.888   T    00200  00500  Air Track        C18001
06:14:09.886   T    00200  00500  Air Track        C18001
06:14:21.893     R  00100  03001  Air Track
```

Figure 51: Test Model Test Driver Decorrelation scenario

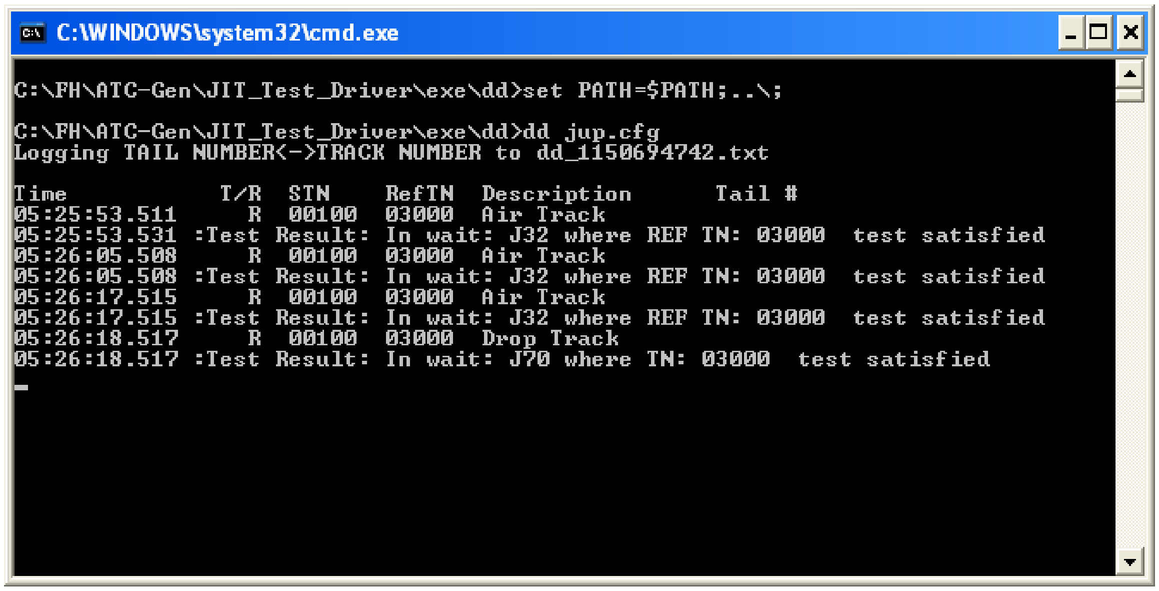


```
C:\WINDOWS\system32\cmd.exe

C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>sut jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150697589.txt

Time          T/R  STN    RefTN  Description      Tail #
06:13:10.881   T    00100  03000  Air Track        C18004
06:13:22.878   T    00100  03000  Air Track        C18004
06:13:22.888     R  00200  00500  Air Track
06:13:34.885   T    00100  03000  Air Track        C18004
06:13:34.895     R  00200  00500  Air Track
06:13:44.880   T    00100  03000  Corr Req
06:13:45.881   T    00100  03000  DropTrack
06:13:57.888     R  00200  00500  Air Track
06:13:57.898 :Test Result: In wait: J32 where REF TN: 00500  test satisfied
06:14:09.886     R  00200  00500  Air Track
06:14:09.886 :Test Result: In wait: J32 where REF TN: 00500  test satisfied
06:14:21.893   T    00100  03001  Air Track        C18004
Run complete!
_
```

Figure 52: SUT Test Driver Decorrelation scenario

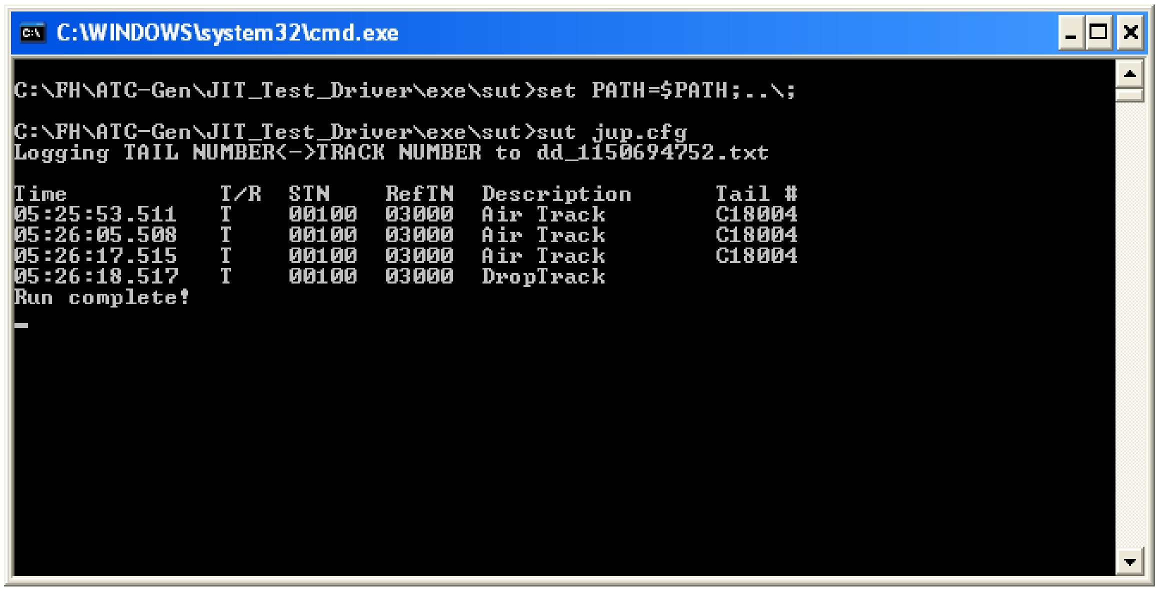
6.3.5 Drop Track

In this scenario, the Test Drivers are communicated via Simple J protocol. The messages setup in the correct sequence and drop track 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. This scenario assumes the SUT TD receives track information from the sensor simulation. When the SUT TD receives the first position of a track, the environment status is set to live track and a J3.2 message is sent. When it receives the second position of the same track, the environment status is set to simulated track and a J3.2 message is sent. Because the environment status is changed, the SUT TD will drop the track and send out a J7.0 drop track message. The Test Model TD will receive these messages and monitor the correctness of the scenario behavior. Figure 53 shows the outputs of the remote system, and Figure 54 illustrates the results of the local system.



```
C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\dd>dd jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150694742.txt

Time           T/R  STN    RefTN  Description      Tail #
05:25:53.511     R  00100  03000  Air Track
05:25:53.531 :Test Result: In wait: J32 where REF TN: 03000  test satisfied
05:26:05.508     R  00100  03000  Air Track
05:26:05.508 :Test Result: In wait: J32 where REF TN: 03000  test satisfied
05:26:17.515     R  00100  03000  Air Track
05:26:17.515 :Test Result: In wait: J32 where REF TN: 03000  test satisfied
05:26:18.517     R  00100  03000  Drop Track
05:26:18.517 :Test Result: In wait: J70 where TN: 03000  test satisfied
_
```

Figure 53: Test Model Test Driver Drop Track scenario



```
C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>set PATH=$PATH;..\;

C:\FH\ATC-Gen\JIT_Test_Driver\exe\sut>sut jup.cfg
Logging TAIL NUMBER<->TRACK NUMBER to dd_1150694752.txt

Time            T/R   STN    RefTN  Description     Tail #
05:25:53.511    T     00100  03000  Air Track       C18004
05:26:05.508    T     00100  03000  Air Track       C18004
05:26:17.515    T     00100  03000  Air Track       C18004
05:26:18.517    T     00100  03000  DropTrack
Run complete!
_
```

Figure 54: SUT Test Driver Drop Track scenario

7. Conclusion and Future Works

A new automated testing approach has been successfully developed in this thesis 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.

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 TIAC 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 TIAC tool. The system analyst interpreted and verified the messages, and determined whether these messages were the intended behavior of the Test Driver.

Future Work

Currently, the DEVS Test models are written in C++ source code. When the models are changed, the Test Driver requires to recompile 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] 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.

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

[3] 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.

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

[5] Cho, Y., B.P. Zeigler, H.S. Sarjoughian, "Design and Implementation of Distributed Real-Time DEVS/CORBA," IEEE Sys. Man. Cyber. Conf., Tucson, Oct. 2001.

[6] High Level Architecture Run-Time Infrastructure RTI 1.3-Next Generation Programmer's Guide Version 5, Defense Modeling and Simulation Office.

[7] Hall, S.B., S.M. Venkatesan, B.P. Zeigler and H.S. Sarjoughian, "Object Oriented HLA Interface Design for Military Simulations," Summer Computer Simulation Conference, pp. 267-273, Vancouver, Canada, July 2000.

[8] 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.

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

[10] 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

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

[12] 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.

[13] 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

[14] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide*. Addison-Wesley. 1999.

[15] DEVSJAVA, http://www.acims.arizona.edu/SOFTWARE/software.html, last accessed April 15, 2006.

[16] Krasner, G., Pope, S., "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system," Journal of Object Oriented Programming, Vol. 1, No. 3, pp. 26-49, 1988.

[17] 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.

[18] 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.

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

[20] 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

[21] Zeigler, B., Hammonds, P., Fulton D., Nunn K., and Mak, E., “Simulation-based Testing of Emerging Defense Information System,” In progress.