SIMULATION-BASED FRAMEWORK FOR AUTOMATED TESTING OF TACTICAL MESSAGE SYSTEMS

by

Stephanie Jarboe

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 COMPUTER ENGINEERING

In the Graduate College

THE UNIVERSITY OF ARIZONA

2009

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the copyright holder.

SIGNED: ______________________

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

______________________________	November 23, 2009
Bernard P. Zeigler	Date
Professor of Electrical and Computer Engineering	

Acknowledgements

I would like to acknowledge Dr. Bernard Zeigler, Dr. James Nutaro, Dasia Benson, Jeffery Johnson, Dale Fulton, and all of the other people at the Joint Interoperability Test Command who have helped me finish the research and come up with the ideas behind this thesis. I would also like to thank them for supporting me while I was working to get my Masters degree.

I would like to extend my heartfelt thanks to my committee members, Dr. Michael Marcellin and Dr. Roman Lysecky without whose patience and help this thesis would not have come to be.

I would like to acknowledge the staff at the Arizona Center for Integrated Modeling and Simulation at the University of Arizona, who helped immensely with the research behind this thesis.

I would like to acknowledge my husband, Matthew Jarboe, for his support during the years and for his help with completing this thesis. I also want to thank my friends, Melanie Yaun, Patrick Kelley, Sarah Kelley, and Brandon Smith, for putting up with the endless amounts of homework and making me have fun occasionally.

I would also like to acknowledge my parents, who paid for my bachelors degree and encouraged me in my love of learning and my desire to go back to school.

Table of Contents

Table of Figures

1 Abstract

With the passing of time, combat systems grow and evolve. It follows then, that testing tools must grow and evolve with the combat systems. As testing requirements become more complex, testing systems need to change to encompass more complex behaviors. In earlier work [5], a system was created to generate test models from an XML rule base and run them in a distributed test environment. The system used the Model/Simulator/View/Control (MSVC) [1] design pattern, based on the Model/View/Control (MVC) [11] design pattern, to separate the complexities of developing models, simulator, and distributed simulation, and to allow for the reuse of models in a repository. This thesis takes that earlier system and expands the model design, in order to create a new system, capable of more automated test case generation. Creating the new system consisted of three tasks: Characterizing a message-passing system, defining a modeling framework microarchitecture to better handle complex test behavior, and automation of test-case creation. The formalism developed to characterize a message-passing system is determined by the Message Interactions, Conditions, and Ordering (MICO) inherent in a message-passing system. The modeling framework microarchitecture defines the structure of the test cases, based on the MICO specification determined. The combination of the MICO specification and the structure allows for automatic generation of the final test case.

This methodology was used to generate 102 test cases that were then used to verify the conformance of a Command and Control (C2) Program developed by the

Single Integrated Air Picture (SIAP) Joint Program Office (JPO) to Military Standard (Mil-Std) 6016C. All results from test scenarios were collected and analyzed using the Theater Air and Missile Defense (TAMD) Interoperability Assessment Capability (TIAC) tool. The messages were verified, and the scenarios were validated, by system analysts at the Joint Interoperability Test Command (JITC).

2 Introduction

2.1 Overview of Context

Modeling and Simulation are widely used concepts. In most cases, modeling and simulation are used to predict the behavior of a system in a situation that is too expensive, either in money or in manpower, to perform in real life. Lately, modeling and simulation have been widely used to predict the feasibility of possible future actions, based on known market or situational conditions. In 2001, the Object Management Group (OMG) put forth the Model Driven Architecture (MDA) concept. This concept was introduced as a way to create a model of a system, and then use that model throughout the lifecycle of the product. Eventually, that model would become the system itself [6]. The DoD adopted that idea, which led to the mandate to implement modeling and simulation testing tools early in the lifecycle of new systems [2][3]. One of the projects which sprang from this decision was the Automated Test Case Generator research project [5]. This project and its subsequent incarnations are discussed in Section 3.4.

The system developed in this thesis is based on concepts from system theory, modeling and simulation theory, knowledge representation theory, software design patterns, and software engineering paradigms. All of these concepts are incorporated into the design of this system.

The systems theory basis involves cybernetics particularly, and how it applies to system design. Cybernetics is a term coined by Norbert Wiener, and is the study of communication and control systems in living organisms and machines, in terms of mathematical analysis [23]. The mathematical theory behind cybernetics expands into the theory behind modeling and simulation, especially in terms of the Discrete EVent Simulator (DEVS), and how the behavior of an overall system can be determined by the sum of the behaviors of the parts [8]. This concept allows the system to be defined by parts, and then verified via closure theories. Section 3.1 discusses the DEVS formalism and Section 3.1.2 and 3.1.3 discuss some extensions of the formalism used in this thesis.

Knowledge representation theories allow systems to be defined in terms of structure and behavior, and in terms of the decomposition of the system. The Universal Modeling Language (UML) is the most commonly used knowledge representation language. UML is mostly used in programming to represent model hierarchies and has multiple different families of visual representations of systems, both structural and behavioral. UML is an extremely large and complex standard. Due to the complexity and some of the limitations of the language, it is not used in this thesis. System Entity Structure (SES) is a knowledge representation language commonly used in conjunction with the DEVS formalism. It is smaller in terms of the rules of specification, but it has some capabilities that are difficult in UML. In particular, formal methods have been developed in SES for defining coupling of a model. These coupling definitions facilitate in the automation process described in Section 4.4 below.

Software design patterns define abstractions that allow systems to be separated into logical components and developed independently. The software design pattern used in this thesis is an extension of the Model/View/Control (MVC) software design pattern created by Krasner and Pope [11]. This extension is known as the Model/Simulator/View/Control (MSVC) software design pattern, and is discussed in further detail in Section 3.4.1. The separation of components in the MVC and MSVC design patterns allow use of the Component-Oriented design paradigm.

Component-Oriented design, or Component-based Software Engineering (CBSE) is a software engineering paradigm, similar to the Object-oriented design paradigm. CBSE differs from Object-Oriented (OO) design in that Component-Oriented design is based on the reuse of prior existing components while OO design is based on the creation of systems by modeling real-world objects. CBSE relies on the creation of generic sets of related functions in software packages or modules. These modules are then reused over and over in different systems. The use of CSBE saves time by allowing programmers to reuse functionality [5].

The system described in this thesis was designed to support the testing community in their increasing needs. This system is based on an existing modeling and simulation application. It is designed to replace the existing system by redefining the existing system in a manner that would allow the testing of expanded capabilities, as well as the automated creation of test cases. The testing community referred to above is the community responsible for testing the behaviors of message-passing systems. The term message-passing systems can refer to systems like internet protocols, instant messaging standards, or web services, as well as systems like radio protocols. In the specific terms of the system described in this thesis, the message-passing system being tested was the Tactical Data Link, J-series (TDL-J) [12], otherwise known as Link 16. Link 16 is a standard used to pass tactical information between participants in a region, also known as a theater. Participants can be aircraft, ships, or ground forces. In the past, testing of Link 16 consisted mostly of performing scripted scenarios, and then verifying via post-analysis if the messages were created correctly and sent at the correct time. The predecessor of this system was designed as a way to verify that messages were correctly sent and received in real-time, rather than using post analysis. The predecessor system proved that real-time analysis was useful and saved time, and was therefore requested to expand its capabilities. The process of expansion led to problems. The system described in this thesis is the proposed solution to the problems with its predecessor.

The system is implemented and proved effective via testing against a System Under Test (SUT) and post-test verification. The SUT tested is a C2 system developed by the Single Integrated Air Picture (SIAP) Joint Program Office (JPO). The tests were validated using the Theater Air and Missile Defense (TAMD) Interoperability Assessment Capability (TIAC) tool. The validation was done via post-analysis by collecting all messages sent by all systems. The data was then verified by Subject Matter Experts (SMEs).

2.2 Statement of Problem

The Automated Test Case Generator (ATC-Gen) Test tool is a Discrete-Event Simulation (DEVS) based system used to conduct Standards Conformance Testing of message passing systems for the Department of Defense (DoD). The project was first implemented as a message injector, designed to stimulate Systems Under Test (SUTs) with messages in order to test their capabilities. It was later expanded to perform standards conformance testing, using an Extensible Markup Language (XML) [7] rule base to semi-automate the generation of test cases. As the project expanded, the requirements began to include conditions that covered more than simple reception or transmission, but the concepts on which the automation was built were never reexamined. The project began to expand too quickly for the automation concepts, and they were left behind. This led to many problems. These problems are described in more detail in Section 3.4.4.

The major problem with the test case automation capability lay in the fact that the automation software did not take into account the states a model had to transition through, the internal behaviors that may have existed in a processing state, or the finer details of integrating with the expanded test capabilities, in particular, interfacing with messaging protocols. In addition, the test case automation capability did not have sufficiently small and well-defined test case models to allow simple reuse in various, unrelated test case situations. These deficiencies called for a redefinition of the DEVS models, a redefinition of the model system, and also a redefinition of the automation software.

The motivation behind this thesis was to redefine the DEVS models, the model system, and the automation software for The Automated Test Case Generator (ATC-Gen) Test tool, to improve automation process for the ATC-Gen Test Tool, to develop a new framework for automating test cases used to test message passing systems, and to design and implement a new software abstraction for message passing systems. The objectives of this research project were to provide a methodology to capture the behavior of Message Passing Systems in order to facilitate test scenario creation, to simplify and decrease test case generation process time by applying DEVS and SES generation methodology, to increase the modularity and code reuse in an existing system, to formulate a new architecture for an existing system, and to provide a methodology that allows developers to create high performance distributed real-time systems that are extensible, flexible, interoperable, reusable, and reliable.

2.3 Organization and Content

There are four remaining sections in this thesis. The next section of the thesis covers the background material behind the project. It has an introduction to the concepts behind the project research, as well as a history of the predecessor projects. The concept introduction covers three main topics: Discrete Event Simulation, System entity structure, and testing using Discrete Event Simulation. The history section covers three previous incarnations of the research project, as well as the first reconfiguration of the original research project. The first reconfiguration failed to satisfy the requirements, but led to the concepts behind the second, successful reconfiguration.

The fourth section of the thesis covers the research behind the project, and discusses the contributions made. The two major topics covered in this section is the Message Input, Condition, Ordering (MICO) concept, which is the major contribution of this thesis, and automated generation of test cases using Finite and Deterministic Discrete Event Simulation. These two concepts allowed the creation of a system that met the requirements necessary for this reconfiguration.

The fifth section covers the results of the reconfiguration. It covers the result of the automated test case generation concept, and also covers the testing done against the SUT. However, due to the nature of the SUT, only general results may be discussed. The sixth section is the conclusion. It summarizes the major points of the paper, as well as mentioning future work to be done on the project.

3 Background

The system in this thesis is based on a number of different theories. The Cybernetics portion of System Theory led to the definition of the Discrete EVent Simulation formalism (DEVS). The DEVS formalism was the theory upon which the A Discrete EVent Simulator (ADEVS) C++ library was implemented. The ADEVS library contains the code to implement DEVS in a C++ environment, such as this system is implemented. The System Entity Structure is the tool used to describe the structure of the system implemented.

The system is the extension of a long-standing research project, called the Automated Test Case Generator (ATC-Gen) project. The Model/View/Control software design pattern was extended as an early topic in the project, and this extension is exploited in later versions of the ATC-Gen system. The below sections cover the theories behind ATC-Gen as well as the early history and evolution of the ATC-Gen research project.

3.1 DEVS Introduction

The DEVS [8] formalism is one of the foundation concepts on which the system designed in this thesis is based. It is an established and widely used modeling and simulation formalism. There are many incarnations of DEVS, but for the sake of brevity, only the branches used in this paper are discussed in the below sections.

3.1.1 Discrete EVent Simulation (DEVS)

In the basic DEVS formalism, model components are represented by a seven-tuple, defined as in Figure 1. The seven-tuple is composed of sets and functions. The set X is a set of input values, which defines what inputs the model can or will receive and process. The set S is composed of all possible states a model can enter. The set Y is a set of the possible output values that the model can create. The function δ_{int}, or the internal transition function, maps the set S to itself to define the states that the model will transition between in the absence of external input. The function δ_{ext}, or the external transition function, is a cross-product of a container, Q, and the set of inputs, X. The set Q is the *total state set*, and is defined as a set of pairs, consisting of all possible states and all possible elapsed times. The function λ maps the set of states, S, to Y, which is a set of output values. The function ta maps the set of states, S, to $\boldsymbol{R}^{+}_{0,\infty}$, or the set of positive real numbers from zero to infinity. Together, a seven-tuple defines a model of a state machine and its behavior over time, including the capability to handle inputs and the capability to output values.

A *Discrete Event System Specification (DEVS)* is a structure

M=<X,S,Y,□int, □ext,□,ta>

where

X is the set of input values

S is a set of states,

Y is the set of output values

δint: S→S is the *internal transition* function

Δext: Q×X→S is the *external transition* function, where

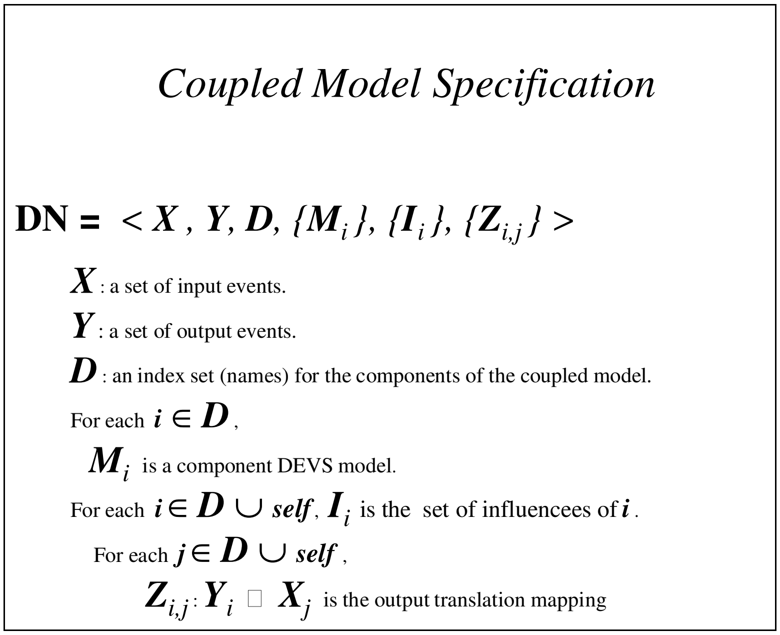
Q = {(s,e)|s εS, 0 □ e □ ta(s)} is the total state set,
and e is the time elapsed since last transition

λ: S→Y is the *output function*

ta: S→$R^{+}_{0,\propto}$ is the *time advance* function

Figure 1 - DEVS Specification

Inputs are received on input ports, and outputs are transmitted on output ports. The coupling of these ports defines DEVS Coupled Models. DEVS Coupled models are formally defined as in Figure 2. This 6-tuple has analogs to the DEVS Atomic models, allowing the system to be extended into a hierarchy. In the DEVS modeling and simulation formalism, a set of models can be grouped into a single model, which can be treated as a black-box and coupled to form hierarchical models. This allows for complexity hiding and black-box reuse of atomic and coupled models. The principle of Closure under Coupling states that, because every DEVS coupled model has a basic DEVS model equivalent, a coupled model is a closed system that can be coupled to other models.



Coupled Model Specification

$$\mathbf{DN} = < X , Y, D, \{M_i\}, \{I_i\}, \{Z_{i,j}\} >$$

X : a set of input events.

Y : a set of output events.

D : an index set (names) for the components of the coupled model.

For each $i \in D$,

M_i is a component DEVS model.

For each $i \in D \cup \mathit{self}$, I_i is the set of influencees of i.

For each $j \in D \cup \mathit{self}$,

$Z_{i,j} : Y_i$ □ X_j is the output translation mapping

Figure 2 - Coupled DEVS Specification

As an example of a DEVS model, consider a simple system. A system can be defined as any set of interacting entities that together form a whole. An example of this is a parking meter. A parking meter has two states, expired and paid. The state expired has a time-advance function of infinity, meaning that, barring the influence of external events, the model will stay in the state expired indefinitely. With a normal parking meter, the time advance of the paid state is not constant, but depends on the amount of money paid into the meter. Each time a quarter has been inserted, if the current state is expired, the state changes to paid and 15 minutes are added to the time advance function for paid. For the sake of simplicity, only quarters are allowed as input, and if another quarter is inserted while in state paid, it is ignored. When the time advance function for paid ends, the state returns to the expired state, and a flag is output called “Expired!”. The formal definition of this DEVS model is as shown below:

X = {?Quarter}

Y = {Expired!}

S = {expired, paid}

ta = τ(paid)=15, τ(expired)= ∞

δ_x(expired, ?Quarter)=(paid)

δ_x(paid, ?Quarter)=(paid)

δ_y(paid)=(Expired!, expired)

δ_y(expired)=(φ, expired)

The formal descriptions of models are unwieldy and not readily understandable, so the rest of the models in this thesis will be described in terms of their input, output, and state sets and a diagram, such as the one below. The conventions described in the legend are used throughout the rest of the paper.

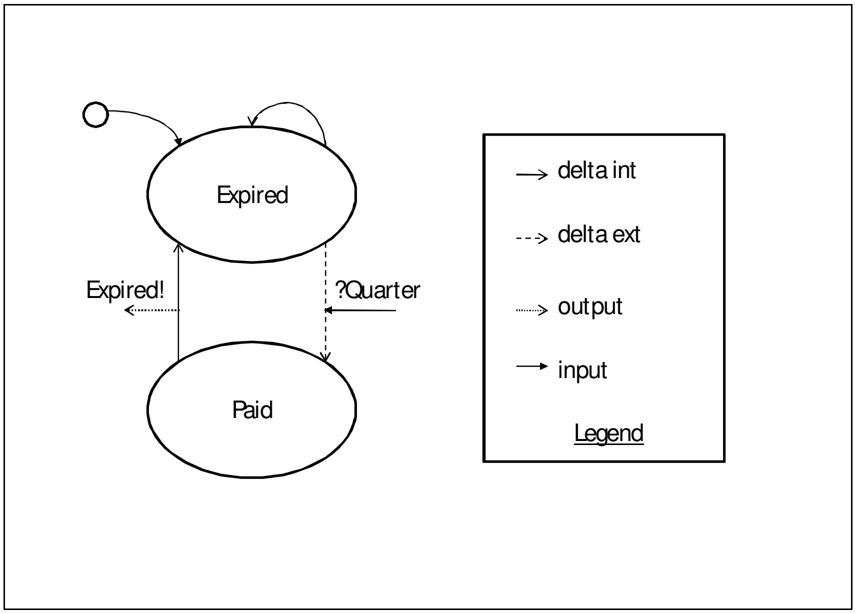
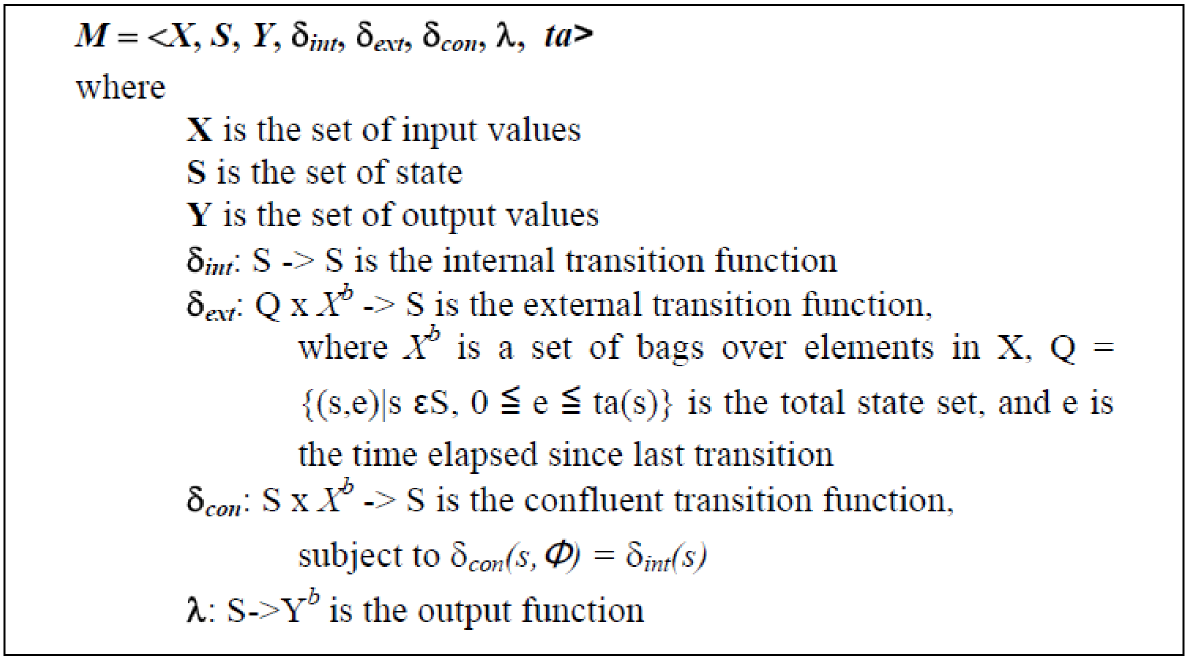


Figure 3 - Parking Meter Example DEVS Diagram

3.1.2 ADEVS and Parallel DEVS

A Discrete EVent Simulator (ADEVS) [10] is an implementation of DEVS using the C++ programming environment. It in particular implements the extensions of DEVS known as Parallel DEVS and Dynamic DEVS. The difference between the Parallel DEVS formalism and the classic DEVS formalism is in two parts. First, it allows bags of inputs to be input to the external transition function. A bag, otherwise known as a multiset, is a set that allows multiple instances of any single object in the set. Second, it introduces the confluent transition function, δ_{conf}. The confluent transition function determines what happens when an internal transition and an external transition happen simultaneously. Its purpose is to control the collision behavior when receiving external events at the time of the internal transition

The first difference also leads to changes in the definitions. The function δ_{ext}, or the external transition function, is a cross-product of two containers, Q and X^b. The set Q is the *total state set*, and is defined as a set of pairs, consisting of all possible states and all possible elapsed times. X^b is a collection of bags, where a bag is a set that can contain one or more instances of the same element. These bags are made up of elements of X, and represent the input to the system at a particular time. The function λ maps the set of states, S, to Y^b, which is a bag of output values.



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

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 = $\{(s,e)|s \varepsilon S, 0 \leqq e \leqq ta(s)\}$ 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 4 - Parallel DEVS specification

Parallel DEVS is the version of DEVS implemented in ADEVS, and as such, the rest of the models in this thesis are assumed to take bags as input, and to give bags as output. The δ_{conf} function is not explicitly described for models in this thesis, but it is generally assumed to handle the external transition first, followed by the internal transition.

3.1.3 Finite and Deterministic Discrete Event Simulator (FDDEVS)

Finite and Deterministic DEVS (FDDEVS) is a restricted version of Classic DEVS defined such that [13]:

(1) The sets of events and states are finite,

(2) The time advance is a mapping from states to non-negative rational numbers,

(3) There is no restriction on the occurrences of external events, and

(4) An external input event can either reschedule or continue processing.

Defining these restrictions allows many useful extensions of DEVS. One extension, discussed in [13], is the ability to define finite reachability graphs of FDDEVS systems. Another extension, discussed in [15], is the ability to verify behavior of a model using the finite reachability graphs. Finally, the most useful extensions available in FDDEVS are the XML representations of FDDEVS test cases and the subsequent constrained English representations. These representations are the heart of the generation process used to create models from formalized descriptions.

The XML representation captures an FDDEVS model by tagging the states, the time advance function, the output function, the input ports, the output ports, the internal transition function, and the external transition function. From this tagged data, a program was written to generate FDDEVS java models. Afterwards, a program was written to translate from constrained English constructs into the XML tags for each of the elements. This allowed an FDDEVS model to be characterized in constrained English and then generated into a Java test model. The generation capability created in this thesis is related to this generation process.

Applying FDDEVS to the example above, the specification is almost exactly the same. The only difference is in the following two lines:

δ_x(expired, ?Quarter)=(paid,1)

δ_x(paid, ?Quarter)=(paid,0)

The second number in the defined ordered pair defines whether or not the model reschedules the time advance function. The internal schedule of a state $s \in S$ is updated by $\tau(s')$ if $\delta x(s) = (s',1)$, otherwise(i.e., $\delta x(s) = (s',0)$), the schedule is preserved [15]. Hence, inputting a quarter only changes the schedule if the current state is expired.

3.2 System Entity Structure (SES)

The System Entity Structure (SES) is a formalism often used in conjunction with DEVS to describe coupled model hierarchies. SES is a knowledge representation scheme, which is used to organize systems and the relationships between objects in systems. SES is used to represent decomposition, taxonomy, and coupling relationships among the parts of a system. Decomposition is the description of the component parts of a system. Taxonomy is the description of the classification of parts of a system. Coupling relationships are descriptions of how the component parts of a system are attached to form the system.

An SES is represented as a labeled tree with attached variables. These attached variables must satisfy the following six axioms [16]:

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

SES Diagrams are composed of four types of nodes: Entity, Aspect, Multi-Aspect and Specialization. Entity nodes represent real-world objects. Aspect nodes represent the decomposition of an entity. The children of an aspect node represent component elements of the Entity node above. Multi-Aspect nodes allow selection of multiple different components, or selection of many of the same component, or some combination of the two. The children of a multi-aspect node represent components whose number can vary in different implementations of a system, or components that can exist in different combinations in different implementations. Specialization nodes represent the taxonomy of an entity. The children of a specialization node represent variations of the Entity node above. The first axiom states that nodes must alternate between Entity nodes and Aspect or Specialization nodes. Figure 5 represents an example of an SES.

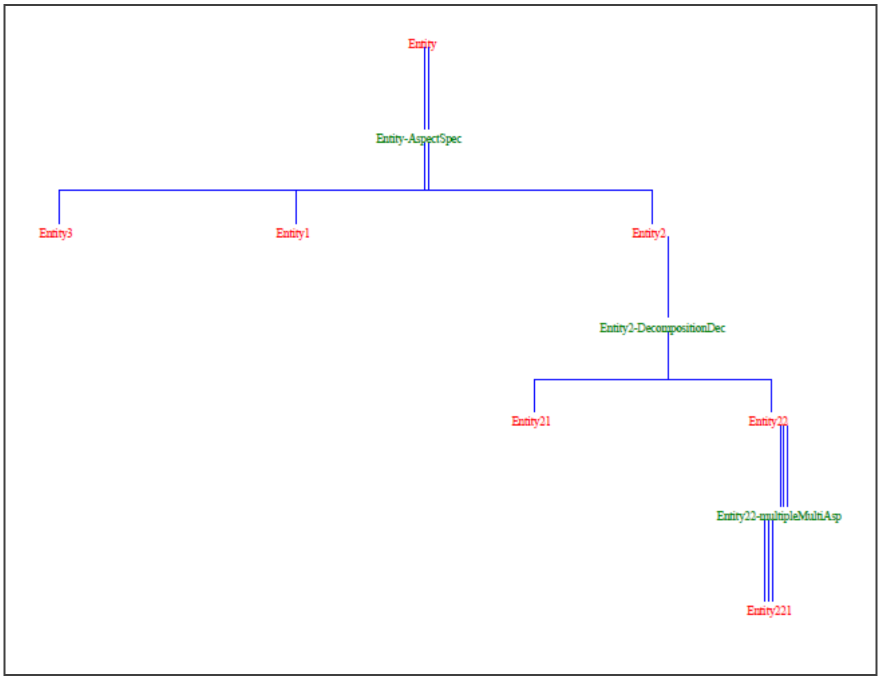


Figure 5 - SES Example

A Pruned Entity Structure (PES) is a SES in which all Specification nodes have been “pruned’ or a single child from each Specification node is selected. The pruning process is the selection of a child at each Specification node and the selection of the appropriate implementation of multi-aspect nodes. This is equivalent to making all final implementation decisions between all possible choices.

3.3 Related Fields and Research

Testing using DEVS-based simulation has been done by many different researchers since DEVS was created [1][4][5][6][20][21]. Testing Message Passing Environments is a less common topic of research, but some work has been done towards describing formalism [18][19]. These are the two major fields to which the topics of this thesis apply. This section covers some of the related work in the fields of DEVS-Based Simulation and Message Passing Environment Testing.

In his doctoral dissertation, Mittal [6] covers much of the background of test case generation. In fact, the latter portions of this thesis, on FDDEVS test case generation, are based on work done by Mittal [6] and Hwang [16]. One of the topics covered in [6] was how DEVS-based testing can add to the process of testing software architecture by providing a mathematical framework to the execution process, allowing for more formalism. This is a fundamental advantage to any DEVS-based testing system, including the system developed in this thesis.

The paper by Hammonds and Nutaro [1] talks about the earliest predecessor or the system developed in this thesis. It discusses the Model/View/Control design pattern and its extension, the Model/Simulate/View/Control design pattern. These design patterns and their combination with DEVS are what made the current system possible. Without the separation of systems proposed in the software design patterns, the test driver system design would be too complicated and require changes to too much code for the reconfiguration performed in this paper to be feasible.

In [4], Zeigler, et.al. put forward a process to use DEVS modeling and simulation throughout the lifecycle of a product. This paper, along with the DoD requirements in [2] and [3], led to the requirement that modeling and simulation be integrated into all stages of the development of DoD systems. This directly led to the creation of modeling and simulation environments for testing systems, from which the systems described in [1] and [5] derived support. Zeigler, et.al. [20] covers using DEVS as middleware for testing. This paper seeks to add formalism to middleware the same way that DEVS adds formalism to the model portion of the system developed in this thesis. The thesis in [5] describes the system developed from the predecessor in [1] that is described in Section 3.4.2. Shang and Wainer [21] discusses using DEVS in a real-time environment, and discusses dynamic structure DEVS.

Bateja and Mukund [18] discuss the difficulties in testing whether a message passing system conforms to a specification. Their paper proposes a tagging system to solve the problem of implied scenarios. It has many constructs that correspond to components in DEVS formalism and MICO, which is developed in this thesis. The message passing automata used in [18] to make the message passing graphs determinate correspond in almost every facet with DEVS message passing environments: Each message is labeled and associated with a time, a construct similar to the delta external function determines the next state for message-receiving processes, and a construct similar to the delta internal function determines the next state for message sending processes. The major difference is that the channels in [18] are FIFO, where the ones in Parallel DEVS are determined by the order that messages are added to the bags. In comparing the systems, it can be seen that MICO when used in conjunction with DEVS solves many of the same issues as [18], while having the added advantage of being based on DEVS, which is a well-established modeling and simulation formalism. MICO also covers message content condition checking, which is not mentioned in [18].

Tsiatsoulis et.al. [19] talk about message passing in parallel environments, such as parallel computers. Their paper puts forward their product, named Ensemble, and a methodology for testing and debugging programs using their product. Ensemble works in a similar manner to the system developed in this thesis: Both use defined structures to generate final models, both have constructs to model port couplings, and both use predefined components to build the final system. Ensemble uses Colored Petri Nets to express its specifications, while the system developed in this thesis uses DEVS coupled models.

3.4 Early History

3.4.1 Model/Simulator/View/Control (MSVC) and the Origins of ATC-Gen

The first predecessor of the ATC-Gen research project was created by Dr. James Nutaro and Phillip Hammonds, in 2004 [1]. As mentioned above, Modeling and Simulation (M&S) became a required portion of the development process in response to DoD directives of the time [2][3]. The system they created in response was an implementation of the Model/Simulator/View/Control (MSVC) simulator design pattern and the DEVS formalism. The MSVC design pattern was based on the Model/View/Control (MVC) system design pattern, used in many software programs. MVC was created by Trygve Reenskaug and later described by Krasner and Pope [11]. It was designed to separate the business model portion of the code from the control and the presentation in order to allow separate development, modification, and testing of each component. In applying this model to a simulation, the pattern separated the simulator and models from the system control and user interface.

Figure 6 shows a diagram of the MSVC design pattern. The extension to the MVC design pattern was an abstraction of the model from the simulator in Modeling and Simulation applications, allowing for simulators to be developed independently from the models the simulator would run. It was envisioned as a solution to problems with multiple distributed simulation protocols in defense-related simulation environments. Applying the MSVC design pattern to a system allowed the protocol-specific code to be developed independently of the models, in the View and Control components. The independent development allowed for multiple different protocols to be developed for a system, and then used interchangeably as needed. This led to a simple way to create a system that could implement multiple protocols, while retaining the underlying behavior of the models and simulator.

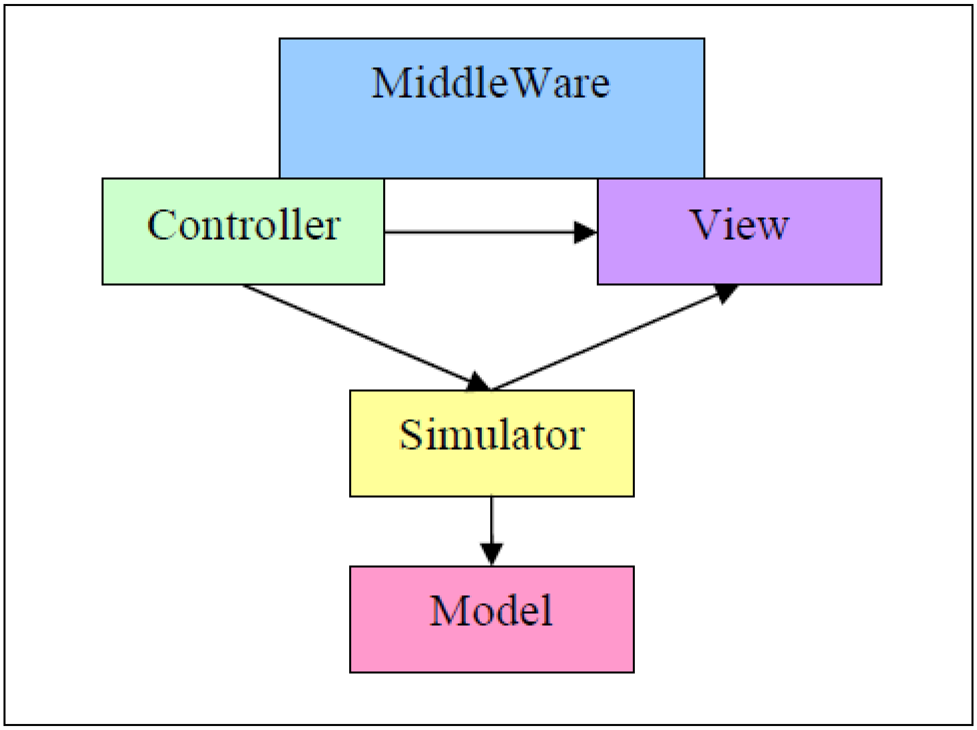


Figure 6 - MSVC Design Pattern Diagram

The system designed using the above concepts was referred to as the Joint Utility Player. The Joint Utility Player (JUP) was a system for simulating tracks, or platforms, over protocols such as the High Level Architecture (HLA) [9] and a North Atlantic Treaty Organization (NATO) standard known as SIMPLE, which was used to encapsulate TDL-J messages. The original JUP did not include any support for modeling scenarios, but rather only for simulating the movement of objects in space, commonly referred to as tracks. The JUP was designed to send messages at intervals using data gathered from files. The JUP was coded using the C++ implementation of DEVS known as ADEVS, allowing the system to keep track of simulation time. The major purpose of the JUP was to inject positional and TDL-J messages in order to set up scenarios for testing.

3.4.2 Original ATC-Gen Concept

The Automated Test Case Generation (ATC-Gen) research project was intended to expand on the testing ideas first put to practice in the JUP. The motivation was to, by implementing message reception capabilities, make a system that was capable of verifying scenarios independently. By creating a system that could automatically verify if a scenario was compliant, it created test cases that could be analyzed at run-time, rather than by post-analysis.

From the beginning, the ATC-Gen research project was designed as a repository of separate models, which could be selected and ordered in such a way as to recreate a test scenario. A test scenario is a series of behaviors that are expected to occur in a real world environment, and stimulate conditions under which conformance to specific rules in a standards document can be tested. A test case is the set of models and the coupling between them that recreate those behaviors in the ATC-Gen Test Driver. The ATC-Gen Test Driver is the software implementation of the concepts described in this thesis. For the rest of the document, the term ATC-Gen will be used to refer to the ATC-Gen System, which is implemented as the ATC-Gen Test Driver.

ATC-Gen is a message-handling system, designed to test the behavior of sections of the Link 16 message standard [12], used by the United States military and NATO. ATC-Gen was first conceptualized as a simple, automatable, message input/output checker. Models called holdSend, waitReceive, and waitNotReceive were put in sequential order to test the input/output behavior of a System Under Test (SUT). An XML structure was formalized so that standards documents could be captured in XML. The XML rules captured from the standard were fed into a program that generated a hierarchical diagram of which rules stimulated others. Then, an analyst would select a path including the desired rules, and either beginning in a message reception, or ending in a message transmission. These rules would be fed into yet another program, called the Test Model Generator (TMG). The TMG created test cases by creating instances of existing ADEVS models known as primitives. These primitives were so called because they encapsulated the most primitive behaviors of the message passing system: receiving and transmitting messages. In order to model system time, receiving messages was associated with a waiting time period, and the resulting primitive was the waitReceive primitive. The DEVS diagram for the waitReceive primitive is shown in Figure 7.

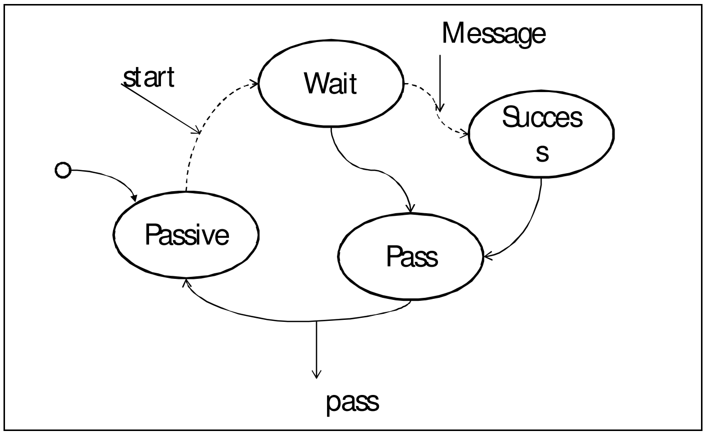


Figure 7 - waitReceive DEVS Diagram

Similarly, the transmission of messages was associated with a time to hold the message until it was to be sent, resulting in the holdSend primitive. Figure 8 shows the DEVS diagram for the holdSend primitive. The TMG would take rules that included message transmission, and use them to generate instances of holdSend primitives to transmit messages to a SUT. If the rules specified a message was to be received, the TMG would generate a waitReceive primitive.

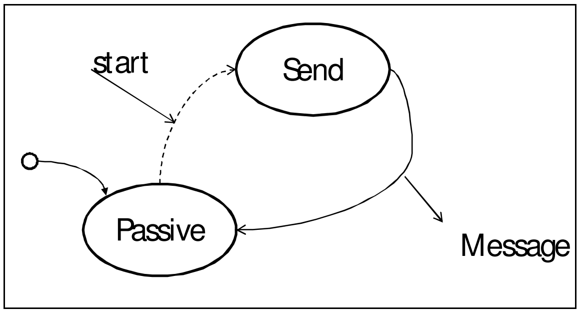


Figure 8 - holdSend DEVS Diagram

Test cases were created from the perspective of the SUT, in order to capture all of the rules, and then "reflected" in order to create a proper environmental frame. The process of reflecting was done by changing all holdSends to waitReceives, and vice-versa. The TMG would also take the input variables into the rules and use them to generate the message that would be output by the holdSend primitive, after the reflection process was completed. Thus, with minimal human interaction, test cases could be generated.

After test cases were generated, they were integrated with an existing simulator and the Translator interface first conceived in the JUP. This system was an implementation of the MSVC model discussed in the previous section. The View and Control portions of the system were implemented in the Translator interface. The

Simulator was created using ADEVS library functions and Microsoft Foundation Class (MFC) threading library functions. The separate portions of the MSVC system were implemented using separate threads. The system is described in more detail in previous work [5].

The original ATC-Gen implementation was the first incarnation that attempted to model more complex behavior. An example of complex behavior that required testing was correlation. A full explanation of a correlation model is covered in Appendix C, Section 1.

3.4.3 ATC-Gen Evolution

The original ATC-Gen system was very simplistic. This simplicity made automation possible in the original system. However, it also limited what the system was capable of. The system did not take into account the content of the messages it sent and received. It required all data for the system, except positional data, to be entered at compile time. As a result of this, if a message changed, the associated test case would need to be recompiled. This made operation in real-time, or against real-time generated data, nearly impossible. It relied upon pre-recorded positional information, formatted into text based files. Testing the existing system against new SUTs required interoperable behaviors, and the system limitations made adaptation difficult. These requirements led to an expansion of the original capabilities of the ATC-Gen Test Driver.

A new mode of ATC-Gen was created to deal with these new requirements, called Reactive mode. This mode was called Reactive because the data used in the message was taken from live data being reported by the SUT, modified, and reflected back. This allowed the Reactive Mode ATC-Gen to run against live SUTs, without having to code full messages at compile time.

Unfortunately, the advantages of Reactive mode were accompanied by disadvantages. The nature of the Reactive mode and the time constraints put on its development made the simple holdSend/waitReceive models of the past impractical. They were replaced by longer DEVS models that encompassed entire scenarios. An example is covered in Appendix C, Section 2.

3.4.4 Problems

The evolution of ATC-Gen was fraught with troubles. The original automation system did not lend itself to generation of the new Reactive Mode test cases, but time constraints made a complete system redesign impossible. The original XML rule base lacked metadata to describe which state changes in the XML rule sets map to actual state changes in the ADEVS models. It lacked metadata to describe information handling or condition checking and what state changes they may induce. The formalism used to create the XML rules just could not encompass all of the changes necessary to generate the new behaviors desired by the SUT.

Because the automation scheme could not handle the requirements for new test cases, the system was expanded manually, by analysts coding new behaviors into ADEVS models. The waitReceive and holdSend models on which the system was originally built were used as templates, with additional code put in to fulfill the new requirements levied by the new SUTs. When these were not enough, the expanded code was itself used as templates and expanded, until the models that were supposed to be "primitives" became nearly complete test scenarios, with an occasional holdSend coupled to the end or a waitReceive coupled to the beginning. The models themselves were hundreds or thousands of lines of code, and required days of work to create. Due to the hurried nature of improvements to the system, much of the formalism and automation of the simplified version of ATC-Gen was lost entirely.

Another problem with the Reactive Mode was size. As more specialized models were needed to test more specific scenarios, the repository of models grew. What had at one time been a small set of primitives became a large repository of specific behaviors. When new test models were required, a new model was created, using an existing model as a template. It soon became clear that, if a new method of creating test cases was not created, the size of the repository would grow linearly with the amount of the standard covered by test cases.

The automation capability of the simplified ATC-Gen was unable to create test cases in the new Reactive Mode. The major reasons the former automation process did not work involve the oversimplification of data in the standard. The manner in which the XML was created did not translate directly into the test driver models as they existed in the code. The XML rules were derived directly from the standard, and each rule was modeled in DEVSJAVA as an atomic model. DEVSJAVA is a Java implementation of the DEVS formalism. Representing the rules as models was misleading, however, as the rules did not model anything other than whether their necessary input variable appeared in the output of a prior rule. The DEVSJAVA representation created a model of the flow of data. This creates a valid model, but it does not create the model of the overall behavior that was desired for the Test Driver test cases. Additionally, the Test Model Generator program turned the DEVSJAVA models into files that contained nothing more than input and output interaction. It left all of the rules and behaviors that were captured out of the generation process, creating such a minimalistic model that the output was incapable of performing any necessary behavior, other than receiving and transmitting messages.

Another problem with the automation process was that modeling one half of an interaction was not sufficient to capture the behavior. “Mirroring” did not work, because the transactions were not symmetrical. Different processing happened on opposite sides, and simple mirroring left out important details. On the other hand, modeling both sides was unnecessary. Only certain details of each side needed to be modeled. Constraints that affect what messages were sent, data that was changed which lead to fields in messages being changed, and input and output were all that required modeling. The XML rules modeled a large number of rules that either did not apply to the simulation, or were assumed not to occur due to the nature of the scenarios tested.

These problems made clear the need to restructure. The structure of the test cases and the structure of the automation were unable to support the growth of the system. This led to the research explored in this thesis. It took two revisions to come to a final working version. The two systems conceived are detailed in the following sections.

4 Approach and Description of Research

4.1 First Reconfiguration

The research behind this project was performed in two stages, which led to two different redesigns. The first redesign of the code focused on the necessity to encompass the pass or fail behavior of full scenarios, as opposed to single message interactions. This was the requirement that made the Reactive mode so successful as opposed to its earlier counterpart, alongside the ability to create messages at real-time. Taking the specified scenario behavior as central, the design was created as specified in the following section.

4.1.1 Original Design Concept

The first redesign of the code attempted to solve some of the problems and limitations inherent in the evolutionary process that ATC-Gen underwent. This attempt to redesign the system used modular message-handling models connected to a single coordinator, which was tasked with retaining the state memory for each test model. The state memory kept track of the scenario, and the current state of the test case in respect to the scenario. The approach allowed for more reuse of code per test model than previous designs, while retaining a central controller for inter-message interaction. The design called for a repository, consisting of a set of message handlers and a family of test coordinators with the ability to select their necessary set of message handling components.

The original redesign concept used the concepts inherent in the Reactive mode extension to ATC-Gen, in that the information in the messages was to come from incoming messages. These incoming messages would be reflected back to the sending SUT, with positions extrapolated from the received position based on the time elapsed between reception and transmission.

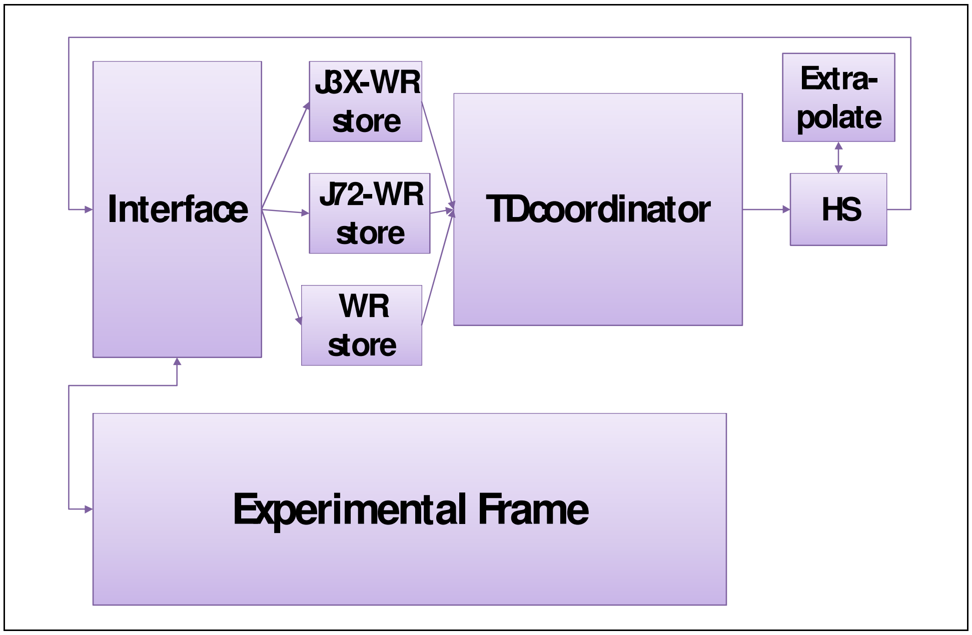


Figure 9 - Original Redesign Concept

The first proposed design was a modular version, rather than a serially-coupled one. Figure 9 shows the design of the system, minus the port interfaces. The models are described in detail in Appendix B: First Reconfiguration Design. The system revolves around a central coordinator, called TDcoordinator, which is responsible for the scenario behavior. The system was designed to have a repository of TDcoordinator models for each scenario to be tested. Coupled to the input of TDcoordinator were a set of models inherited from a model known as waitReceiveSend. This model family was supposed to fill the role of "reflector", by combining the waitReceive behavior of earlier models with the creation of new messages from received messages of the Reactive Mode. On the other side, a model known as holdSend was designed to capture all the models forwarded by the TDcoordinator model, send to the Extrapolator all those messages needing extrapolated positional information, and then to send all messages at the appropriate time. The Extrapolator model took received messages and updated the positional information with extrapolated positional information based on received time versus projected send time. The interface model was a placeholder for the Translator model in the system, and the Experimental Frame was a DEVS model designed to verify that the model behaved as expected.

4.1.2 Improvements over Historical Design

The modular Test Driver was a feasible solution to many of the problems facing ATC-Gen. It behaved in the same manner as the prior code, and fulfilled the same requirements as the prior code, as per experimentation with different Experimental Frame models. In addition, it allowed the reuse of message-handling code through inclusion of modular test models, rather than reusing code via copying into multiple, different test cases. It separated the sending and receiving of messages from the test case specific coding requirements by having a centralized model for each test case. By doing this, it reduced the overall size of the models.

The first reconfiguration also formalized many concepts that the original design had overlooked. Particularly, it emphasized the importance of the entire scenario in determining whether a test case passed or failed. In the original design of the automation, only message reception was used to determine whether a test case passed or failed. If the messages were received as expected, the test case was assumed to have passed. This did not allow for conditions, such as the number of messages received or the value of fields. As such, these conditions either had to be coded later by hand, or ignored. The first reconfiguration solved that issue, but not in an optimal way. The first configuration also had many problems, which are discussed in the next section.

4.1.3 Problems

After implementing the preliminary design, problems were found with the model. The models did not allow for modular interchanging of test condition checking. The models inherited from waitReceiveSend implemented the testing of conditions on incoming messages. As such, they had to either include any possible condition that could be tested for compliance in its specific message, which required hundreds of lines of code, or had to be rewritten for specific conditions. This violated one of the requirements that led to redesigning the code, that of modularity for reuse.

Another problem with this design was in the TDCoordinator code. The TDCoordinator models were overly large, as they implemented too much of the original code per model. The TDCoordinator implemented the behavior to verify that messages were received in the correct order, as well as the code to create new outgoing messages. This meant that, for each test case, a different TDCoordinator would be required, each implementing many of the same code lines, in particular the lines necessary to modify and forward messages. This violated the requirement for code to be reusable in ways other than copying sections of code.

The non-modularity of the code contradicted the requirements for automation. The reuse of code in multiple models by copying and the fact that so much of the code needed to be copied in order to make useful test cases violated one of the major requirements of the redesign. The size of the models inherited from waitReceiveStore and of the TDCoordinator violated the requirement for small, reusable models. In all, the problems made automation very difficult, and the system was deemed a failure to meet requirements.

4.2 Overview of Final Design

The second stage of research led to the final design of the project. This time, instead of focusing on the scenario as a central idea, the design attempted to go back to the origins of ATC-Gen, by focusing on the message interaction as separate models. This meant revisiting the concept of holdSend models and waitReceive models. The original holdSend and waitReceive models satisfied the requirement for small, modular code. However, they failed in two areas: allowing the testing of message conditions, and testing of scenario-wide behavior. This led to the creation of the Message Interaction, Condition checking, Ordering (MICO) concept, described in Section 4.3. The design based on this concept is described in Section 4.2.1 below.

4.2.1 Final Design Concept

The original design split the workflow into three sections: a set of waitReceive(s), a centralized processing unit, and an output. The final revised design splits the workflow into a sequence of waitReceive(s) and holdSend(s), and a set of processing modules. This allows more reuse of code, because the modules allow for the condition checking to be separated from the input/output. A few abstractions were used to make this system more modular and less protocol-dependent than other ATC-Gen models. First of all, every other version of ATC-Gen relied on messages that were either defined inside of models or hard coded and passed to models. This system took all message processing out of the models and put it into an external module. In this way, the model size was greatly reduced, since all the models needed to do was pass around copies of messages that were created elsewhere. Also, the MICO concept allowed the scenario to be abstracted into three separate portions. By separating these portions, the test case specific code was taken out of the reusable models and relegated to test case specific models named ConditionChecker, Acceptor, and HierarchySequence. HierarchySequence is not shown in Figure 10 because it is the coupled model that encloses the sequence of waitReceive(s) and holdSend(s). Another advantage of separating the test case specific code is that the size of the ConditionChecker is reduced since it only needs to implement the code used in a particular test case, rather than any possible conditions that may need to be checked for a particular message.

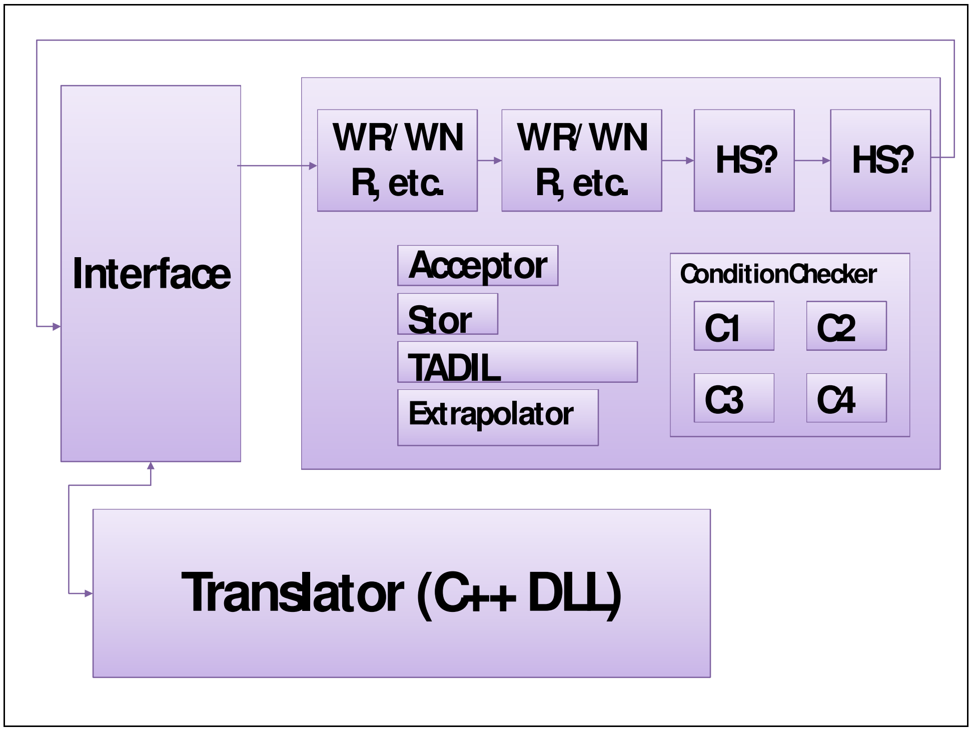


Figure 10 - Final Design

INTERFACE MODEL

The Interface model shown in Figure 10 is the same in both the original and revised designs. This Interface model encapsulates the translation between a Translator Activity and the test models. In the final implementation, this model is replaced by DEVS Activity models, which allow for an interface between real-time message passing protocols, such as Mil-Std 6016C Link 16 and Distributed Interactive Simulation (DIS). Each protocol implemented in ATC-Gen has a separate Activity, and one or more Activities may be present in a test case.

CONDITIONCHECKER MODEL

X = {inStart, inStop, inQueryCondition}

Y = {outCondition}

S = {Passive, Ready, SendCondition}

The ConditionChecker Model contains all the behavior that exemplifies the differences between the ATC-Gen as first modeled and the ATC-Gen as it exists currently. The ConditionChecker allows more granularity of behavior than the minimal input-output pairs implemented in the first incarnation of ATC-Gen.

The Condition Checker is a model that checks conditions and decides whether the model will proceed based on the outcome of the test. The types of conditions tested in this new implementation of ATC-Gen include message reception, message field values as compared to expected values, and timing constraints. Conditions can have 4 possible outcomes: Pass, Fail, Critical Fail, or Stop. Pass indicates that a condition passed, and is used with waitReceive, waitNotReceive, and holdSend models. Stop indicates that, although a prior condition may have passed, the condition no longer holds, and the model will not proceed. It is used with the holdSendRepeat model to indicate that the model should stop its periodic transmissions. Fail indicates that a condition involved in standards conformance failed, but that the test case as a whole is not corrupted by the failure and may continue. Critical Fail indicates that a condition necessary for the validity of the test case failed, and that all subsequent models are no longer valid, requiring the test case to end. The ConditionChecker model is a test case specific model, so a different copy will exist for each test case. Currently, this model is created in a template-based fashion. The model described in this section is the template model.

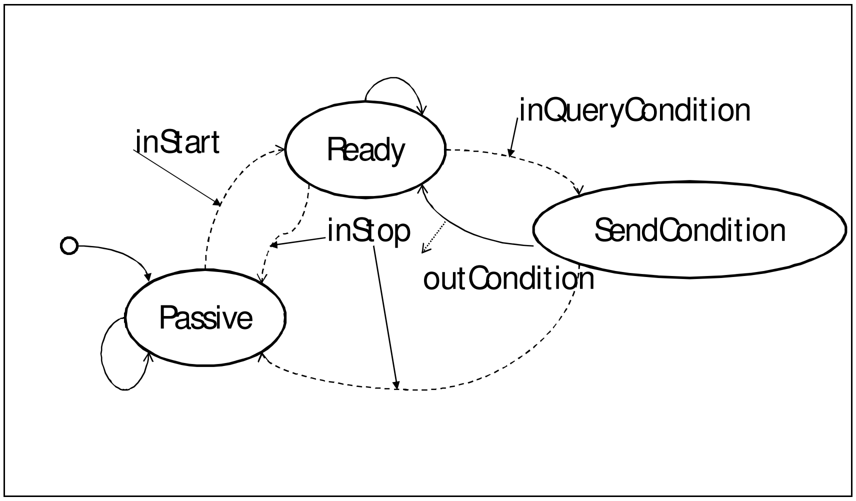


Figure 11 - ConditionChecker DEVS Diagram

The ConditionChecker model starts in state Passive. The input inStart is accepted in state Passive and puts the model in state Ready. The states Passive and

Ready have a time advance of infinity. The inputs inQueryCondition and inStop are accepted in state Ready. The input inQueryCondition puts the model in state SendCondition. The input inStop puts the model in state Passive. The state SendCondition has a time advance of zero. The state SendCondition outputs the condition on port outCondition and goes into state Ready.

Acceptor Model

X = {inStart, inPassFail}

Y = {outStop}

S = {Passive, Ready, Finished}

The Acceptor model is the analog to the TDCoordinator model in the original design. The purpose of the Acceptor model is to test the scenario for compliance, print test conditions to the console, and to do any inter-module coordination that is found to be necessary. This model is test case specific, and each test case has its own copy of this model. Currently, this model is created in a template-based fashion. The model described in this section is the template model.

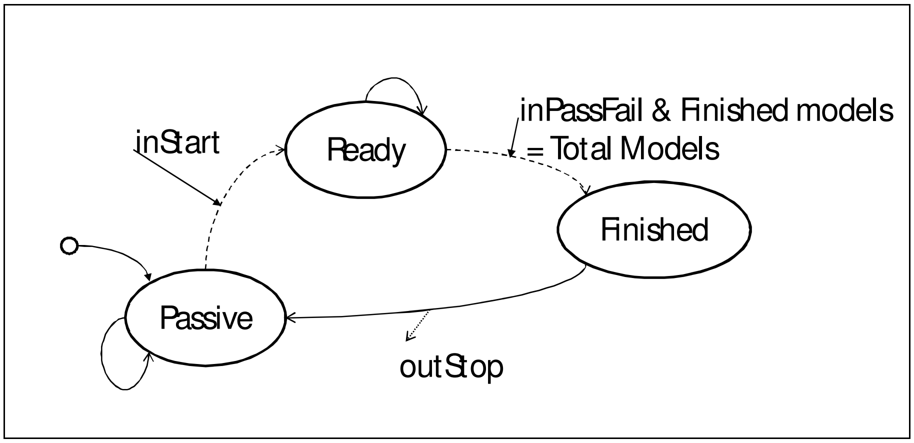


Figure 12 - Acceptor DEVS Diagram

The Acceptor model starts in state Passive. The input inStart is accepted in state Passive and puts the model in state Ready. The states Passive and Ready have a time advance of infinity. The input inPassFail is accepted in state Ready. If a scenario has ended, the input inPassFail puts the model in state Finished. The state Finished has a time advance of zero. It outputs outStop and goes into state Passive.

WAITRECEIVE MODEL

X = {inStart, inMessage, inCondition}

Y = {storeMessage, outQueryCondition, outPassFail}

S = {passive, storeMessage, QueryCondition, inWaitCondition, inWaitMessage, sendPassFail}

The waitReceive model is something of a reversion back to the origins of ATC-Gen, where the only behaviors modeled by test models were transmission and reception of messages. The waitReceive model contains the code necessary to test an incoming message for a) Message Type and b) Time Constraints. The model would also contain the code to save the received message by sending it to Store with a PlatformState value and a storage key.

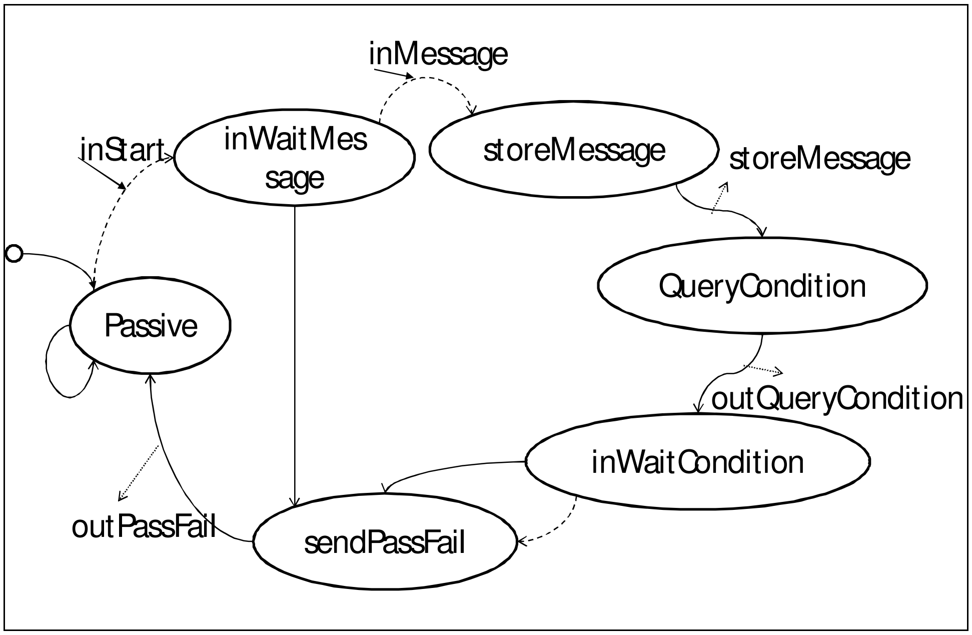


Figure 13 - WaitReceive DEVS Diagram

The waitReceive model starts in state passive. The input inStart is accepted in state passive and puts the model into state inWaitMessage. The state inWaitMessage has a time advance equal to a configuration parameter. The input inMessage is accepted in state inWaitMessage and puts the model in state storeMessage. If no input is received, state inWaitMessage goes to state sendPassFail. The state storeMessage has a time advance of zero. It outputs the message received on the storeMessage port and then goes into state QueryCondition. The state QueryCondition has a time advance of zero. It outputs a condition query on the outQueryCondition port and then goes into state inWaitCondition. The input inCondition is accepted in state inWaitCondition and puts the model in state sendPassFail. The state inWaitCondition has a time advance of 1.0. If no input is received, state inWaitCondition goes to state sendPassFail. The state sendPassFail has a time advance of zero. It outputs the condition of the model on the outPassFail port and then goes into state passive.

WAITNOTRECEIVE MODEL

X = {inStart, inMessage, inCondition}

Y = {storeMessage, outQueryCondition, outPassFail}

S = {passive, storeMessage, QueryCondition, inWaitCondition, inWaitMessage, sendPassFail}

The waitNotReceive Model is the opposite of the waitReceive Model. The waitReceive model tests if a certain type of message is received in a time period. The waitNotReceive model tests whether a) a certain type or b) any type, of message is not received in a time period. The waitReceive model would contain the code necessary to test an incoming message for a) Message Type and b) Time Constraints. However, these would be used as fail criterion instead of pass criterion. In terms of implementation, the only difference between the waitReceive model and the waitNotReceive model is the default value of the condition passed from the model when a message is not received. The waitNotReceive model has a default value of Pass rather than a default value of Fail.

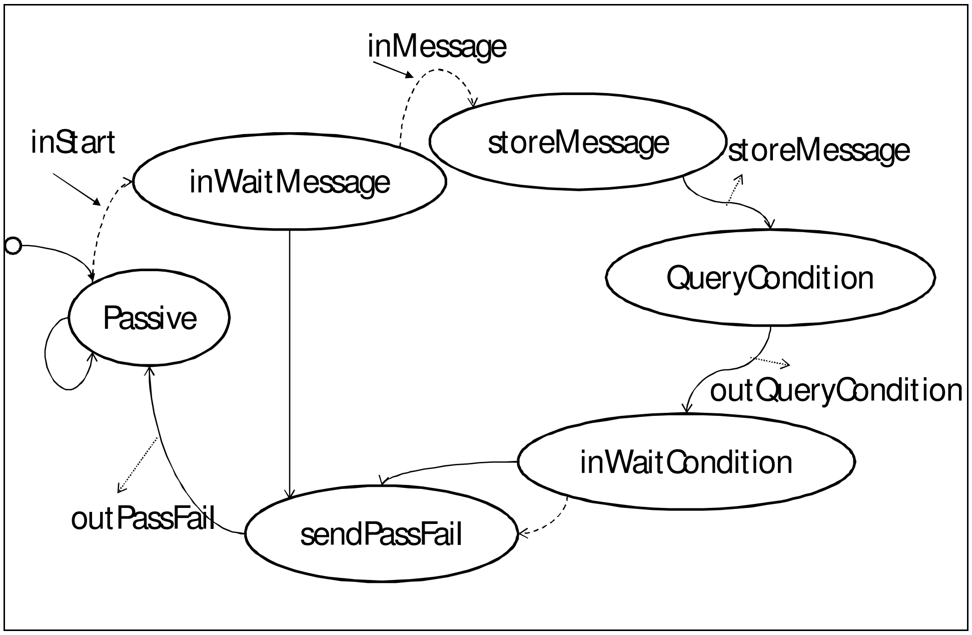


Figure 14 - waitNotReceive DEVS Diagram

The waitNotReceive model starts in state passive. The input inStart is accepted in state passive and puts the model into state inWaitMessage. The state inWaitMessage has a time advance equal to a configuration parameter. The input inMessage is accepted in state inWaitMessage and puts the model in state storeMessage. If no input is received, state inWaitMessage goes to state sendPassFail. The state storeMessage has a time advance of zero. It outputs the message received on the storeMessage port and then goes into state QueryCondition. The state QueryCondition has a time advance of zero. It outputs a condition query on the outQueryCondition port and then goes into state inWaitCondition. The input inCondition is accepted in state inWaitCondition and puts the model in state sendPassFail. The state inWaitCondition has a time advance of 1.0. If no input is received, state inWaitCondition goes to state sendPassFail. The state sendPassFail has a time advance of zero. It outputs the condition of the model on the outPassFail port and then goes into state passive.

HOLDSEND MODEL

X = {inStart, inStop, inMessage, inCondition}

Y = {outQueryCondition, outQueryMessage, outMessage, outPassFail}

S = {passive, inWaitMessage, inWaitCondition, sendPassFail, sendMessage, QueryCondition, QueryMessage}

The holdSend model models the transmission portion of the original ATC-Gen behavior. The holdSend requests conditions to be checked before it processes a message, and then requests Store to create a new message, either using data determined by the configuration of the system, or using data from a previously received message associated with the positional data of the message to be created.

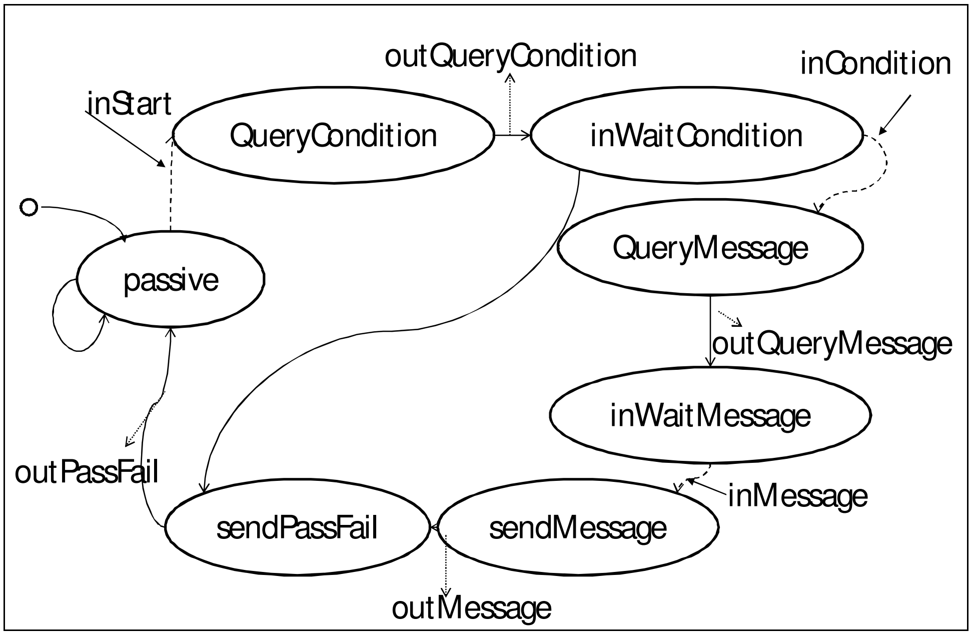


Figure 15 - holdSend DEVS Diagram

The holdSend model starts in state passive. The input inStart is accepted in state passive and puts the model in state QueryCondition. The state QueryCondition has a time advance equal to the hold time defined at configuration. It outputs a condition query on port outQueryCondition and goes into state inWaitCondition. The state inWaitCondition has a time advance of 1. The input inCondition is accepted in state inWaitCondition and puts the model into state QueryMessage. If no input is received, it goes into state sendPassFail. The state Query Message has a time advance of zero. It outputs a message query on port outQueryMessage and goes into state inWaitMessage. The state inWaitMessage has a time advance of 1. The input inMessage is accepted in state inWaitCondtion and puts the model into state sendMessage. If no input is received, it goes into state sendPassFail. The state sendMessage has a time advance of zero. It outputs a message on port outMessage and goes into state sendPassFail. The state sendPassFail has a time advance of zero. It outputs the condition of the model on port outPassFail and goes into state passive.

holdSendRepeat Model

X = {inStart, inStop, inMessage, inCondition}

Y = {outQueryCondition, outQueryMessage, outMessage, outPassFail}

S = {passive, sendMessage, FirstWait, RepeatTime, inWaitMessage, inWaitCondition, QueryMessage, QueryCondition, SendPassFail}

The holdSendRepeat model is used when a message needs to be sent at regular intervals. In Link 16, this model usually corresponds to track updates, which are sent periodically. The behavior that makes this model desirable can be modeled by a set of holdSend models in sequence, but the holdSendRepeat model can send a message an indeterminate amount of times, which makes it very useful. Also, because real-time testing behavior is often not exactly the same as the model behavior, being able to base the number of repetitions of a message on other factors than a predetermined number of models being coded is important.

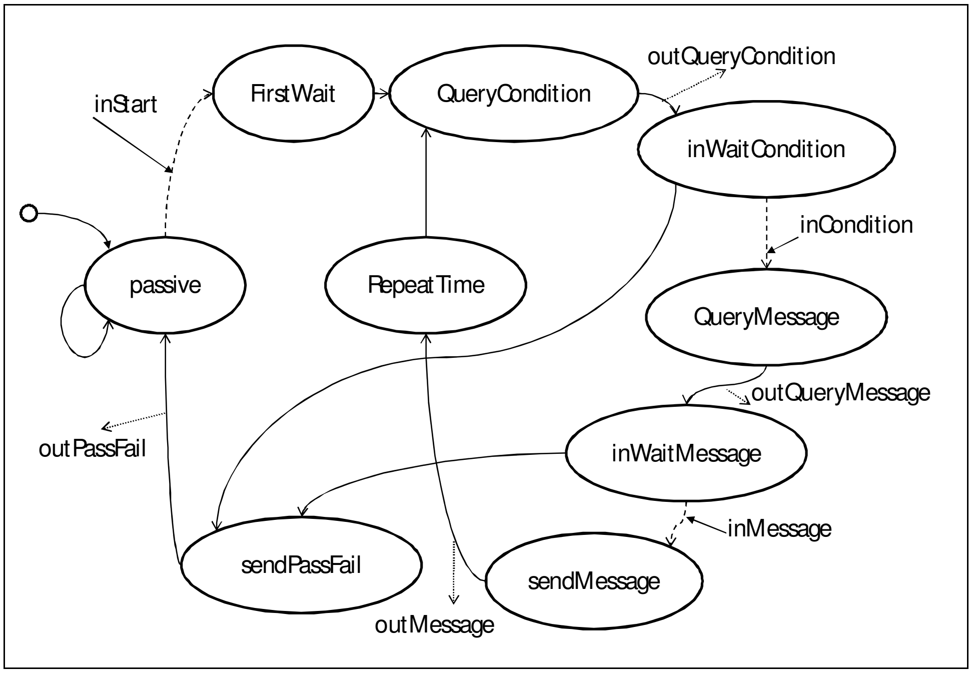


Figure 16 - holdSendRepeat DEVS Diagram

The holdSendRepeat model starts in state passive. The input inStart is accepted in state passive and puts the model in state FirstWait. The state FirstWait has a time advance equal to the first hold time defined at configuration. It goes into state QueryCondition when its time advance elapses. The state QueryCondition has a time advance of zero. It outputs a condition query on port outQueryCondition and goes into state inWaitCondition. The state inWaitCondition has a time advance of 1. The input inCondition is accepted in state inWaitCondition and puts the model into state QueryMessage. If no input is received, it goes into state sendPassFail. The state QueryMessage has a time advance of zero. It outputs a message query on port outQueryMessage and goes into state inWaitMessage. The state inWaitMessage has a time advance of 1. The output inMessage is accepted in state inWaitCondition and puts the model into state sendMessage. If no input is received, it goes into state sendPassFail. The state sendMessage has a time advance of zero. It outputs a message on port outMessage and goes into state RepeatTime. The state RepeatTime has a time advance equal to the second hold time defined at configuration. It goes into state QueryCondition when its time advance elapses. The state sendPassFail has a time advance of zero. It outputs the condition of the model on port outPassFail and goes into state passive.

TRACKSIMULATOR

X = {inStart, inStop, inQueryPosition}

Y = {outPosition}

S = {Passive, Ready, SendPosition}

The TrackSimulator model controls the timing for the track motion models, as well as allowing access to positional information. This model implements a system for generating synthetic tracks, described in [17]. It takes queries from the Store model and returns positional information. This is the equivalent of the extrapolator class in the Reactive mode, but the positional information involved is read from Script files, included in the system configuration. Each script file represents the positional information of one track, and the Simulator model has a map of tracks, each identified by a code. This code is referred to as the Internal ID, and is how the system associates positional information with holdSend type and waitReceive type models.

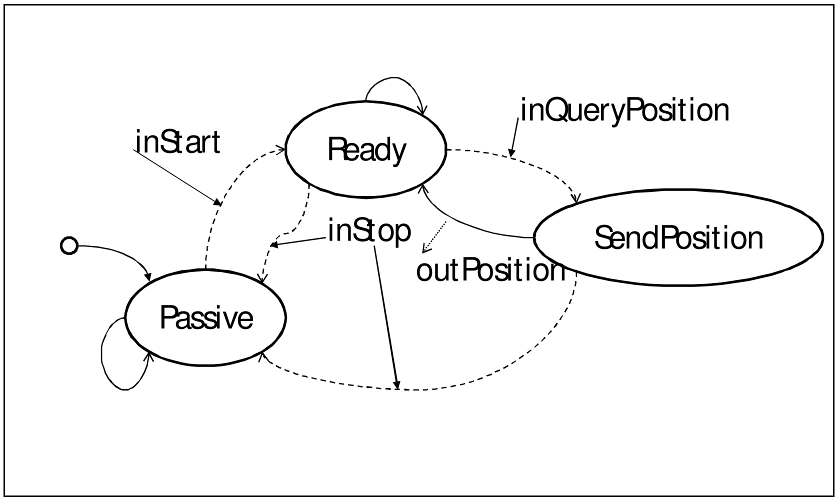


Figure 17 - TrackSimulator DEVS Diagram

The TrackSimulator model starts in state Passive. An input inStart is only accepted if the state is Passive, and puts the model in state Ready. The Passive and Ready states both have a time advance of infinity. The inputs inQueryPosition and inStop are allowed in state Ready. The input inStop puts the model into state Passive. The input inQueryPosition puts the model into state SendPosition. The state SendPosition has a time advance of zero. It outputs outPosition and goes into the state Ready.

STORE

X = {inStart, inMessage, inQueryMessage, inStop, inPosition}

Y = {outQueryPosition, outMessage}

S = {Passive, QueryPosition, InWaitPosition, SendMessage, Ready}

The Store model holds records of incoming and outgoing messages. The Store model is responsible for logging and creating the messages that are received by waitReceive type models or transmitted by holdSend type models. Logging is the process of collecting and organizing data for the purpose of data verification and records keeping.

For the purpose of logging, it is desirable to log the time and type of all messages associated with a given track. There needs to be a method of associating records of differing message type, but the same track, where a PlatformState message could be stored or retrieved using a key. Store implements this with a map, which uses the Internal ID associated with the track as a key. Each message received and each message sent is stored in the map, with a time tag noting when the message was sent or received. A public function exists in the ADEVS model to print the contents of this map to an output file.

In order to make Store able to handle multiple message queries, it has a queue that stores message queries. If this queue is not empty when the model goes from state SendMessage to state Ready, or state InWaitPosition to state Ready, it instead goes to state QueryPosition for the next message in the queue, and the query is deleted from the queue. However, for simplicity, this is left out of the diagram, as is the inMessage input.

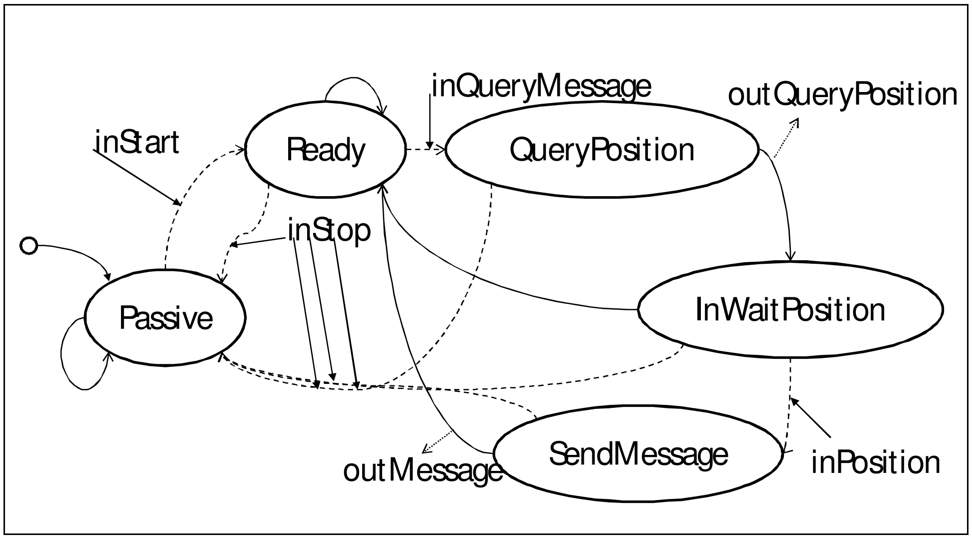


Figure 18 - Store DEVS Diagram

The Store message starts in state passive. An input inStart is only accepted if the state is Passive, and puts the model in state Ready. The Passive and Ready states both have a time advance of infinity. The inputs inMessage and inStop are allowed in all states but passive. The input inMessage does not cause any state changes, it only stores any messages received on that port into a message queue for logging. The input inStop puts the model into the passive state. The input inQueryMessage is accepted in state Ready, and puts the model into state QueryPosition. State QueryPosition has a time advance of zero, and it outputs outQueryPosition and goes into state InWaitPosition. State InWaitPosition has a time advance of 1.0. If no input is received, it goes into state Ready. The input inPosition is accepted in state InWaitPosition, and puts the model into state SendMessage. The state SendMessage outputs outMessage and goes into state Ready.

4.2.2 Improvements over Historical and First Reconfiguration

The final design of the test driver is in many ways superior to the previous implementations. The design is much more modular. It hails back to the original ATC-Gen, in that the major message handling code is in sequentially coupled holdSend and waitReceive models. It then expands upon this by abstracting the contents of the message away from the code that handles sending and receiving messages. By separating the contents from the handling, it allows the handling code to be reused without modification. In fact, the final code is such that only the ConditionChecker and Acceptor models require code changes in order to support a multitude of different test cases. The Store model requires only loading information differences. The Test Sequence primitive models require no coding changes, only configuration information.

The separation of contents from handling also has another benefit. The size of the primitive models is significantly smaller than the waitReceiveSendJSearch model or comparable models from the Reactive Mode ATC-Gen. The Reactive Mode test cases averaged around 250 lines of code, with the longest being 440 lines. The new test cases average around 200 lines of code, with the longest being 239 lines. In addition, there are only four primitive models, and all of the behaviors necessary can actually be modeled using only two of the four. In comparison, there is a primitive for each different behavior scenario to be tested in the Reactive Mode ATC-Gen, a total of 18. This meant that, as new scenarios were created to be tested, the size of the repository would grow linearly. With the new system, no growth of the repository is necessary, although a few new primitives may be introduced to simplify complicated behaviors.

The automation for the final design is based on a completely different concept than the original ATC-Gen automation. The original ATC-Gen automation relied on translating the document directly into XML rules, then finding paths through the rules and generating test cases from those sequences. The automation behind this design relies a little more heavily on Subject Matter Experts. The generation is still based on tracing paths through the rules of the standard; however these rules are combined with the analyst's familiarity with the behavior behind the paths, and the scenarios related with testing behavior. As such, a little more in-depth knowledge is necessary for the preloading portion of the automation process. However, the preloading process is much faster than the previous preloading process, and also includes much more scenario information than the previous process. It includes most of the data that was left out in the previous automation process, and can include more, given time and formalization. The new automation process is discussed more in detail in section 4.4 below.

4.2.3 Solutions for Problems

The test condition checking models solve many of the issues with the previous designs. They contain only the condition checking necessary for a given test case, which makes them smaller in coding size than the models suggested in the first redesign. They also have the ability to be formalized, which makes it possible to automate the creation of the ConditionChecker files. Also, since many conditions are similar, the code used to check a condition may be reused in many test cases.

The solution to the issues caused by the TDCoordinator model in the first redesign is split into two parts: the Store model and the Acceptor model. The Store model contains all of the message creation and logging code, which was in the TDCoordinator in the first redesign. The Acceptor model contains all the scenario testing code, which verifies that the order of messages received and sent is as expected, and handles pass/fail behavior. The Store model is a backbone model, which can be reused in every test case. The Acceptor model contains all test case specific behavior, and as such may be different for different test cases. Also, a new capability of FDDEVS automation allows for test scenario generation, and the Acceptor model can be generated based on this capability.

The automation portion of the new system also solves problems with the old automation process. The old automation portion was designed to create test cases without the intervention of analysts. As such, it did not take into account analysts' knowledge of how the system worked. This oversimplified the models and made testing of deeper behaviors impossible. Also, the XML representation of rules was faulty for many of the reasons mentioned in section 3.4.4. The new automation capability leverages analysts' knowledge by taking analyst-created scenarios, translating them in to a formalized FDDEVS description, and then creating the system. This process is discussed further in section 4.4.

Overall, this design seems to be successful in addressing the problems with the previous design. It adds the ability to automate testing of message conditions and scenario ordering that the original system lacked. It reduces the size of models while increasing their ability to be reused. Also, most importantly, it adds the formalism that the evolved version of ATC-Gen had lost, which is how it supports automation of test cases. The research of this thesis resulted in a system that successfully meets the requirements set out at the beginning of the project.

4.2.4 Description of Contributions

The system upon which this thesis was based takes advantage of the many benefits gained through abstracting code into separate portions, connected solely through interfaces. This thesis takes one of the portions, the model portion, and applies another layer of abstraction, thus allowing for reuse of broad categories of behavior. This abstraction makes automation of the system possible in ways that the earlier versions of ATC-Gen could never implement. The concept behind this abstraction is a new way of looking at testing message passing systems, called Message Interaction, Condition checking, Ordering (MICO). MICO allows a message passing system to be fully characterized in terms of the messages sent or received (Message Interaction), the message conditions required for test case pass or fail behavior (Condition checking), and the expected sequence of the messages, as it applies to system behavior (Ordering). Also, this thesis applies a new method of test case generation, developed in conjunction with Dr. Bernard Zeigler. This method generates ADEVS code, using sets of three keywords, referred to as triples. Each of these triples represents a certain behavior that commonly occurs in the course of a test scenario, such as the transmission or reception of a message. By coupling the code generated from these triples, multiple defined behaviors can be executed, allowing a full test scenario to be generated using a minimal amount of operator input. The final system is more fully defined, and the test cases are faster and easier to generate, than previous implementations of this research project.

The major contributions of this thesis are to introduce MICO as a methodology to describe message-passing systems for automation and testing, to develop a method using FDDEVS to automate the development of complex, distributed, real-systems, and to develop a new framework for a simulation-based message passing system standards conformance test system.

4.3 MICO Concept

4.3.1 MICO Description

MICO stands for Message Interactions, Condition (or Condition Checking), and Ordering. MICO does to the Model portion of the MSVC design pattern what the MSVC design pattern does to system design: It separates behaviors into separate entities with defined interfaces, so that each behavior can be implemented autonomously from every other behavior. This means that once a behavior is defined, it can be reused as many times as necessary to perform a task. This also means that, if a behavior changes, only the entity that changed needs to be recoded. This allows for a high percentage of reuse of code, as well as extensive modularity.

MICO is useful because it separates message interaction testing into three separate parts. This allows each of the parts to be formalized and automated. MICO is a new approach because it applies DEVS modeling and simulation to the characterization and testing of Message Passing systems, a problem that has not received as much attention [19].

4.3.1.1 Origin of MICO

MICO originated as a result of the failed attempt to redesign ATC-Gen in a modular way. The need for separation was inspired by the problems found when attempting to create the waitReceiveSend-derived classes and the TDCoordinator. Each attempted implementation had its own inherent problems, and each of these problems required its own separate solution.

The problem inherent in creating the waitReceiveSend-derived classes was that the classes attempted to include the checking of message-internal conditions. This required that the classes either contain all possible conditions that could be necessary to test, or that the classes include only the conditions necessary for a specific test case. The possible solutions for this problem included repositories for each message type, containing possible sets of conditions, or separate implementation of each class for each message and each condition necessary to be tested in a test case. Both of these solutions required repetition of large amounts of code that would need to be inserted into each implementation or member of a repository. The amount of time and code necessary to implement any of these options made the idea of including the condition checking in the model responsible for receiving messages unfeasible. This led to the concept of separating the condition checking portion of a test case from the reception or transmission of a message. The Condition Checking portion of MICO comes from these problems.

The problem inherent in the TDCoordinator was that it attempted to encompass too much of the behavior of the model. The TDCoordinator concept was first originated because the Reactive Mode models brought to light the need to verify pass or fail behavior for an entire scenario, rather than for a single message interaction. In the Reactive Mode implementation, the primitives had grown in size for two reasons. One reason was that conditions were added that required testing. The other reason was that the order of messages became a necessary part of the test, and if the correct messages were received in the wrong order, the scenario failed even if all of the message interaction models passed. This behavior was partially covered in the original ATC-Gen through the coupling, but sophisticated handling, such as console output of failure conditions or overall test scenario pass/fail behavior was unsupported, and unexpected real time testing behaviors could cause the system to incorrectly pass when it had actually failed, and vice versa. Thus, the system needed to include a portion to cover scenario behavior. The major duty of this portion was to test that messages in interactions were received in the correct order. The Ordering portion of MICO handles this duty.

4.3.1.2 Differences from Earlier ATC-Gen Concept

The first ATC-Gen concept characterized scenarios based on the transmission and reception of message. This characterization is equivalent to the Message Interaction specification of the MICO concept. The first ATC-Gen concept also captured some of the ordering behavior of a model through stimulating models through coupling. This corresponds to portions of the Message Ordering specification of MICO, although is far less complete. The coupling only allows for messages to depend on the reception or transmission of other messages. It does not capture the full behavior of a scenario, and does not allow for automated pass/fail behavior as the MICO concept does. The first ATC-Gen concept did not support condition checking of any sort. The Condition Checking specification of MICO fills this gap by specifying what conditions must be tested.

4.3.2 Message Interaction Specification

The Message Interaction Specification defines what messages a particular system is supposed to send or receive. Message interactions occur between two systems in a message-passing system. The Message Interaction Specification is from the perspective of one of the two systems. Each message sent by a system is mapped to a HoldSend model, and each message received by a system is mapped to a WaitReceive model.

The Message Interaction specification handles two things. First, it is responsible for characterizing what messages are sent or received by a participant in a given test scenario. Second, it is responsible for the timing associated with a particular message. This timing can be either universal, i.e., starting at the beginning of a test scenario, or local, i.e., starting after another message interaction completes.

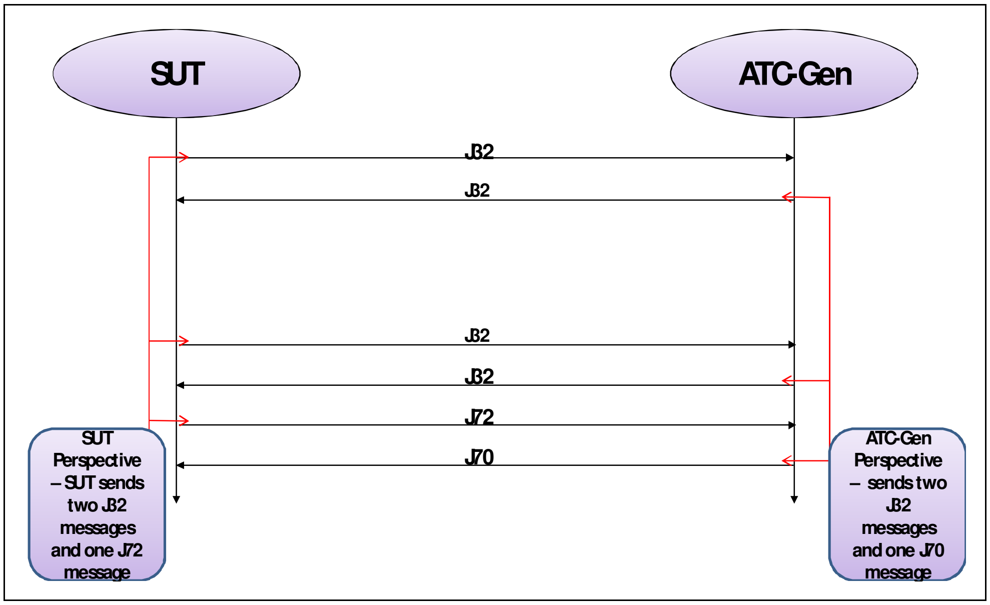

SUT
ATC-Gen
J32
J32
J32
J32
J72
J70
SUT Perspective – SUT sends two J32 messages and one J72 message
ATC-Gen Perspective – sends two J32 messages and one J70 message

Figure 19 - Message Interaction Example

The message interaction specification consists of a set of message interactions. A message interaction consists of a message, which, depending on the system, can be characterized by type, by port, or by some other criterion, an originator, a destination, associated expiration time, and time mode indicator. The expiration time indicates either when a message is scheduled to be sent, or how long it takes to send a message after a given stimulus, depending on the timing mode. The time mode indicator is 0 if the time mode is universal and 1 if the time mode is local. Figure 19 shows an example of a Message Interaction specification. The example is the RemoteTNDrop scenario used in earlier examples. As shown by the arrows, the specification from the perspective of the SUT is three messages sent, two J32 messages and one J72 message, and three messages received, two J32 messages and one J70 message. From the perspective of the ATC-Gen, the specification is the opposite. Because of this, only one specification is necessary, but the resulting test depends on the perspective. The timing specification is left out of the example.

4.3.3 Condition Specification

The Condition Specification allows for the testing of message contents and their affect on the behavior or state of the message-passing system. By separating the Conditions from the Message Interactions, the primitive models (HoldSend, WaitReceive) can be reused for different messages/test cases. The Messages can be separated from their internal data, removing the test-case specific code from the primitives. This reusability enables the test case generation to be automated quickly and with much less effort than a non-separated Test Model.

A Condition can be one of two things: Either a field in a sent/received message equals a predefined variable, or a precondition on which the message depends is satisfied. Message field testing requires prior knowledge of the required value or a method of making the required value available to the system at either configuration time or run time. Due to the nature of the standard being tested in this system, specific examples of field conditions cannot be included in this thesis. Preconditions usually come from the standard to which the model is being tested. For example, in order to perform correlation on a track, the track must have sufficient resolution. If this precondition is not met, the test case cannot proceed.

4.3.4 Ordering Specification

The Message Ordering Specification allows scenarios as a whole to be tested for pass/fail behavior. Message Ordering specifies if the sequence of behaviors occurs in the correct order, allowing for Pass/Fail behavior to be determined. Message Ordering also specifies what behavior should occur before and after a particular Message Interaction. In particular, a message may depend on a sequence of previous and following actions. The reception or transmission may change the state of a model such that another message that would otherwise be correct becomes incorrect. In contrast, Message Interaction specifies what the message is, who the message is from, and to whom the message is sent.

Message Ordering is implemented using two methods: Stimulus of test models via coupling and enforcement via the Acceptor model. The coupling method is inherent in Coupled DEVS models, and controls to which starting point a particular inStart point is connected. In order to start the system as a whole, two ways of starting were included in the system. The first way was a starter model which has no behavior but to output a start message on its outStart port. The second way was to receive a start message from the Interface, or the middleware protocol, allowing for a SUT to control when the test case starts. Enforcement using the Acceptor model is for the purpose of verifying scenario behavior for Pass or Fail. For scenarios where the Message Interaction specification is the same but the scenario is different, the only way to characterize the difference is with the Message Ordering specification. The Acceptor model is designed to receive notifications of completion from message handling primitives, including an indication of whether the associated message conditions passed or failed. From those notifications, the Acceptor model is responsible for verification of the scenario as a whole.

4.3.5 Characterizing a Message-Passing System

MICO is a general concept and can be applied to message passing systems other than the one used in this thesis. As an example, this section will characterize a simplistic Message Passing System in terms of the message passing behavior and the MICO specifications.

When characterizing a system, the most concrete step is to define the components. This is done in this system by employing the DEVS formalism to develop models and coupling. The next step is to define the set of messages. This can be done by message type, or in systems that have no defined message types, it can be done by port. Once the system has been defined on these levels, the MICO specification can be created.

The example system is as follows: Two routers are setting up a secure connection. The protocol for the negotiation begins with a greeting. If the greeting is accepted, a greeting reply is returned. If it is not, no message is sent back and the state times out. After the greeting reply is sent, a password message is sent. If the password is correct, a password acknowledgement is sent. If the password is incorrect, a Nack is sent. If a password acknowledgement is sent, the system is in a connected state and the system ends with a pass. If a nack is received or the state times out, the system ends with a fail.

The MICO system specification for the system starts with specifying the players. In this case, there is a service requestor and a service provider. The messages sent by the system are greeting, greeting reply, password, password acknowledgement, and nack. The first message has universal time and sends at time zero. The other messages are all assumed to have local time and one second of processing time. The Message Interaction specification is as shown in Figure 20. The messages are specified in a five-tuple, where the first element is the message type, the second is the transmitter, the third is the recipient, the fourth is the expiration time, and fifth and final is the time mode indicator. A time indicator of 0 indicates universal time and a time indicator of 1 indicates local time.

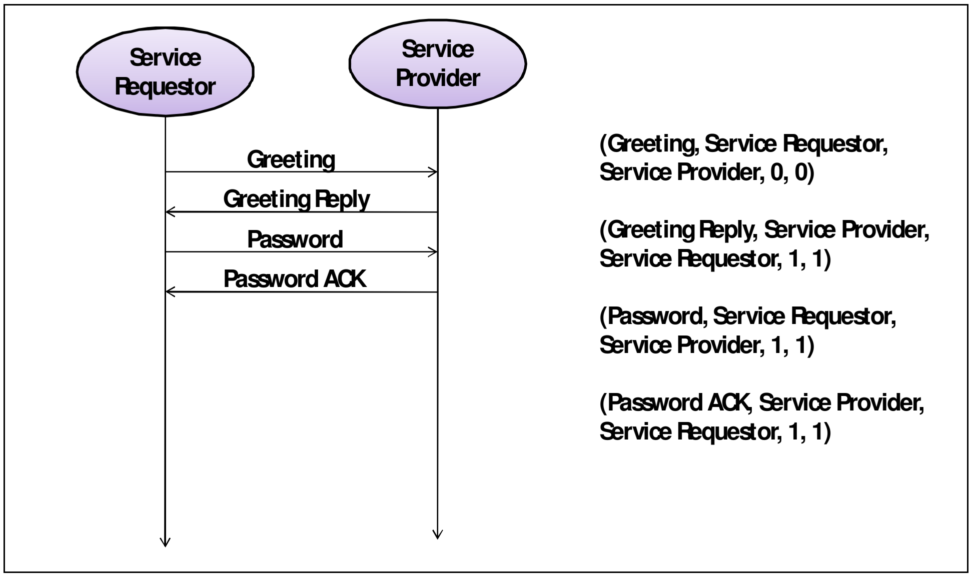

Service Requestor
Service Provider
Greeting
Greeting Reply
Password
Password ACK
(Greeting, Service Requestor, Service Provider, 0, 0)
(Greeting Reply, Service Provider, Service Requestor, 1, 1)
(Password, Service Requestor, Service Provider, 1, 1)
(Password ACK, Service Provider, Service Requestor, 1, 1)

Figure 20 - Message Interaction Specification

The condition specification is dependent on perspective. From the Service Provider side, the conditions specify when a received message meets conditions, and when to send a message. For this system, the conditions are:

1.) If the Greeting is formatted correctly and the name in the Greeting is in the authorized users list, the reception passes.

2.) If a passed Greeting is received, send a Greeting Reply

3.) If the Password received matches the name received in the last Greeting message, the reception passes

4.) If a passed Password is received, send a Password ACK

If the given conditions are not met, the reception or transmission fails. From the Service Requestor side, the messages themselves are not tested, but the following conditions are tested:

1.) If the system is started correctly, send a Greeting

2.) If a Greeting Reply is received, the reception passes

3.) If a Greeting Reply was received, send a Password

4.) If a Password ACK is received, the reception passes

Finally, the Message Ordering specification includes the scenario conditions. As mentioned above, the system passes if all the messages are received in the order shown in Figure 20. If any variations occur, the system as defined fails. However, not every system is as simple as this, and sometimes many variations may lead to passing behavior. The Message Ordering specification has 3 cases, 1 pass and 2 fail.

Case 1: Messages are sent and received as in Figure 20.

— (TX, Greeting, 0)

— (RX, Greeting Reply, 1)

— (TX, Password, 2)

— (RX, Password ACK, 3)

Case 2: Incorrect Password

— (TX, Greeting, 0)

— (RX, Greeting Reply, 1)

— (TX, Password, 2)

— (RX, NACK, 3) - Fail

Case 3: Invalid User Name

— (TX, Greeting, 0)

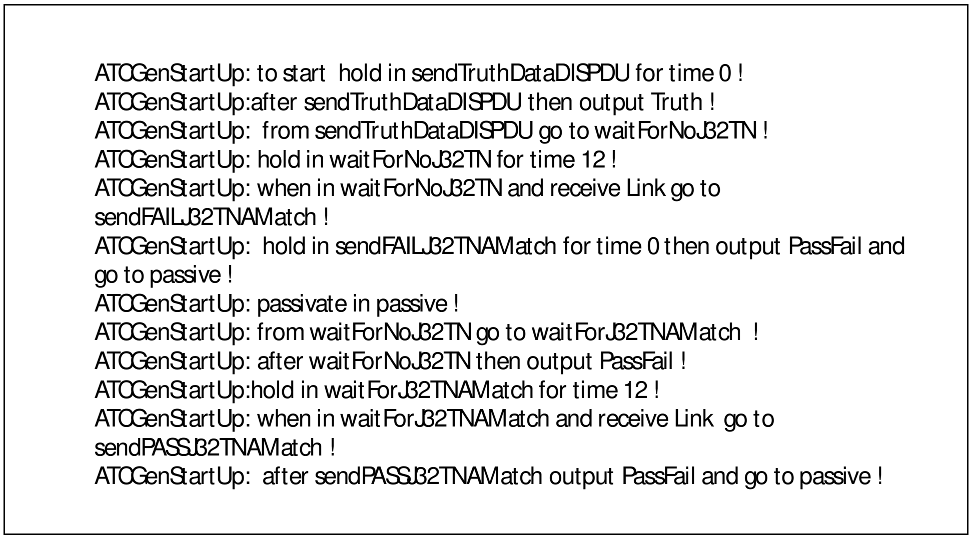
— (RX, NULL, 0) - Fail

The specification fully characterizes the example message system. All pass and fail behavior is defined, as well as the conditions for each transmission and reception. If this system were to be defined in the system in this thesis, the Message Interaction specification would translate into waitReceive and holdSend models, the Condition specification would translate into ConditionChecker models, and the Message Ordering specification would translate into Acceptor models.

4.4 FDDEVS Automation

4.4.1 FDDEVS description

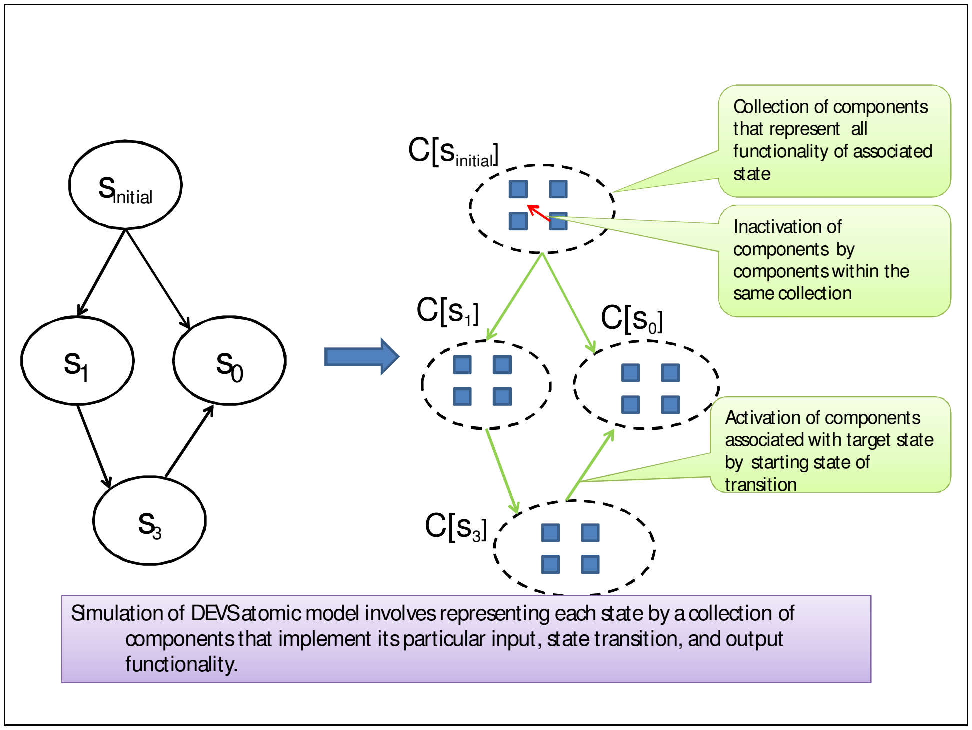
When automation of the test cases was first discussed, it was thought that the test cases could be broken into sections, each of which could be described using FDDEVS-constrained English. As discussed in Section 3.1.3, FDDEVS is a restricted version of classic DEVS. One of the positive aspects of FDDEVS is that the well-definedness of the specification has allowed for FDDEVS models to be automated very simply. At first, the automation was done using an XML specification, and then later, a constrained English specification was defined. An example of a constrained English FDDEVS specification is given below.



```
ATCGenStartUp: to start  hold in sendTruthDataDISPDU for time 0 !
ATCGenStartUp:after sendTruthDataDISPDU then output Truth !
ATCGenStartUp:  from sendTruthDataDISPDU go to waitForNoJ32TN !
ATCGenStartUp: hold in waitForNoJ32TN for time 12 !
ATCGenStartUp: when in waitForNoJ32TN and receive Link go to
sendFAILJ32TNAMatch !
ATCGenStartUp:  hold in sendFAILJ32TNAMatch for time 0 then output PassFail and
go to passive !
ATCGenStartUp: passivate in passive !
ATCGenStartUp: from waitForNoJ32TN go to waitForJ32TNAMatch  !
ATCGenStartUp: after waitForNoJ32TN then output PassFail !
ATCGenStartUp:hold in waitForJ32TNAMatch for time 12 !
ATCGenStartUp: when in waitForJ32TNAMatch and receive Link  go to
sendPASSJ32TNAMatch !
ATCGenStartUp:  after sendPASSJ32TNAMatch output PassFail and go to passive !
```

Figure 21 - FDDEVS Example

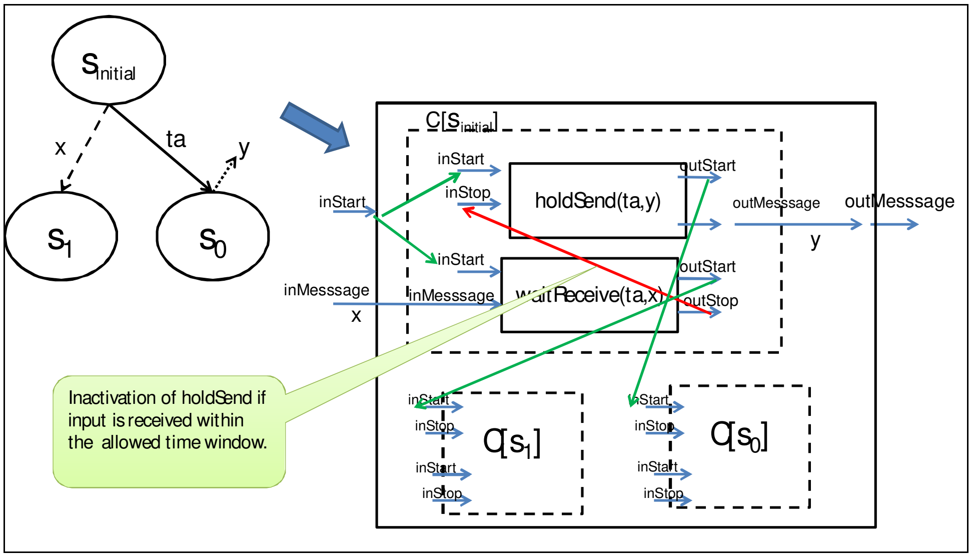
However, the FDDEVS models generated using the automation schemes are meant to be atomic models. While these atomic models can be coupled to form a system, what was desirable for this system was to map FDDEVS specifications into pre-existing models. In order to do this, the concept in Figure 22 was created.


$S_{initial}$
S_1
S_0
S_3
$C[s_{initial}]$
$C[s_1]$
$C[s_0]$
$C[s_3]$
Collection of components that represent all functionality of associated state
Inactivation of components by components within the same collection
Activation of components associated with target state by starting state of transition

Simulation of DEVS atomic model involves representing each state by a collection of components that implement its particular input, state transition, and output functionality.

Figure 22 - Mapping Concept

As the figure shows, the states in an FDDEVS were mapped to a set of message primitive models. Each of the states has functions that map the state transition: the delta int, delta ext, and delta conf. As shown in Figure 23, a mapping was developed that modeled a state using a waitReceive model for each message received in the delta ext state transition, and a holdSend for the delta int transition. This mapping required additional ports be added to the primitives, namely inStop, outStart, and outStop. However, this mapping assumed the simplistic version of the waitReceive primitive from the original ATC-Gen. Due to the actual final implementation of the waitReceive model, the use of an additional holdSend model and the addition of the said ports was unnecessary. The behavior was instead handled by the outPassFail message, which handled state transitions by outputting either a pass message or a fail message.



Inactivation of holdSend if input is received within the allowed time window.

Figure 23 – Mapping a State to holdSend and waitReceive Primitives

In the final model, an expected external input was modeled by a waitReceive primitive, and an expected output of a message to the interface was modeled by a holdSend primitive. The stimulus behavior that moves between states is handled using the outPassFail port. These allow the system to be implemented as described in the system overview section above, but to be created automatically using FDDEVS constrained English specifications.

As more test cases were characterized, sections of similar behavior were found to occur regularly. This led to a new idea to speed up the characterization of test cases by mapping the FDDEVS-constrained English descriptions of portions to macros with three portions. These were called triples throughout the test case characterization portion, and are described below.

4.4.2 Test Case Creation using Macros

In order to make the characterization of test cases as easy as possible on the analysts, similar portions of behavior were classified and turned into macros. These macros were named according to their behavior, and were implemented as three arguments. The first argument was the name of the macro. The second argument was the desired name of the generated model. The third argument was the message to be handled by the given portion. The remainder of this section is an example of the test case generation process.

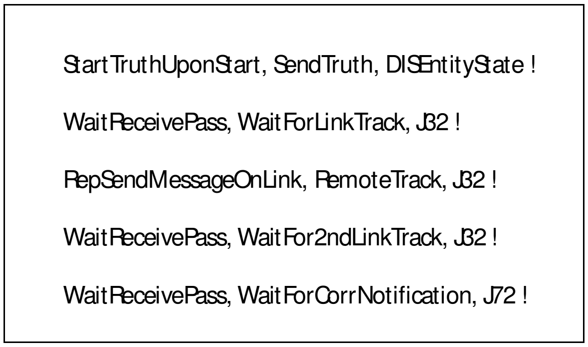
4.4.3 Test Case Generation

The scenario generated in this example is called X. It is as described. The first step in generating a scenario is to characterize the scenario into steps. This is done by a Subject Matter Expert (SME) analyst. The output of this step is a numbered list of steps that define the scenario. An example of this is shown in Figure 24.

0. Configure SUT for Mil Std 6016C chg1 default rule set.
1. Inject air truth track TN A into the SUT.[DIS, J3.2a]
Observe the SUT produces a L16 track TN A for the truth Track.
2. Send a remote TDL L16 track TN B into the SUT with the exact location and essential information (but higher TN) as TN A, such that it will correlate.
Observe the correlation and drop track of the remote track TN B. Verify number of correlation tests counted by ATC-Gen and in the SUT log file (LDDM). [J3.2b, J7.2, J3.2b, J 7.0 (0)]
3. Drop all tracks. [DIS, J7.0 (0)]

Figure 24 - Test Case Specification Example

Next, the characterization is translated into triples. These triples are the macros discussed earlier, and represent a particular behavior as captured in an FDDEVS constrained English representation. Examples of these triples are: SendTruthUponStart, SendMessageOnLink, SendTwice, WaitNotReceivePass, and WaitReceivePass. In the example, the translation to triples is as shown in Figure 25.



StartTruthUponStart, SendTruth, DISEntityState !

WaitReceivePass, WaitForLinkTrack, J32 !

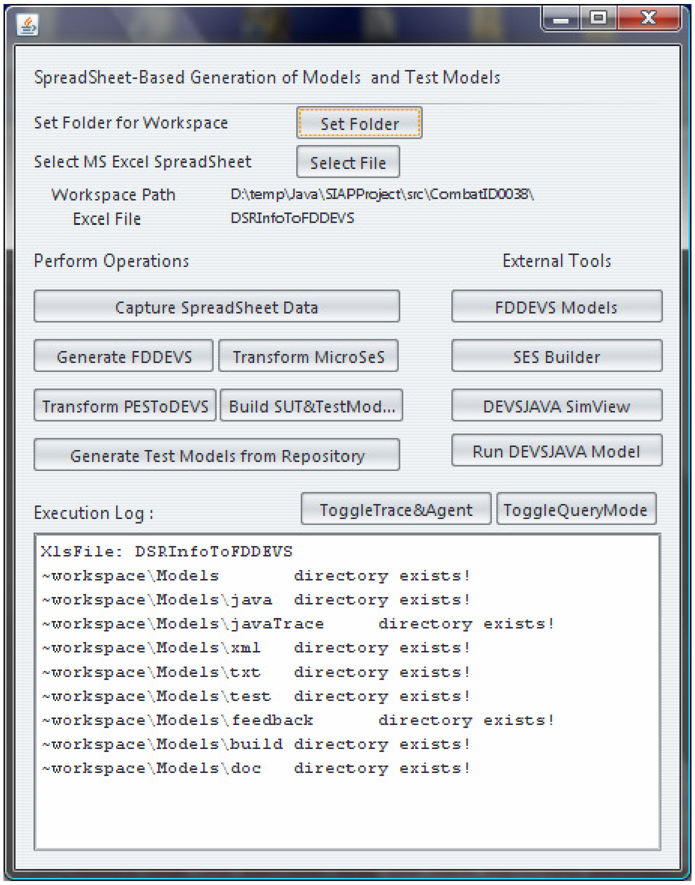
RepSendMessageOnLink, RemoteTrack, J32 !

WaitReceivePass, WaitFor2ndLinkTrack, J32 !

WaitReceivePass, WaitForCorrNotification, J72 !

Figure 25 – Triple (Macro) Specification Example

The triples are input into a premade Excel spreadsheet. This spreadsheet is put into a directory along with a few pre-generated files which serve to direct the automation software in its creation of files. Then, the generation software is started. The FDDEVS generation suite has a few different parts. It has a portion that generates models in a Java implementation of DEVS called DEVSJAVA. It has a portion that generates models in ADEVS. It also has a portion that generates model hierarchies from SES specifications. The first step of generation is to start the Java code generator. This Graphical User Interface (GUI) is shown in Figure 26.



Workspace Path D:\temp\Java\SIAPProject\src\CombatID0038\
Excel File DSRInfoToFDDEVS

```
XlsFile: DSRInfoToFDDEVS
~workspace\Models       directory exists!
~workspace\Models\java  directory exists!
~workspace\Models\javaTrace      directory exists!
~workspace\Models\xml   directory exists!
~workspace\Models\txt   directory exists!
~workspace\Models\test  directory exists!
~workspace\Models\feedback       directory exists!
~workspace\Models\build directory exists!
~workspace\Models\doc   directory exists!
```

Figure 26 - FDDEVS Java Generation GUI

The Java generating GUI has six operation buttons, two file/folder selection buttons, four tool buttons, and two log-related buttons. After running the GUI, the first step is to select the folder where the models will be generated, which is the folder the spreadsheet mentioned earlier is in, and to select the spreadsheet. Next, a series of buttons are pressed to capture the spreadsheet data, create FDDEVS definitions of models from the macros, and create models from the FDDEVS constrained English definitions.

The SES tool is then used to create a Pruned Entity Structure (PES) which is a pruned implementation of a generic SES. The pre-generated files take care of the creation of the PES, and the only steps necessary to create it from the tool is to open a generated file, hit a button to process the file, create the PES, update the PES, and then close the SES tool. Figure 27 shows the SES GUI. Figure 28 shows the PES created for the example in the SES GUI.

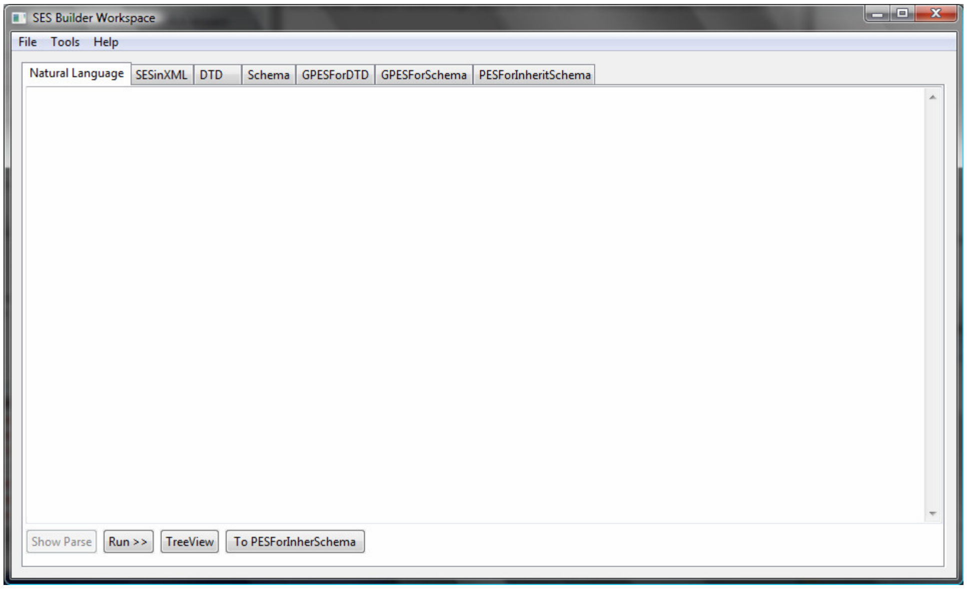
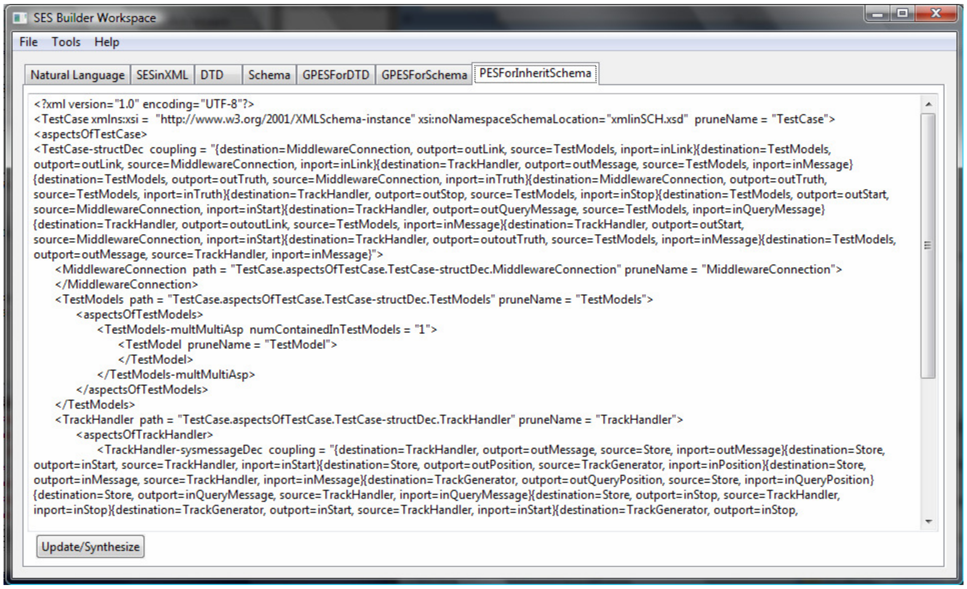


Figure 27 - SES Builder GUI

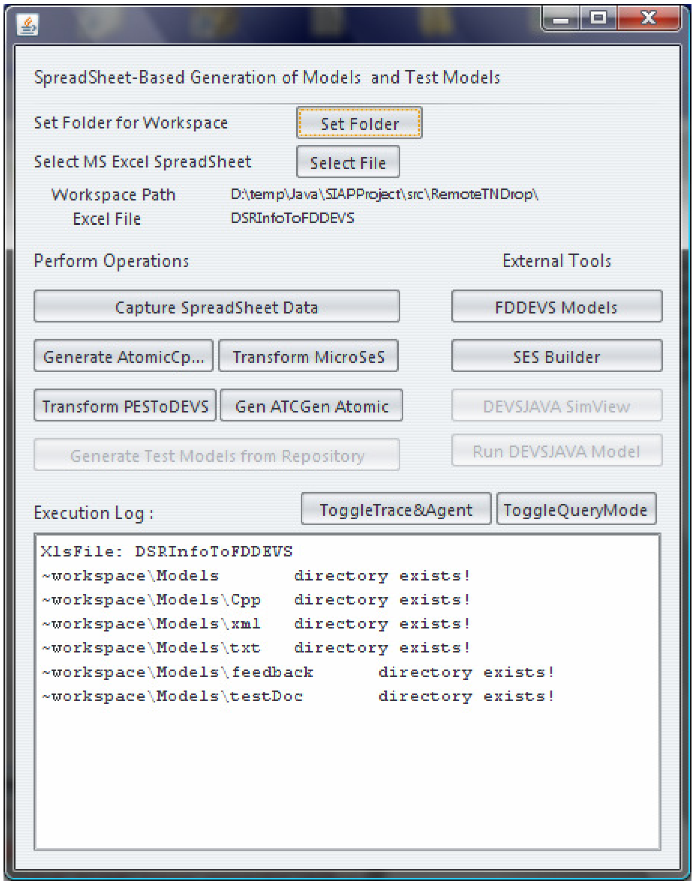


```
<?xml version="1.0" encoding="UTF-8"?>
<TestCase xmlns:xsi =  "http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="xmlinSCH.xsd"  pruneName = "TestCase">
<aspectsOfTestCase>
<TestCase-structDec  coupling = "{destination=MiddlewareConnection, outport=outLink, source=TestModels, inport=inLink}{destination=TestModels,
outport=outLink, source=MiddlewareConnection, inport=inLink}{destination=TrackHandler, outport=outMessage, source=TestModels, inport=inMessage}
{destination=TestModels, outport=outTruth, source=MiddlewareConnection, inport=inTruth}{destination=MiddlewareConnection, outport=outTruth,
source=TestModels, inport=inTruth}{destination=TrackHandler, outport=outStop, source=TestModels, inport=inStop}{destination=TestModels, outport=outStart,
source=MiddlewareConnection, inport=inStart}{destination=TrackHandler, outport=outQueryMessage, source=TestModels, inport=inQueryMessage}
{destination=TrackHandler, outport=outoutLink, source=TestModels, inport=inMessage}{destination=TrackHandler, outport=outStart,
source=MiddlewareConnection, inport=inStart}{destination=TrackHandler, outport=outoutTruth, source=TestModels, inport=inMessage}{destination=TestModels,
outport=outMessage, source=TrackHandler, inport=inMessage}">
    <MiddlewareConnection  path = "TestCase.aspectsOfTestCase.TestCase-structDec.MiddlewareConnection" pruneName = "MiddlewareConnection">
    </MiddlewareConnection>
    <TestModels  path = "TestCase.aspectsOfTestCase.TestCase-structDec.TestModels" pruneName = "TestModels">
        <aspectsOfTestModels>
            <TestModels-multMultiAsp  numContainedInTestModels = "1">
                <TestModel  pruneName = "TestModel">
                </TestModel>
            </TestModels-multMultiAsp>
        </aspectsOfTestModels>
    </TestModels>
    <TrackHandler  path = "TestCase.aspectsOfTestCase.TestCase-structDec.TrackHandler" pruneName = "TrackHandler">
        <aspectsOfTrackHandler>
            <TrackHandler-sysmessageDec  coupling = "{destination=TrackHandler, outport=outMessage, source=Store, inport=outMessage}{destination=Store,
outport=inStart, source=TrackHandler, inport=inStart}{destination=Store, outport=outPosition, source=TrackGenerator, inport=inPosition}{destination=Store,
outport=inMessage, source=TrackHandler, inport=inMessage}{destination=TrackGenerator, outport=outQueryPosition, source=Store, inport=inQueryPosition}
{destination=Store, outport=inQueryMessage, source=TrackHandler, inport=inQueryMessage}{destination=Store, outport=inStop, source=TrackHandler,
inport=inStop}{destination=TrackGenerator, outport=inStart, source=TrackHandler, inport=inStart}{destination=TrackGenerator, outport=inStop,
```

Figure 28 - SES Builder PES Window

After the PES is created, the Java GUI is reopened and the hierarchy of models that form structure of the system are generated from the PES. After this step, the models are created in a Java-based simulation language known as DEVSJAVA. The next step is to add the coupling between the sections created through the macros. After the coupling is added, the system is specified. At this point, the system can be viewed and the message interactions simulated in the Java environment. Examples of the visualization tools are shown in the model hierarchy section below.

The next part of the generation is to generate the C++ version of the models. The C++ generation GUI is shown in Figure 29. It has many of the same buttons as the Java generation GUI, and works in much the same way. It also stores the captured spreadsheet data in the same place, so once data is captured by one, it can be used by both. The generation of the C++ files is performed by hitting a series of buttons, which generate the Atomic C++ models, create the model hierarchy which forms the structure, and create the coupled models formed using the mapping concept discussed in Section 4.4.1.



Workspace Path D:\temp\Java\SIAPProject\src\RemoteTNDrop\
Excel File DSRInfoToFDDEVS

```
XlsFile: DSRInfoToFDDEVS
~workspace\Models        directory exists!
~workspace\Models\Cpp    directory exists!
~workspace\Models\xml    directory exists!
~workspace\Models\txt    directory exists!
~workspace\Models\feedback       directory exists!
~workspace\Models\testDoc        directory exists!
```

Figure 29 - C++ Generation GUI

After the files are generated, they are put into a Microsoft Visual Studio 2008 project. This project is checked by an analyst to verify correctness. At this point, the ConditionChecker and Acceptor models are modified by hand. This is because the ConditionChecker and Acceptor models have not yet been characterized to the point of automation. This is a future direction of this research project. Finally, the project is built and the configuration files are created. The result is a testable executable that can be run against a SUT.

4.5 Final System Configuration

4.5.1 Overview of Descriptions and System Hierarchy

In the system implementation, the code is broken into two projects. One project contains the backbone of the code and all of the communications mechanisms. This part is described in the System Description section below. The other project contains a repository of Test Cases, each with its test case specific code and each built into its own executable. An example of one of these test models, based on the generated example from section 4.4.3, is described in the Test Model Description section below. The system is connected through coupled ports. The Test Model Description section is contained in a portion of the System Description, as mentioned below. The system as a whole is described according to its parts.

4.5.2 System Description and Hierarchy

The system as a whole consists of the different abstractions of the MVSC design pattern mentioned in section 3.4.1 above. This section describes the portions of the system and how they correspond to the concepts mentioned earlier in the paper.

The Model portion of the design pattern consists of the components described in section 4.2. The model portion has been described in detail, and so a detailed description will be left out of this section. The Model portion connects to the Simulator through a simulator interface, and connects to the View and Controller through models called Activities, which implement the protocol interfaces.

The Simulator in this system is a simulated real-time ADEVS simulator. This means that it runs on discrete time, but schedules events using the system clock of the computer it is running on. It is implemented hierarchically, by adding the coupled models to a tree structure and simulating atomic models through one interface, and coordinating between coupled models through another interface. This is a hierarchical implementation of a DEVS simulator, as discussed in [8].

The View and Control portions are implemented in models called Activities. Activities are interfaces between a discrete-time modeling and simulation environment and a real-time environment, such as the High Level Architecture (HLA) simulation environment. In this system, three protocols are implemented, SIMPLE, HLA, and DIS. The view portion is implemented as console output that allows the tester to view what messages are sent and received. The controllers are in charge of controlling the protocol behaviors. The Activity models mediate passing messages between the simulation environment and the protocol environment.

An overview of the hierarchy of the system is shown in Figure 30. The LegacyActivityManager and RTSim_Activity are ADEVS 2.1 wrappers that facilitate communication between the protocol classes, which are the classes derived from Activity, and the Test Case classes, which are described in the next section. Messages are passed between the LegacyActivityManager class and the RTSim_Activity class, and also between the protocol classes and the Translator, which is a separate dll containing all of the protocol implementation code. This system is the backbone upon which the test cases run.

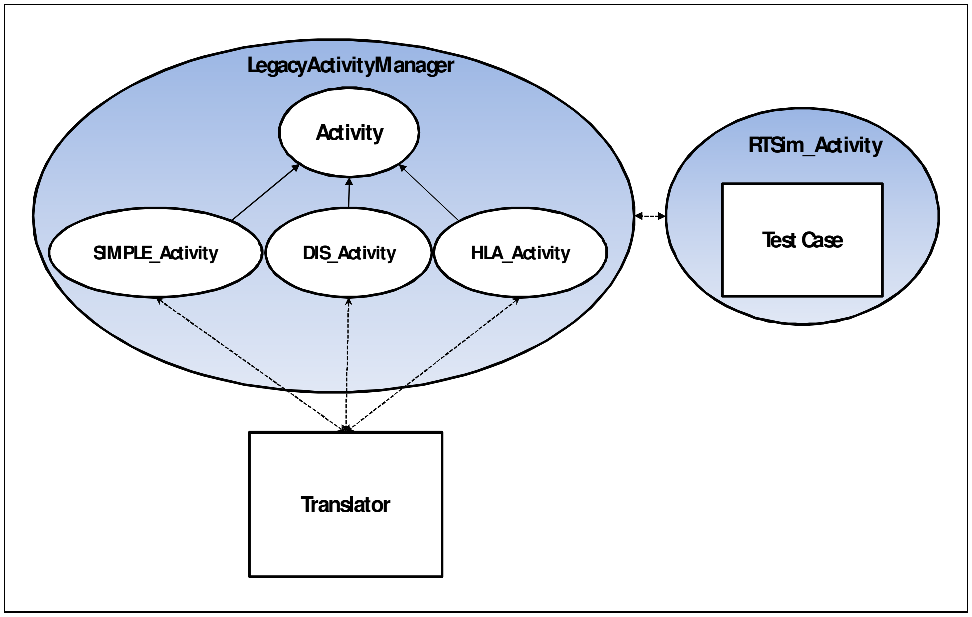


Figure 30 - System Hierarchy

4.5.3 Test Model Description and Model Hierarchy

The test model contains the backbone model and the test cases. Test cases are made up of the MICO-specified portions, the hierarchical models, and the main model, which is responsible for connecting all of the pieces and creating the executable. The backbone model is made up of the Store model and the TrackSimulator model, which are coupled into a model called the TrackHandler, and the interface to the Activity classes, which is coupled into a model known as MiddlewareConnection. Figure 31 shows the structure of the MiddlewareConnection coupled model.

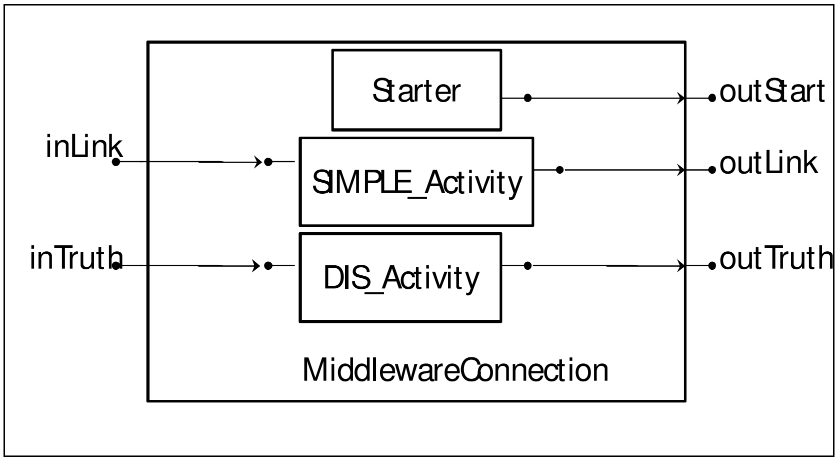


Figure 31 - MiddlewareConnection Coupled Model

Figure 32 shows the structure of the TrackHandler coupled model. The diagram is an example of the output of the DEVSJAVA viewer. The TrackSimulator model is named TrackGenerator in the Java version of the code.

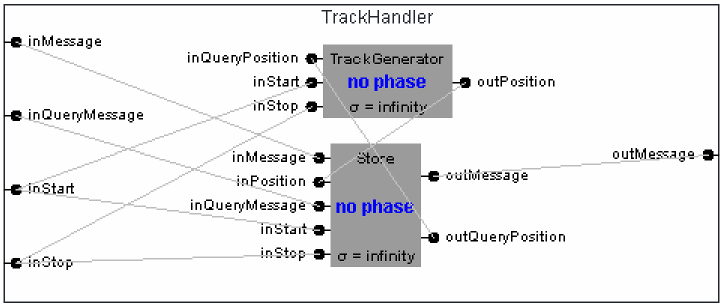


Figure 32 - TrackHandler Coupled Model

Figure 33 is the equivalent of the Test Case box in Figure 30 above. It is the top-level view of the structure of a test case. It includes the TrackHandler model shown in Figure 32, the MiddlewareConnection model shown in Figure 31, and the TestModels model. The TestModels model is the test case specific portion of the code, and as such is different for each given scenario.

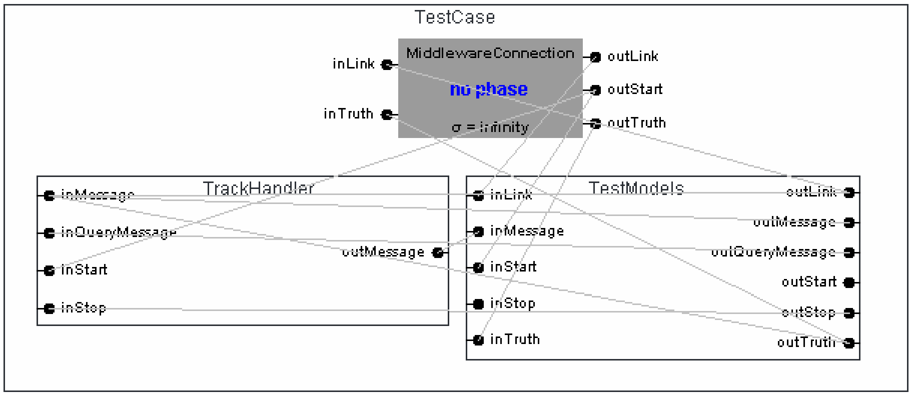


Figure 33 - Test Case Structure

Figure 34 shows an example of the contents of the TestModels model. This example was generated from the FDDEVS macro specification in Figure 25. It shows each macro generated into a portion of a test case.

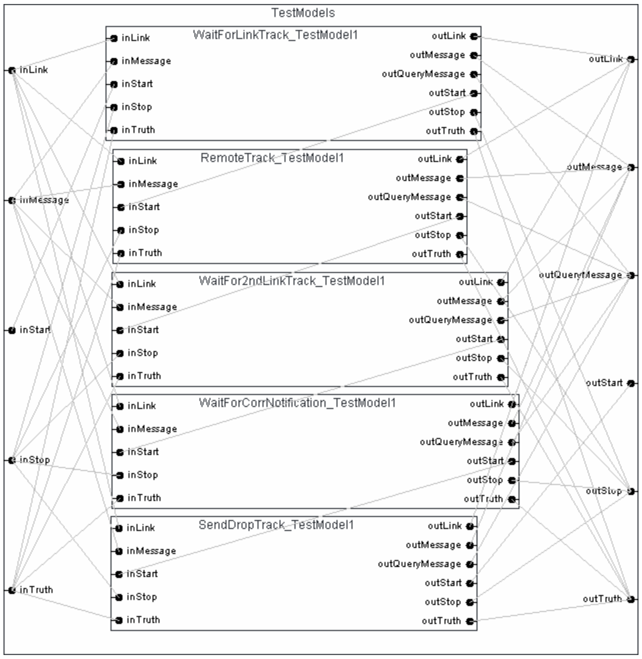


Figure 34 - Structure of Generated Example Test Case

Figure 35 shows the structure of one of the portions of the models inside the TestModels model. This is the level at which the MICO specification of the test case comes into play. Each portion of a test case has an associated Message Logic, which contains the Message Interaction code, a ConditionChecker, which contains the Condition code, and an Acceptor, which contains the Message Ordering code. Altogether, these hierarchical models make up the structure of the system.

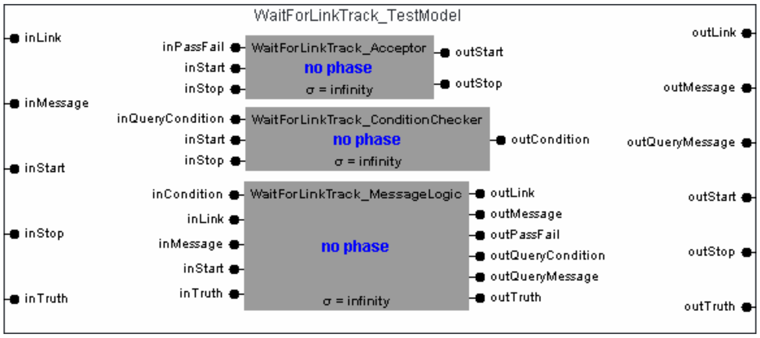


Figure 35 - Structure of WaitForLinkTrack Test Case Portion

5 Results

5.1 System Design Results

The system in this thesis was designed to resolve the issues with previous versions. These issues and the resulting resolutions are discussed in the sections below.

5.1.1 Handling Conditions and Message Contents

The original ATC-Gen did not test message contents. The only conditions tested in the original ATC-Gen were reception or transmission of messages. The expanded capabilities of the Reactive Mode allowed message contents and other conditions to be tested, but did not allow for them to be generated. All message content tests had to be coded into large scenario-testing models.

This system makes generation of condition checking a reasonable possibility. It was not implemented at the time of this thesis, due to the complexity and number of conditions to be formalized. However, the mechanisms exist to make condition checking automated, and the condition checking in this system was implemented using templates, which lowers the amount of code to be created by hand.

5.1.2 Testing Scenarios Instead of Messages

The original ATC-Gen tested the transmission and reception of messages. This required that a test case be characterized in terms of the messages passed and received. The automation process developed to generate test cases left out important behavioral information.

The new system characterizes testing not on the level of the messages sent and received, but on the level of the scenario. This takes into account the state changes that reception or transmission of a particular message can have on the system as a whole. This allows scenarios to pass or fail based on the overall behavior, rather than the reception or transmission of a single message. This version more correctly models the behavior of a message passing scenario being modeled.

5.1.3 Formalized Generation of Test Cases

The original ATC-Gen had a formalized generation process. However, as the system evolved, this process was lost and replaced by a copy-paste recreation method.

This evolutionary process left the Automated Test Case Generator without an automated generation capability.

The system developed in this thesis combines the original formal generation process with new concepts in order to generate test cases that have more depth of condition and ordering testing than the original ATC-Gen test cases. The final generation process developed in this thesis is also easier and faster than the original generation process.

5.1.4 Modularity and Reuse of Code, and Model Size Issues

The major motivating factor of redesigning the ATC-Gen system was to create a system that was modular, reused large portions of code, and reduced the size of the model primitives. This system does all of these things. The test cases are created using small, pre-existing message primitives. The portions of code corresponding to creation of messages are reused in every system by including the Store module. There no longer exist any model primitives that require copying and pasting hundreds of lines of code to function. This system successfully satisfies the motivating requirements behind this project.

5.2 Automation and Test Case Generation

It took approximately 3 months to generate and perform the final calibration step to complete all of the test cases. In all, 74 test cases were generated using the FDDEVS triples methodology. In addition, 23 test cases were created from existing test cases and 5 test cases were created by hand. In total, 102 test cases were created and run against the SUT.

The amount of time necessary to generate a new test case completely by hand proved the superiority of this test case creation system against the previous Reactive Mode. On average, a new Reactive Mode test case takes 1-2 days of work to complete. On average, a new system test case required 2-6 hours. In the worst case, it is still a reduction of more than half.

The generation process itself took approximately a half an hour per test case. In addition, the preloading and post-processing stages took between 3 hours per test case and 12 hours. The worst case time of 12 hours was mostly due to analyst miscommunication leading to redesign of test cases, and even the worst case was only one and a half days, which is less than the upper limit of the average time of the Reactive Mode. Overall, redesigning the system for ease of test case creation and fast test case generation was successful.

5.3 Testing Results

Due to the classification levels of this system, the experimental results cannot be shown. However, without discussing proprietary information, the following results can be mentioned.

The SUT that the system developed in this thesis was run against was a C2 program developed by the Single Integrated Air Picture (SIAP) Joint Program Office (JPO). The testing took two weeks, and a total of 102 tests were run. Final test case results were: 85 passed, 15 failed, and 2 partially passed. The testing was successful, and detected a number of issues that cannot be discussed in this paper.

6 Conclusion

6.1 Summary and Context

The ATC-Gen research project had many incarnations prior to this thesis. Each was useful at the time it was created. However, the uncontrolled evolution from a formal but limited system into a broader but unformalized system made the further development of the system problematic. Solutions were needed for testing message contents and order. The system designed in this thesis introduced the MICO concept as a method to characterize message passing systems. This concept helped formalize methods to generate test cases with message content testing and scenario ordering testing. The FDDEVS formalism was used to create tools that allowed the generation of test cases. These test cases were developed faster and with less effort than the test cases in previous implementations of ATC-Gen, while allowing for the generation of more in-depth testing of message contents and scenario ordering. The final system was generated in less time than previous incarnations. The system was verified using the TIAC blah and then tested against a SUT. Over 100 test cases were generated and run against a SUT. Final test case results were: 85 passed, 15 failed, and 2 partially passed. Overall, this design is an improvement over previous incarnations of the ATC-Gen research project and the research project successfully fulfilled the requirements.

6.2 Contributions and Conclusions

The major contributions of this thesis are the MICO concept and the FDDEVS generation environment. The MICO system applies DEVS formalism to Message Passing systems, and gives a basis for formalism of testing. It allows characterization of Message Passing systems so that test case generation can be abstracted into separate, well-defined portions. It allows further characterization which allows for future automation. The final system developed based on the MICO concept is more easily and more fully automated than any incarnation of the ATC-Gen research project that existed before.

The FDDEVS automation software allows for the fast, easy characterization of scenarios, and the fast, formal generation process of test cases. Using triples, test cases can be characterized faster and with less analyst effort than previous systems. Using the generation software, the principles of FDDEVS, SES, and MICO are applied to create a system that is well-defined and easily understandable. The final product gives the ability to generate full test cases in a matter of hours, where test case generation used to take days with previous versions of the ATC-Gen software.

The system developed in this thesis successfully met the requirements defined at the start of the research project. It introduced the MICO concept as well as the FDDEVS generation software. It tests message conditions and contents in a more formal manner than previous implementations. It tests for scenario behavior without excessively long primitive models. It implements a formalized test case generation tool. It consists of modular, reusable code, and is implemented with reasonably sized models. The system implements all of the defined requirements.

The test case automation methodology combines DEVS and SES formalisms to allow the efficient and hierarchical creation of structured test scenarios. The MICO concept allows message passing systems to be characterized in a way that allows for separate development of the message interaction, conditions, and ordering of messages. The final system developed based on the MICO concept is more easily and more fully automated than any incarnation of the ATC-Gen research project that existed before

6.3 Future Work

At the current time, only the Message Interaction portion of the code is generated automatically from the FDDEVS specifications. The scenarios that encompass pass/fail behavior are generated in the process, but this information is not used to generate code for test cases. Work is being done to use this scenario description to automate generation of the Ordering portion of a test case, in the form of the Acceptor model. The final form of the Acceptor model should allow test cases to be determined as a pass or fail automatically by the system.

In addition, characterizations of the types of conditions that commonly appear in test cases are being discussed. If conditions can be characterized in a general manner, automation of the Condition checking portion of the code, the ConditionChecker model, can be done in the future. When both the Condition portion and the Message Ordering portion of the MICO specification can be automated, truly automated test case generation will be possible.

7 References

[1]. Nutaro, J., Hammonds, P.; "*Combining the Model/View/Control Design Pattern with the DEVS Formalism to Achieve Rigor and Reusability in Distributed Simulation"*. JDMS, Vol. 1, Issue 1, April 2004 Page 19-28

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

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

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

[5]. Eddie Mak, Saurabh Mittal, Moon Ho Hwang, James Nutaro, "Automated Link 16 Testing using DEVS and XML", submitted to Journal of Defense Modeling and Simulation (JDMS), accessed at http://www.acims.arizona.edu/PUBLICATIONS/PDF/AutomatedTestingPaper.pdf

[6]. Mittal, S., “DEVS Unified Process for Integrated Development and Testing of Service Oriented Architectures”, Thesis, University of Arizona, http://www.acims.arizona.edu/PUBLICATIONS/PDF/Thesis_Mittal.pdf

[7]. “Extensible Markup Language”, http://www.w3.org/XML/Core/, last accessed July 16, 2009

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

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

[10]. Nutaro, J., “Adevs: A Discrete EVent System simulator”, http://www.ornl.gov/~1qn/adevs/. Last accessed July 26, 2009.

[11]. 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.

[12]. MIL-STD-6016C, Change 1, “TACTICAL DATA LINK (TDL) 16, MESSAGE STANDARD”. Department of Defense.

[13]. Hwang, M.H., Zeigler, B.P., “A Reachable Graph of Finite and Deterministic DEVS Networks”, Proceedings of 2006 DEVS Symposium, Huntsville, Alabama, USA, April, pp 48-56.

[14]. Hwang, M.H., Zeigler, B.P., “A Modular Verification Framework Based on Finite & Deterministic DEVS”, Proceedings of 2006 DEVS Symposium, Huntsville, Alabama, USA, April, pp 57-65.

[15]. M.H. Hwang, ``Generating Finite-State Global Behavior of Reconfigurable Automation Systems: DEVS Approach``, Proceedings of 2005 IEEE-CASE, Edmonton, Canada, Aug. 1-2, 2005

[16]. 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

[17]. Nutaro, J.; Jarboe, S.; Zeigler, B.; Fulton, D.; “A Method for Generating Synthetic Air Tracks”, Distributed Simulation and Real-Time Applications, 2008. DS-RT 2008. 12th IEEE/ACM International Symposium on 27-29 Oct. 2008 Page(s):245 – 251

[18]. Bhateja, P.; Mukund, M.; “Tagging Make Local Testing of Message-Passing Systems Feasible” Software Engineering and Formal Methods, 2008. SEFM '08. Sixth IEEE International Conference on 10-14 Nov. 2008 Page(s):171 – 180

[19]. Tsiatsoulis, Z.; Cotronis, J.Y.; Floros, E.; “Testing and debugging message passing applications based on the synergy of program and specification executions” Parallel and Distributed Processing, 1999. PDP '99. Proceedings of the Seventh Euromicro Workshop on 3-5 Feb. 1999 Page(s):196 - 203

[20]. Zeigler, B.P.; Sarjoughian, H.S.; Sunwoo Park; Nutaro, J.J.; Lee, J.S.; Cho, Y.K.; "DEVS modeling and simulation: a new layer of middleware", Active Middleware Services, 2001. Third Annual International Workshop on 6 Aug. 2001 Page(s):22 - 31

[21]. Hui Shang; Wainer, G.A.; "Dynamic Structure DEVS: Improving the Real-Time Embedded Systems Simulation and Design". Simulation Symposium, 2008. ANSS 2008. 41st Annual 13-16 April 2008 Page(s):271 – 278

[22]. Manuel C. Salas and B.P. Zeigler, AutoDEVS: A Methodology for Automating M&S Software Development and Testing of Interoperable Systems, submitted to Journal of Defense Modeling and Simulation (JDMS)

[23]. Dictionary.com, "Cybernetics," in The American Heritage® New Dictionary of Cultural Literacy, Third Edition. Source location: Houghton Mifflin Company, 2005. http://dictionary.reference.com/browse/Cybernetics. Available: http://dictionary.reference.com. Accessed: August 31, 2009

Appendix A: Acronyms

ATC-Gen	-	Automated Test Case Generator
ADEVS	-	A Discrete EVent Simulator
C2	-	Command and Control
DEVS	-	Discrete EVent Simulator
DIS	-	Distributed Interactive Simulation
DoD	-	Department of Defense
FDDEVS	-	Finite and Deterministic DEVS
HLA	-	High Level Architecture
JITC	-	Joint Interoperability Test Command
JPO	-	Joint Program Office
JUP	-	Joint Utility Player
MICO	-	Message Interaction, Condition-checking, Ordering model
Mil-Std	-	Military Standard
MSVC	-	Model/Simulator/View/Control

MVC	-	Model/View/Control
PES	-	Pruned Entity Structure
SES	-	System Entity Structure
SME	-	Subject Matter Expert(s)
SUT(s)	-	System(s) Under Test
TAMD	-	Theater Air and Missile Defense
TDL-J	-	Tactical Data Link, J-series
TIAC	-	TAMD Interoperability Assessment Capability
TMG	-	Test Model Generator
XML	-	eXtensible Markup Language

Appendix B: First Reconfiguration Design

The system designed First Reconfiguration consisted of the following models:

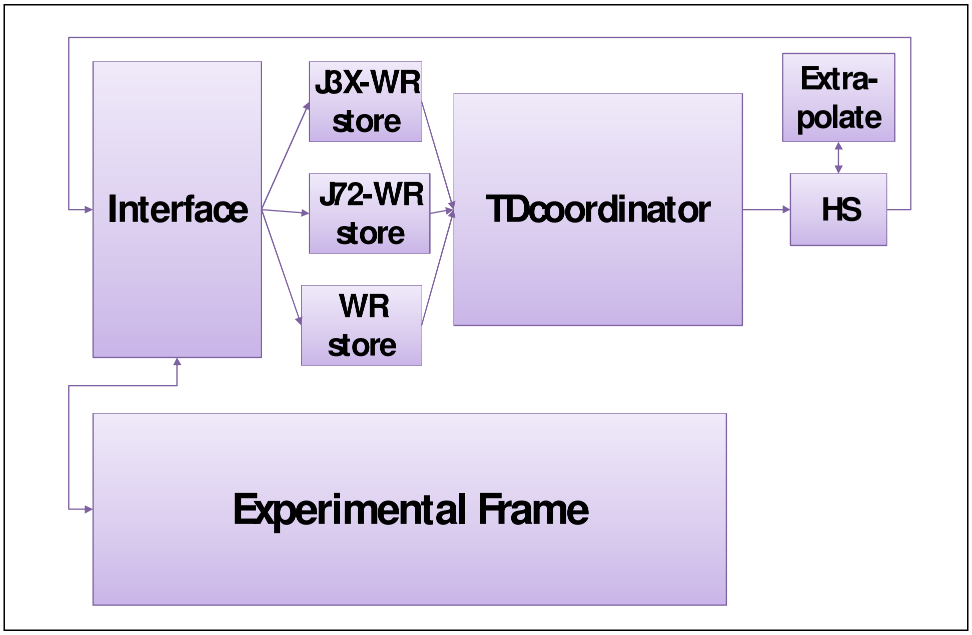


Figure 36 - Original Redesign Concept

INTERFACE

X = {inPlatformState, inJMsg, inStart, inStop}

Y = {outJMsg, outPlatformState}

S = {passive, Listen, PlatformStateToJMsg, JMsgToPlatformState}

The interface model represents the interface between the testing models and the real-world protocols on which the messages to be tested are sent and received. These are modeled externally by a class called JMsg, and they are translated to internal objects inherited from a class called PlatformState. The Interface can take in a message of type JMsg from the Experimental Frame and send out a message of type PlatformState to all instances of WaitReceiveStore. It can also take a message of type PlatformState bag from HoldSend and send out a message of type JMsg to the Experimental Frame.

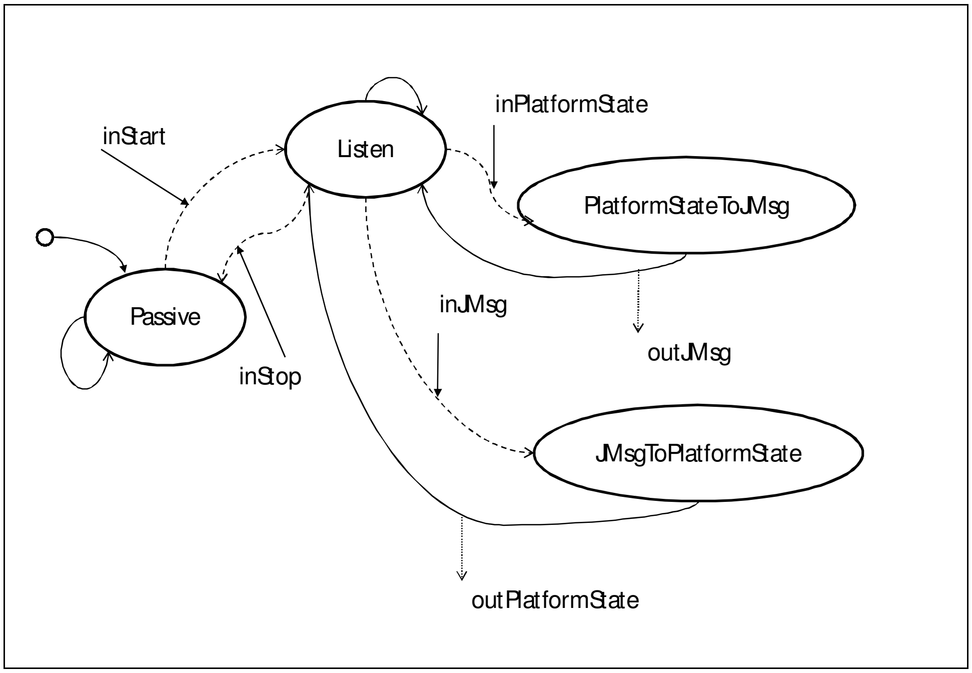


Figure 37 - Interface DEVS Diagram

The interface model starts in state passive. An input inStart is only accepted if the state is passive, and puts the model in state Listen. The passive and Listening states both have a time advance of infinity. Inputs inPlatformState, inJMsg, and inStop are accepted in state Listen. Input inPlatformState puts the model in state PlatformStateToJMsg, inJMsg puts the model in state JMsgToPlatformState, and inStop puts the model in state passive. The input inStop is accepted in state

PlatformStateToJMsg and causes a transition to state passive. PlatformStateToJMsg has a time advance of zero, and takes a PlatformState bag from HoldSend, transforms it into a JMsg bag, and outputs it on outJMsg to the Experimental Frame. The input inStop is accepted in state JMsgToPlatformState and causes a transition to state passive. JMsgToPlatformState has a time advance of zero, and takes a JMsg from the Experimental Frame, transforms it into a PlatformState, and outputs it to all WaitReceiveStore instances or inherited instances.

WAITRECEIVESTORE

X = {inPlatformState, inStart, inStop}

Y = {outPlatformState, outControl}

S = {passive, wait, Processing}

The WaitReceiveStore model is the base class for reception processing classes. It receives messages from Interface, tests them against a criterion, and, if they pass, creates two outputs, one containing the message for transmission and one containing control information, to the TDCoordinator.

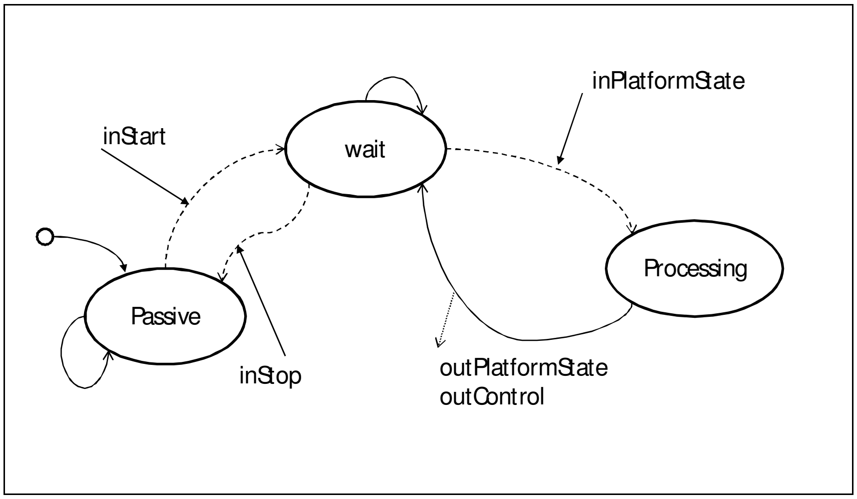


Figure 38 - WaitReceiveStore DEVS Diagram

The WaitReceiveStore model starts in state passive. An input inStart is only accepted if the state is passive, and puts the model in state wait. The passive and wait states both have a time advance of infinity. Inputs inPlatformState and inStop are accepted in state wait. Input inPlatformState puts the model from state wait into state Processing and input inStop puts the model from state wait into state passive. Processing has a time advance of zero, and takes a PlatformState from Interface, checks a condition or otherwise processes the message, and outputs it to TDCoordinator, along with an accompanying Control message.

<u>J3X</u>

X = {inPlatformState, inStart, inStop}

Y = {outPlatformState, outControl}

S = {passive, wait, Processing}

The J3X model inherits from the WaitReceiveStore class. All inputs, outputs, and states behave the same, with the exception of what is done during the Processing state. In J3X, Processing stores, processes, and transmits an incoming PlatformState only if it is a J3X message.

J70

X = {inPlatformState, inStart, inStop}

Y = {outPlatformState, outControl}

S = {passive, wait, Processing}

The J70 model inherits from the WaitReceiveStore class. All inputs, outputs, and states behave the same, with the exception of what is done during the Processing state. In J70, Processing processes an incoming PlatformState only if it is a J70 message. If the incoming J70 matches certain criteria, the message is passed to the TDCoordinator. Otherwise, it is ignored.

J72

X = {inPlatformState, inStart, inStop}

Y = {outPlatformState, outControl}

S = {passive, wait, Processing}

The J72 model inherits from the WaitReceiveStore class. All inputs, outputs, and states behave the same, with the exception of what is done during the Processing state. In J72, Processing stores, processes, and transmits an incoming PlatformState only if it is a J72 message. If the incoming J72 matches certain criteria, the message is passed to the TDCoordinator with a Control message of true. If not, the message is passed to the TDCoordinator with a Control message of false.

HoldSend

X = {inPlatformState, inExtrapolate, inStart, inStop}

Y = {outPlatformState, outExtrapolate}

S = {passive, waitInit, wait, send, Extrapolate}

The HoldSend model receives a message from the TDCoordinator and holds it until time for transmission. The time to transmission is determined from the Mil-Std 6016C document on which the system is based. The standard defines that all systems should send their messages at intervals. An interval of 12 seconds is chosen for the HoldSend model. The HoldSend is also responsible for passing the time received and time to send to the Extrapolate method for positional extrapolation, and putting the extrapolated position into the message to be sent.

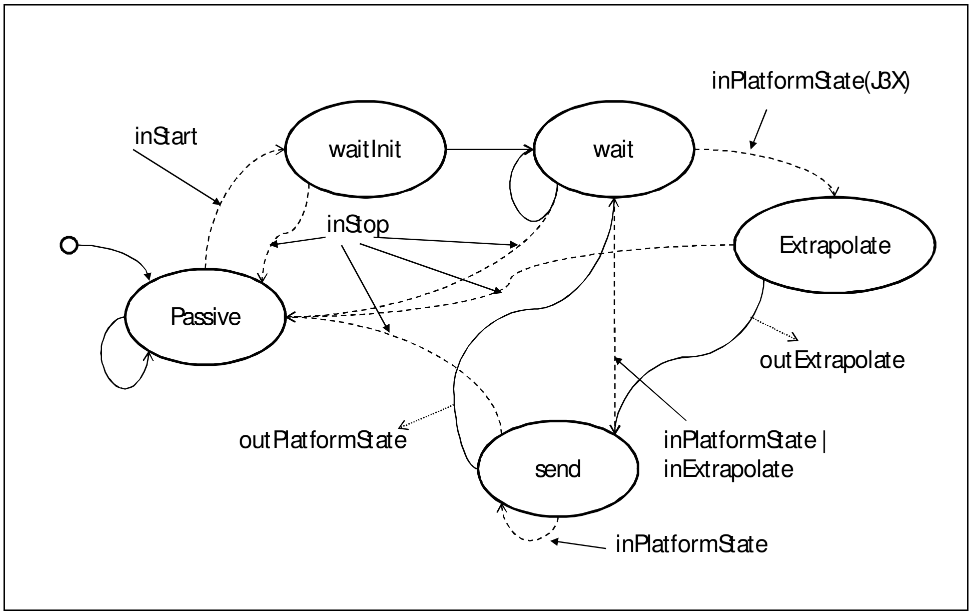


Figure 39 - HoldSend DEVS Diagram

The HoldSend model starts in state passive. The passive state has a time advance of infinity. An input inStart is only accepted if the state is passive, and puts the model in state waitInit. The waitInit state is the same as the wait state, with the exception that its time advance is determined by the coupling class, with a default of 2 seconds. The waitInit state goes into the wait state. Inputs inPlatformState and inStop are accepted in state wait. The wait state has a time advance of 12.0, and goes into a wait state. Input inPlatformState puts the model from state wait into state Extrapolate if PlatformState is of type J3X or into send if it is not, and input inStop puts the model from state wait into state passive. Extrapolate has a time advance of zero, and takes a J3X from Interface, and outputs it to the Extrapolate model. The send state has a time advance equal to the time remaining before wait would otherwise end, and takes a PlatformState from Interface, puts it into a bag, and sends it to the Interface model.

EXTRAPOLATE

X = {inExtrapolate, inStart, inStop}

Y = {outExtrapolate}

S = {passive, Wait, Extrapolate}

The Extrapolate model receives a set of positional data, a time received, and a time to send from HS, and outputs the extrapolated position at the time to send to HS. Since this system was only to be used for verifying behavior, no actual calculations were implemented in the Extrapolator model.

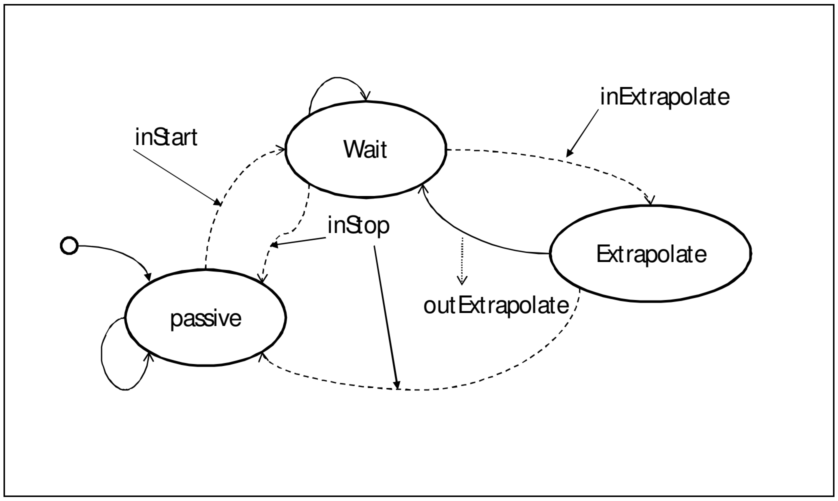


Figure 40 - Extrapolate DEVS Diagram

The Extrapolate model starts in state passive. An input inStart is only accepted if the state is passive, and puts the model in state Wait. The passive and Wait states both have a time advance of infinity. Inputs inExtrapolate and inStop are accepted in state Wait. Input inExtrapolate puts the model from state Wait into state Extrapolate and input inStop puts the model from state Wait into state passive. The input inStop is the only input accepted in state Extrapolate. Input inStop puts the model from state Extrapolate into state passive. Extrapolate has a time advance of zero, and takes a PlatformState from HoldSend, manipulates the positional data, and outputs it to HoldSend.

TDCOORDINATOR

X = {inStart, inMsg, inControl}

Y = {outPlatformState}

S = {passive, wait, waitForMessage, waitForControl, sendJ3X, sendJ70, fail, succeed}

The TDCoordinator model receives messages from WaitReceiveStore models, tests for behaviors, and outputs messages to HoldSend. The behavior tested depends on what tests the model implements. For the basic WaitReceiveSendJSearch model in the RemoteTNDrop scenario, it tests if 2 or more J3X are received and sent before a J72. However, it also includes a state that sends a J70 Drop Track report, which was modeled separately in the previous test cases. This illustrates that TDCoordinator is designed to encapsulate entire scenarios, rather than portions of scenarios. The TDCoordinator also tests for fail cases, such as receiving a J70 before a J72, receiving an incorrect J72, and receiving less than 2 J3X before a J72.

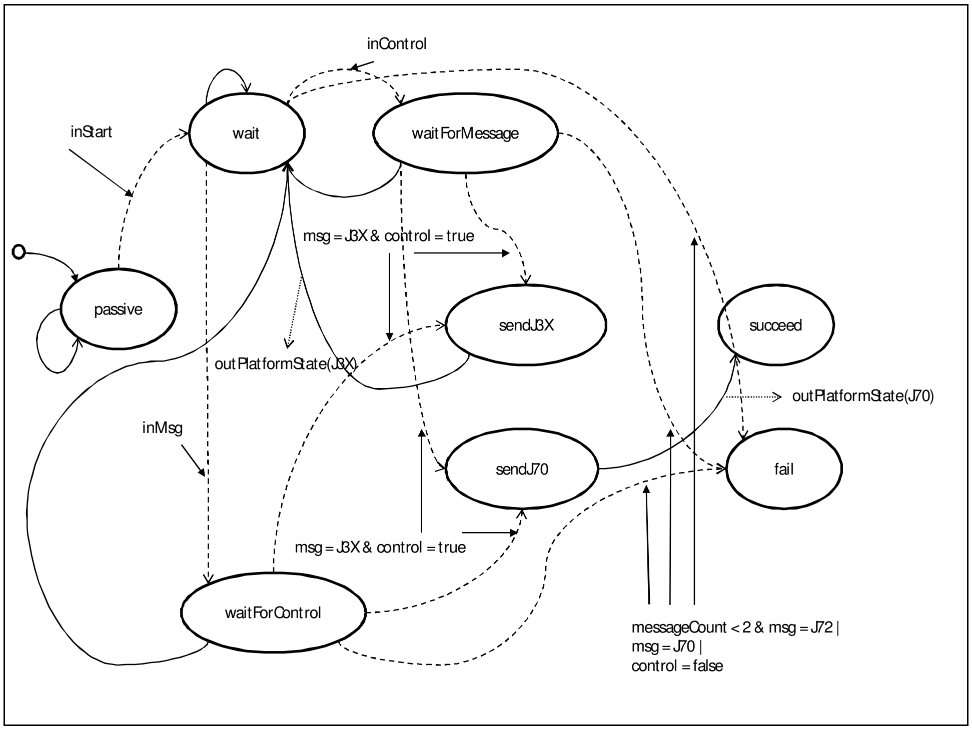


Figure 41 - TDCoordinator DEVS Diagram

The TDCoordinator model starts in state passive. An input inStart is only accepted if the state is passive, and puts the model in state wait. The passive and wait states both have a time advance of infinity. The input inStop is accepted in all states. Input inStop in any state but passive puts the model from into state passive. Due to the complexity of this model, the inStop inputs are omitted in the figure. Inputs inMsg and inControl are accepted in state wait.. Receiving input inMsg in state wait checks the message type, processes the message by type, and puts the model from state wait into state waitForControl, if the processing succeeds. If the processing fails, the model passivates in fail. Input inControl in state wait checks the control message and puts the model from state wait into state waitForMessage if the processing succeeds. If the processing fails, the model passivates in fail. State waitForControl has a time advance of 0.1, and accepts an input inControl. Input inControl in state waitForControl checks the control message and puts the model from state waitForControl into state sendJ3X if the processing succeeds and the message is a J3X, or into state sendJ70 if the processing succeeds and the message is a J72. If the processing fails, the model passivates in fail. The state waitForMessage has a time advance of 0.1 and accepts an input of inMsg or inStop. Input inMsg in state waitForMessage checks the incoming message and puts the model from state waitForMessage into state sendJ3X if the processing succeeds and the message is a J3X, or into state sendJ70 if the processing succeeds and the message is a J72. If the processing fails, the model passivates in fail. The state sendJ3X has a time advance of zero. It takes a received J3X and outputs it to the HoldSend model. The model sendJ3X goes into state wait when its time advance elapses. The state sendJ70 has a time advance of zero. It takes a received J72, and outputs a J70 to the HoldSend model. The model sendJ70 goes to state succeed when its time advance elapses.

Appendix C – Example implementation of Test Scenarios

1. Original ATC-Gen Model

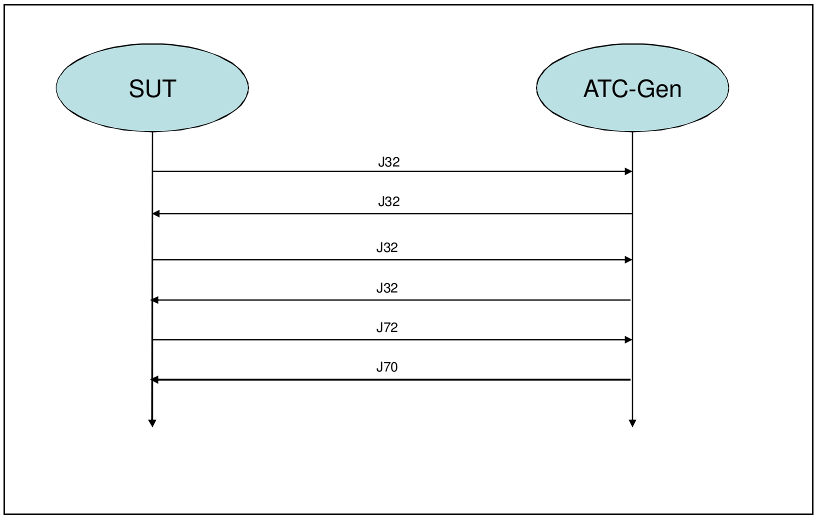


Figure 42 - Basic Correlation Scenario

Correlation is the process by which two Control Units, units in a Link 16 system with the capability to report tracks, resolve that an object seen by both is in fact the same object. In this system, the basic model for correlation was developed as a set of message interactions, as shown in Figure 42. From the perspective of the SUT, this test case required that, while the SUT was periodically sending J32 Track Report messages, it would expect to receive two J32 Track Report messages. On each reception, it would test criteria for correlation. Reception of the second J32 Track Report message would trigger a correlation, if the tests passed, causing the SUT to send a J72 Correlation message. The SUT would then wait for a J70 Drop Track message to indicate that the correlation resolved successfully and the other system would stop sending updates on the track. After the generation reached this point, the model still needed to undergo the "reflection" described earlier. The reflection is done as in Figure 43 [5]. All waitReceives become holdSends, and all holdSends become waitReceives. This resulted in a scenario that would cause the SUT to behave in a manner as described in the standard.

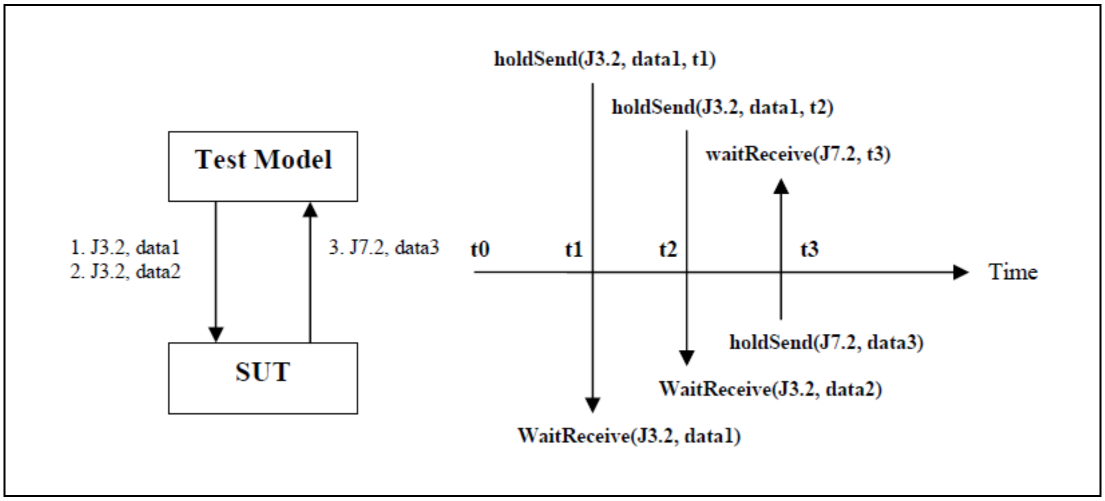


Figure 43 - Reflecting the Test Model

The final resulting model is shown in Figure 44. The first holdSend model sends a J3X Track report, then triggers the second holdSend model. The second holdSend model sends a second J3X Track report, then triggers the waitReceive. The waitReceive waits for a J72 Correlation report, then sends a pass message to the external model. When a holdSend that ouputs a J70 drop track message is coupled to the pass output of the coupled model in Figure 44, this scenario is named RemoteTNDrop in the terminology used by the analysts. It is the basic test case used throughout this paper as an example, as well as being the commonly used proof of concept test case.

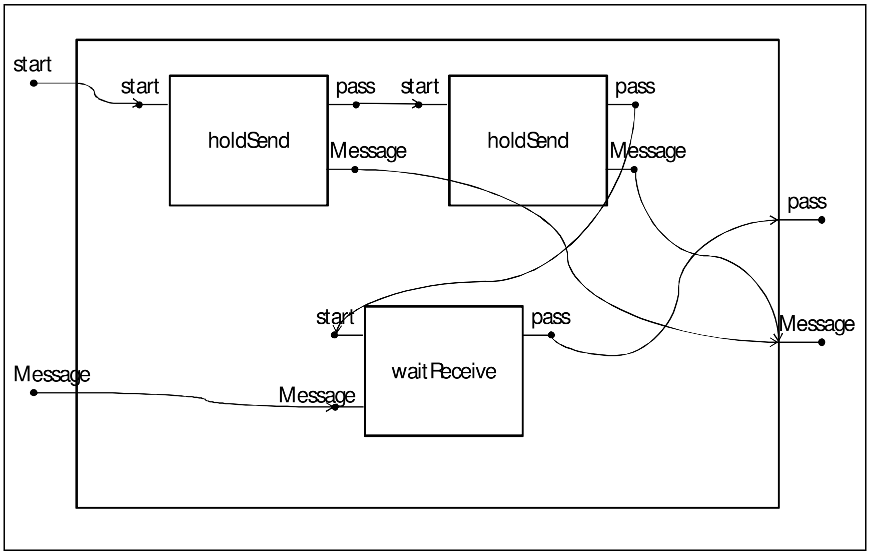


Figure 44 - Correlation Coupled Model, ATC-Gen perspective

The external box encapsulating the coupled model is the interface with the simulator and the message protocols. In relation to the MSVC design pattern described in the above section, the coupled model in the figure represents the Model portion. In the implementation of the system, it is coupled to an interface to the Simulator portion, and an interface to the message protocol implementation, which is where the View and Control portion is coded.

2. Reactive Mode Model

In the Reactive Mode implementation of the code, the correlation example described in the last section became a model known as waitRecieveSendJSearch. This model is shown in Figure 45.

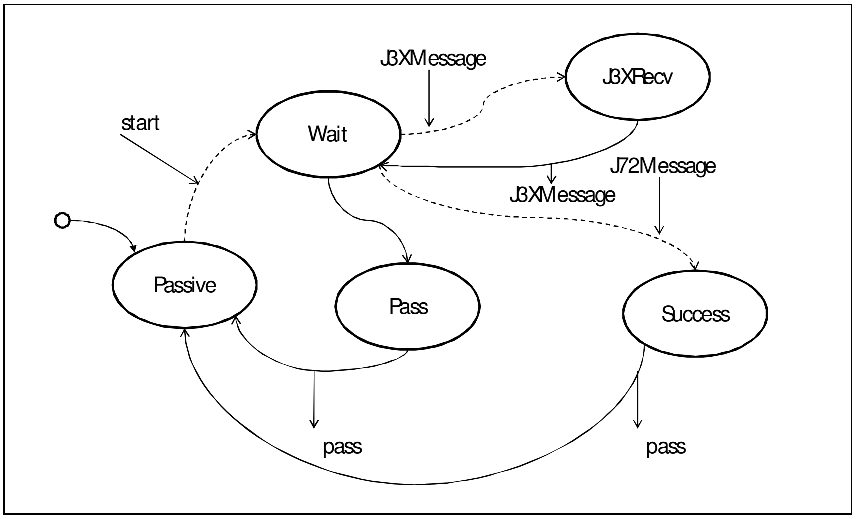


Figure 45 - waitReceiveJSearch DEVS Diagram

The model waitReceiveSendJSearch encompasses all the behavior in the previous coupled model. It has a state to keep track of the number of J3X track update reports received, called J3XRecv, which also reflects the message and outputs a J3X based on the received J3X message. It has a state called Success which keeps track of the receipt of a J72 Correlation message, and takes the place of the waitReceive model in the previous coupled model. It takes a start input to put the model into wait, in the same way as the coupled model before. It expects messages as input and creates messages as output, in the same manner as the coupled model. It also creates a pass message when the model is complete, in the same manner as the coupled model. If a J70 drop track is coupled to the pass model, the scenario is once again the RemoteTNDrop scenario mentioned in the last section. In all ways, this model encapsulates the behavior of the previous coupled model.