Specification Of Distributed Interactive Simulation In The Estelle Formal Description Technique: Investigation Of OSI Protocols For Distributed Interactive Simulation

David T. Shen

Huat K. Ng

INSTITUTE FOR SIMULATION AND TRAINING

Contract Number N61339-91-C-0103
Investigation of OSI Protocols for Distributed Interactive Simulation
Prepared for
U.S. Army Project Manager for Training Devices

April 23, 1992

Specification of Distributed Interactive Simulation in the Estelle Formal Description Technique

IST

Institute for Simulation and Training
12424 Research Parkway Suite 300
Orlando FL 32826

University of Central Florida
Division of Sponsored Research

IST-TR-92-17

INSTITUTE FOR SIMULATION AND TRAINING

Specification of Distributed Interactive Simulation in the Estelle Formal Description Technique

Contract N61339-91-C-0103
Investigation of OSI Protocols for Distributed Interactive Simulation

Prepared for
U.S. Army Project Manager for Training Devices
12350 Research Parkway
Orlando, FL 32826-3276

April 23, 1992

IST-TR-92-17

Prepared by [illegible]

David T. Shen, Huat K. Ng [illegible]

Reviewed by [illegible]

Scott Smith

SPECIFICATION OF DISTRIBUTED INTERACTIVE SIMULATION IN THE ESTELLE FORMAL DESCRIPTION TECHNIQUE

PREPARED FOR:

U.S. ARMY PROJECT MANAGER FOR TRAINING DEVICES
12350 Research Parkway
Orlando, FL 32826-3276

INVESTIGATION OF OSI PROTOCOLS FOR
DISTRIBUTED INTERACTIVE SIMULATION

CONTRACT N61339-91-C-0103

April 23, 1992

Institute for Simulation and Training
University of Central Florida
12424 Research Parkway
Orlando, FL 32826

ABSTRACT

The Distributed Interactive Simulation (DIS) Entity Information and Entity Interaction Draft Standard defines a communication protocol to interconnect simulators in a real-time environment. This protocol focuses on the information for describing the state of the simulated entities and their interactions during a battle simulation.

Many of the concepts in the DIS standard are derived from the Simulation Network (SIMNET) project. The SIMNET program has demonstrated the feasibility of interconnecting multiple autonomous simulators, primarily of ground based armor vehicles, via a communication network (LAN/WAN), such that the simulators could interact in real-time [POPE]. DIS is based upon the foundations of SIMNET and will be enhanced to provide a standard for connecting existing and future simulators. However, a formal description of the DIS standard is yet to be developed, which would, in turn, speed its prototyping, development, and testing processes.

This report describes the approach taken by the Institute for Simulation and Training (IST) to develop a formal description of DIS, to generate a prototype DIS protocol machine derived directly from the developed DIS standard, and to test the DIS protocol. IST used an International Organization for Standardization (ISO) standard Formal Description Technique (FDT) called Estelle for the formal specification, and a FDT prototyping tool known as the Portable Estelle Translator - Distributed Implementation Generator (PET-DINGO) to generate a DIS prototype. Furthermore IST developed a procedure to test the generated prototype.

The aim of this work was to develop a formal specification for DIS and to identify possible shortcomings and inconsistencies in the DIS standard. This task identified some areas in the current standard that need to be modified or clarified. The recommended modifications to the DIS standard were submitted to the respective DIS standard working group for consideration.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

The Institute for Simulation and Training along with the U. S. ARMY Project Manager for Training Devices (PM TRADE) and Defense Advance Research Project Agency (DARPA) has been promoting the interoperability of defense simulations through the DIS workshops. The goal of the workshops is to gather expertise from the simulation community and to advance the development of a standard to allow defense simulations to interact through networking. These types of simulations provide an environment for training inter-crew skills and provide an environment for evaluation of tactics, doctrines, and new equipment features.

The developed standard must be tested for validation. This test will create confidence in the standard's completeness and domain of applicability for implementers. This validation should begin with the initial stage of release.

The DIS draft standard is in its second year of development. The standard is being reviewed and is proposed to the IEEE in 1992 for adoption. Currently, the DIS standard covers only layer 7 of the Open System Interconnection (OSI) Reference Model [ISO7498]. Layer 7 is the Application Layer, which is the simulation entity information level. It does not cover the network architecture or communication protocols below the Application Layer. Several defense procurements have specified the DIS standard as the baseline for the interoperability requirements.

IST has experience in many areas of computer and simulator networking. In its effort to assess the suitability of existing OSI protocols for real-time simulations, IST has developed a formal specification of the DIS draft standard using the Estelle Formal Description Technique (FDT) and the Portable Estelle Translator - Distributed Implementation Generator (PET-DINGO) compiler.

Estelle is a standard FDT developed in 1989 by the International Organization for Standardization (ISO) to specify distributed, concurrent information processing systems using a Pascal like language. Estelle is not a programming language for implementation, but simply for specification, with its own syntax and semantics.

The PET-DINGO compiler, developed by the National Institute of Standards and Technology (NIST), is a protocol prototyping tool that accepts an Estelle specification and produces a runtime environment simulating the specified protocol behavior. The PET-DINGO was developed to familiarize the scientific community with the Estelle specification language and thus promote its use.

2 GENERAL DESCRIPTION

This section briefly describes the DIS standard, the Estelle FDT and the PET-DINGO compiler.

2.1 Distributed Interactive Simulation

The October 30, 1991 release of the DIS standard defines 10 types of Protocol Data Units (PDUs) to be used by networked simulators to represent the state of the simulation entities during a battle simulation [DIS-STD]:

1) Entity State

2) Service Request

3) Repair Complete

4) Repair Response

5) Resupply Offer

6) Resupply Received

7) Resupply Cancel

8) Collision

9) Fire

10) Detonation

These 10 PDU types can support the entity behavior during repair, resupply, fire, collision, motion and entity appearance updates, which are the vital components of a simulation environment, and they are the core of this research task.

The Entity State PDU is used most often during a simulation exercise. It describes the status of its simulated entity's location, velocity, acceleration, ammunition, and articulated parts. It is transmitted when a dead reckoning model of an entity's state diverges from its own high fidelity model by a predetermined threshold. When an Entity State PDU is received by other simulators in the exercise, these simulators update their state information regarding the entity.

The Service Request, Repair Complete and Repair Response PDUs are used to represent a repair event involving two simulated entities. These PDUs carry the necessary information to determine the types of repair performed and the level of satisfaction of the repair. The repair PDUs are request/response type, meaning, some PDUs are used in response to the receipt of other PDUs.

The Service request, Resupply Offer, Resupply Received and Resupply Cancel are used to represent a resupply event involving two simulated entities. These PDUs contain the necessary information to determine the type of resupply needed. The resupply PDUs are also request/response type, meaning, some PDUs are used in response to the receipt of other PDUs.

The Collision PDU is used by two simulated entities when involved in a collision. The information contained in this PDU is needed for collision damage assessment.

The Fire PDU and the Detonation PDU are used to inform a target entity of the weapon fire event and the associated detonation of the fired munition. The entity which fires a weapon models the munition's trajectory and informs the target of the point of impact. It is the responsibility of the targeted entity to assess its damage using the information in the Detonation PDU.

2.2 Estelle Formal Description Technique

The problem of specifying distributed systems is more difficult than that of specifying a sequential system. The difficulties are related to the necessity of describing various sequential components which may cooperate and execute in parallel. To attain reliability in production software, a protocol development should begin with a formal specification.

Estelle is a language for specifying distributed systems with a particular application in mind, namely that of communication protocols and services [BUDKO, ISO9074]. The semantics for Estelle have been formally defined and are aimed at describing structured communicating automata (states) whose internal actions are defined by Pascal programming language statements (with some restrictions and extensions).

The benefit in using a FDT, in particular Estelle, is to remove the ambiguities from the protocol description, traditionally defined in a combination of natural language and state tables. Another benefit is the availability of tools that use Estelle to generate rapid prototypes and test suites.

The three main components of the Estelle structure are the Module, the Interaction, and the Channel. The correct use of these components is essential for a specification.

The Modules have a number of input/output access points known as the Interaction Points. A module will be represented graphically as a rectangle and the interaction points will be represented by dots on its boundary. An active module includes in its transition part at least one transition. Each active module specifies its own states and the rules for state transition. The collection of the states, the possible state transitions, and the module variables is known as the Extended State Transition Model (ESTM).

The modules can be organized in a hierarchical (tree) structure, with parent/children relationships. Each module can have several embedded modules (children), and these modules can, in turn, include other embedded modules.

The Interactions are the messages sent and received by the modules. The interactions serve as the means of communicating information between modules. The modules can send an interaction at any time, yielding non-blocking communication. An interaction sent through an interaction point that is an end-point of a communication link directly arrives at the other end point of this link and is always accepted by the receiving module. Thus only end-to-end communication between modules is possible.

Modules (parents or children) can establish communication with other modules using channels. Channels are defined as one-to-one connections between the modules through which the modules interact. Each channel essentially connects two interaction points. Each channel is associated with an unbounded first-in-first-out queue for the incoming interactions as they arrive for processing. There are several restrictions imposed in the way channels may interconnect the modules. Channels also define the types of interactions that may pass through it.

Each Estelle module definition is composed of a heading and a body that describes its behavior. A detailed description of Estelle is presented in Appendix A. A transition takes place in response to an internal or external event to the module and it may generate an interaction to the connected modules. The global properties of the Estelle FDT supports a logical specification (modeling) of a communication protocol.

2.3 Portable Estelle Translator & Distributed Implementation Generator

The PET-DINGO prototyping tool, developed by NIST, is a twofold compiler that generates a static model and a dynamic model from an Estelle specification. The PET compiler takes an Estelle specification and checks it for syntax, semantic or lexical error and then compiles it into an object code (static model) [SIJEL1]. The DINGO compiler takes the a static model and generates a series of C++ source files associated with specified Estelle modules describing their behavior (dynamic model) [SIJEL2].

The PET-DINGO was written in the C++ programming language. It supports network communication using either Transmission Control Protocol/Internet

Protocol (TCP/IP) or Remote Procedure Call (RPC), as a consequence, generated processes can run on diverse computers on a network . It is designed to run on a Sun3 or Sun SPARC hardware platform. It also supports the X-window environment, which facilitates the user's interaction with the specified protocol processes.

Figure 2.1 shows the process to build and execute a runtime instance of an Estelle specification:

1) Compile the Estelle specification using the PET to generate an object file. At this stage, the decision whether to run the specification in a multi-host environment is made.

2) Compile the PET generated object file using DINGO to generate C++ source files and the Makefile.

3) Add user defined programs [see Appendix C] and set system parameters.

4) Modify the Makefile [see Appendix C] as needed by including user defined programs.

5) Use the Makefile to generate system executables from the C++ files, which are the Estelle modules.

6) Initialize the site-daemon (called site_serv) to manage the runtime processes and the network interface. The site-daemon has to be initialized in each host if the processes is to run in a multi-host environment.

7) Call the Estelle specification top level module name for modules initialization.

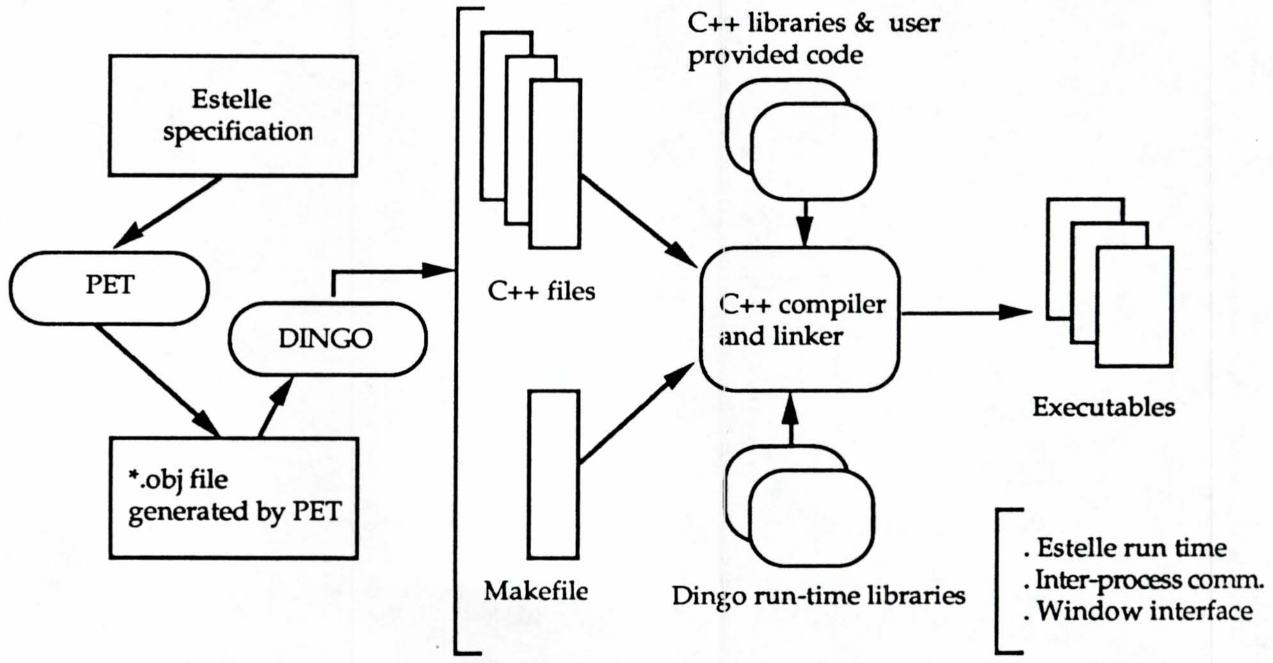


Figure 2.1 - PET-DINGO Compilation Procedure

Each Estelle module is a process that can be accessed through a window interface. A user specifies interactions by clicking the appropriate item of the window. The windows display modules' state, the value of the variables at any given time, and the queue of interactions associated with each channel connected to the module.

The module windows can be opened by clicking the name of the module on the root window. The embedded process windows can be opened by first opening their parent's window. The hierarchy of the windows follows the hierarchy of the specified Estelle modules.

3 SPECIFICATION PHASE

This section describes the approach taken to specify the DIS standard using the Estelle FDT. The model used for the DIS protocol with the associated assumptions and constraints is also described.

3.1 Approach

A formal specification of a protocol using the Estelle FDT is not unique, meaning there are several valid ways, to specify a particular protocol. IST has experienced several difficulties in specifying the DIS PDU interactions in Estelle because of the nature of the information passed from host to host and the type of interaction specified in the DIS standard.

The approach taken by IST to specify the DIS PDU interactions using Estelle is a model following an initiator/responder paradigm. This model uses a general view of a simulation entity from a driver's perspective, meaning that the model follows a hierarchical structure and the lower modules view the upper modules as the initiators of the processes by making the appropriate choices as to what actions to take.

3.2 Model

The model identifies the Repair and Resupply activities as the ones having a initiator/responder type of interaction. Other activities like fire and collision are essentially non-replied interactions, which do not follow the initiator/responder scheme.

The model includes four basic modules shown in Figure 3.1:

Driver Module: interfaces the user to the protocols specified within the Core module. The Driver Module is responsible for starting a protocol process and communicating appropriate decisions.

Core Module: includes 5 embedded modules, namely the Splitter, Combiner, Logistic, Fire and Assess. It models the DIS PDU interactions.

Network: models the physical/logical linkage between the protocols (the entities). It is responsible for end-to-end transmission of protocol messages between two entities.

Network Switch: allows user control of the network, giving the user the ability to interrupt the message transmission. Its existence is purely for testing purposes.

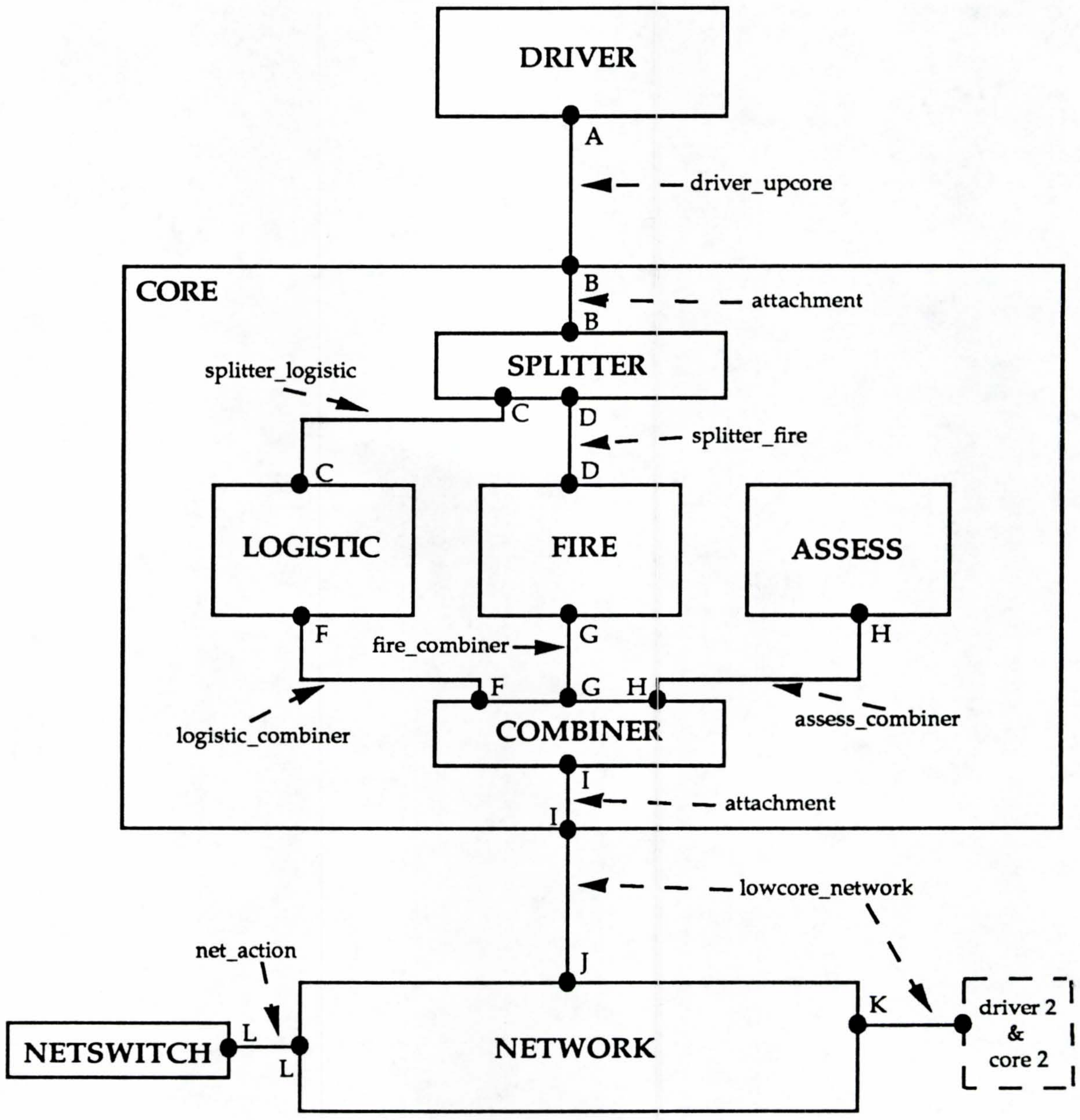


Figure 3.1 - Estelle Structure of DIS Protocol

Figure 3.1 shows the modules with their names in bold type, the associated interaction points in capital letters and the channel names indicated by arrows.

Within the Core module are the independent sub-modules of interest of this task:

Logistics Module: models both the repair and resupply (logistic) events of DIS. It can act as either initiator or responder, but not both. This module uses six types of PDUs (2 to 7 defined in sec. 2.1) to inform the peer entity of the action taken by the other. It incorporates reliability features by using some of the PDUs to acknowledge other PDUs. Such use of PDUs can be classified as an application level acknowledgement.

Fire Module: models the fire and detonation events. It uses the Fire PDU and Detonation PDU to convey related information to peers. The Fire PDU and the Detonation PDU do not require acknowledgement from their intended target.

Assess Module: responsible for updating the internal representation of a simulation. For instance, the Assess module represents updates to an entity's appearance during a simulation exercise when it receives an Entity State PDU, and it is also responsible for representing the fire event and damage assessment caused by a munition detonation or a collision. As far as the specification is concerned, the assessment means that the module transitions from an idle state to a particular type of assessment state and then back to the idle state.

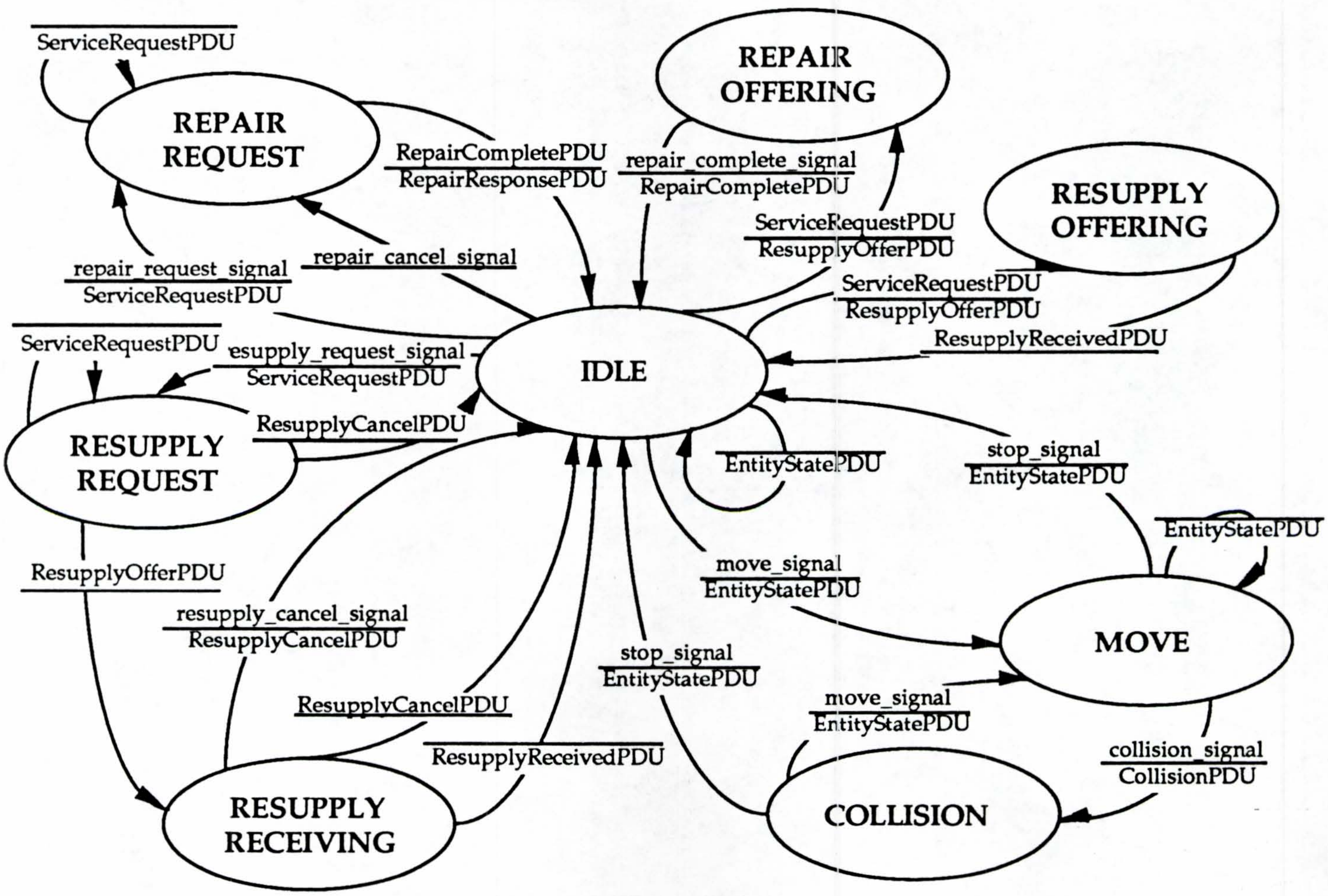
There are two auxiliary sub-modules within the Core module:

Splitter Module: responsible for vectoring the user interactions (signals) to the appropriate protocols within peer sub-modules. The rationale is to allow a higher level of abstraction for the connection between the Driver module and the Core module. Without the Splitter module, one would have to be concerned with the various connections from the Driver module to the Core sub-modules.

Combiner Module: responsible for piping the outgoing interactions (PDUs) from protocols within peer sub-modules (i.e., Logistics, Fire or Assess modules) to a single channel connected to the Network module. It is analogous to the Splitter module performing the reverse function. The rationale is to allow a higher level of abstraction for the connection between the Core sub-modules and the Network module. It is also responsible for conveying state variables among sibling modules.

Among all the referred modules, the Logistic, the Fire and the Assess modules are the only ones that are mapped into the DIS PDU interaction, other modules are intended for interface and testing purposes and are not part of the DIS specification.

The extended state transition model of the modules that maps to the DIS PDU standard are explained below:



ServiceRequestPDU
REPAIR REQUEST
REPAIR OFFERING
RepairCompletePDU / RepairResponsePDU
repair_complete_signal / RepairCompletePDU
RESUPPLY OFFERING
ServiceRequestPDU / ResupplyOfferPDU
repair_request_signal / ServiceRequestPDU
repair_cancel_signal
ServiceRequestPDU / ResupplyOfferPDU
ResupplyReceivedPDU
ServiceRequestPDU
resupply_request_signal / ServiceRequestPDU
IDLE
ResupplyCancelPDU
RESUPPLY REQUEST
EntityStatePDU
stop_signal / EntityStatePDU
EntityStatePDU
ResupplyOfferPDU
move_signal / EntityStatePDU
resupply_cancel_signal / ResupplyCancelPDU
MOVE
stop_signal / EntityStatePDU
ResupplyCancelPDU
move_signal / EntityStatePDU
ResupplyReceivedPDU
collision_signal / CollisionPDU
RESUPPLY RECEIVING
COLLISION

Figure 3.2 - Extended State Transition Model for Logistic Module

Figure 3.2 unifies the existing repair/resupply related state transition models defined in the DIS standard with the entity movement and collision activities. The repair/resupply side from one entity responds to the repair/resupply side of another entity, which creates a reply/response type of interaction between two Logistic modules. A clock routine returns the system time which is used in the repair and resupply transition models.

Because no collision ESTM is specified in DIS, the SIMNET model is assumed. An entity involved in a collision sends a Collision PDU to the other entity, communicating the collision. The receiver of the Collision PDU replies to the first entity with another Collision PDU, as an acknowledgement of the collision.

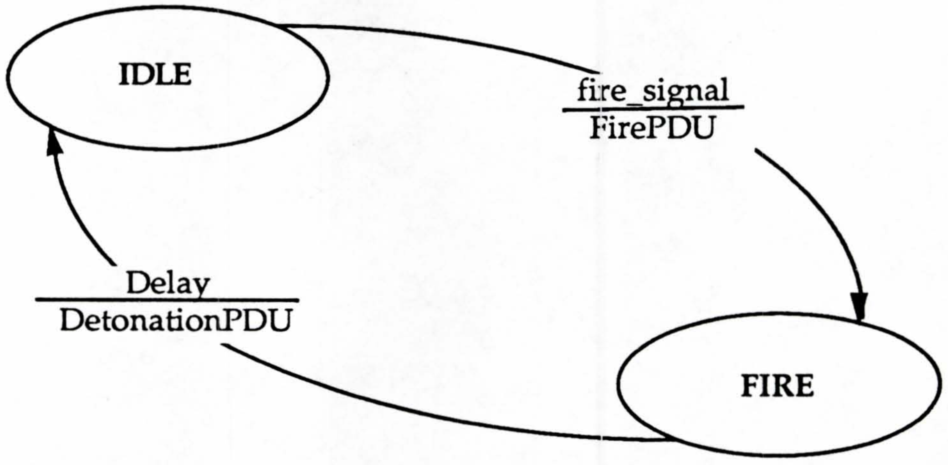


Figure 3.3 - Extended State Transition Model for Fire Module

The Fire module, shown in Figure 3.3, is composed of two states: IDLE and FIRE. An entity transitions to the FIRE state on the Driver's signal representing weapons fire and returns to IDLE state after a delay, simulating the time required for flight and detonation of the fired munition.

Because no fire ESTM is specified in DIS, the SIMNET model is assumed. The initiator sends a Fire PDU immediately followed by a Detonation PDU. These PDUs are conveyed to the Assessment module which simulates the internal processing of the receiving simulator.

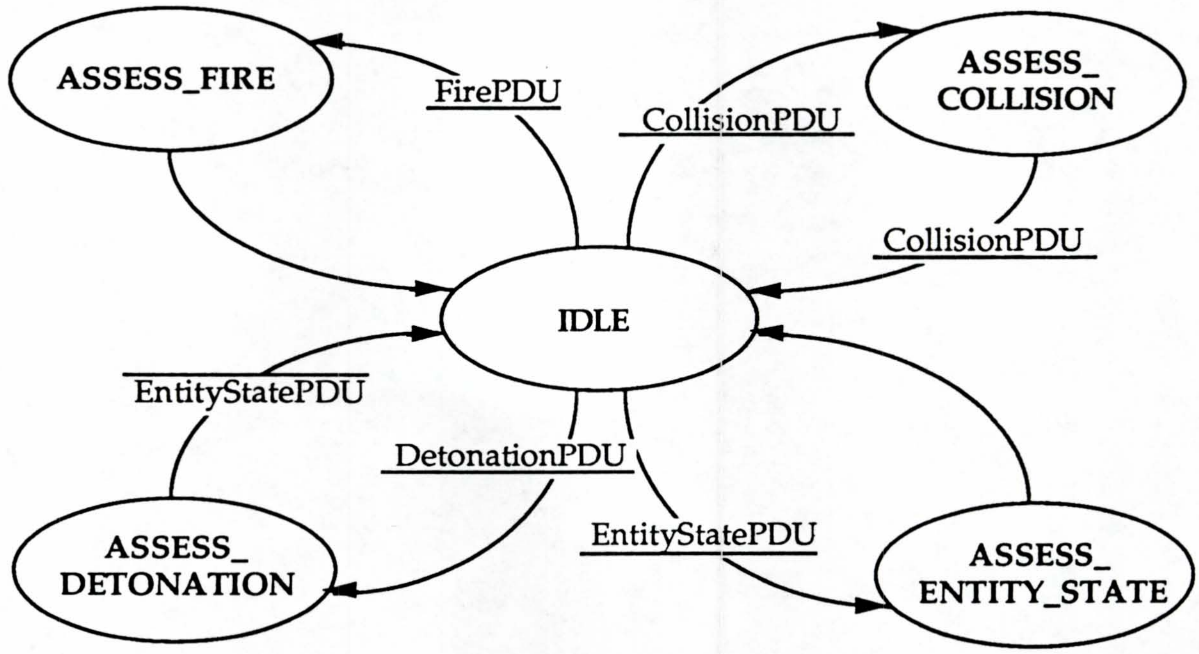


Figure 3.4 - Extended State Transition Model for Assess Module

The ASSESS module in Figure 3.4 represents the possible assessments a simulator can perform: fire assessment, detonation assessment (damage assessment), collision assessment (damage assessment), and entity state assessment.

Because no assessment ESTM is specified in DIS, the SIMNET model is assumed. It serves the purpose of isolating the information update event from the usual simulation related events allowing concurrent processing for the specified model.

3.3 Assumptions

There are several assumptions made to facilitate the model's creation:

1) The repair activities and resupply activities (logistic) do not take place concurrently. This allows all four state transition models defined in the standard associated with these activities to be combined in a single transition model.

2) An entity cannot repair or resupply while in motion. This assumption allows the inclusion of collision and movement events in the logistic module.

3) The logistics activities, the fire activities and the PDU assessment activities can be processed concurrently. This is represented in the core module through three independent sub-modules.

4) The resupply activity, once initiated, takes 15 seconds to complete.

5) The interval between consecutive Entity State PDUs is 1 second for a moving entity.

6) The interval between a weapon fire and the detonation of the fired munition is 2 seconds.

7) The assessment related activities are instantaneous, which is represented by the immediate transition from any type of assessment state to idle state in the Assess module.

8) The IDs used by the entities for identifying the collision events are consecutive integers starting at 1.

9) All interactions are received by the Assess module except for the repair and resupply related interactions.

3.4 Constraints

The following constraints are imposed by the Estelle specification:

1) There are only two entities modeled in the module for testing purpose.

2) The module interactions do not carry variable structure or concrete values.

4 TESTING PHASE

As mentioned earlier, the DIS Standard is in its infancy. Efforts must be channeled toward writing a test plan that will discover whether the DIS protocol is complete and valid. The DIS standard specifies procedures and formats for the exchange of information between heterogeneous simulators. Based on these specifications, a test plan can be written.

Protocol tests can be divided into three different types: Valid, Inopportune, and Invalid Message Tests [BERTINE]. Valid Tests are those where the tester sends messages at times and in sequences that are expected or normal for the Implementation Under Test's (IUT's) state. Inopportune Tests are those where the tester sends messages at times when they should not occur or are out of sequence. Invalid Message Tests determine if the IUT correctly handles receipt of messages that are incorrectly encoded, have illegal fields, or have parameters outside their legal bounds. Since the DIS specification in this report does not include the actual bit structure of the PDUs, Invalid Message Tests are inappropriate and will not be included in the test plan.

The DIS validation is an interactive process. Experience gained during the testing phase is used to enhance the quality of the specification and testing process. If the test plan uncovers an ambiguity or incompleteness in the specification, the specification can be modified and re-tested, leading to further refinement.

The hardware used in the IST testing environment consists of two Sun SPARC Workstations connected by Ethernet. The systems used Sun Open Window, a graphical user interface based on the X-window environment. The experiments were conducted in a multi-host environment, and both computers shared the files generated by PET-DINGO.

4.1 Testing Procedure

Each test step follows this procedure:

- Initialize the IUT and manipulate it into the desired state.

- Apply the specified input.
- Observe and validate the output.
- Verify the new state is as expected.

There are two choices available when executing the runtime PET-DINGO files of an Estelle specification: continuous mode or single-step mode. Running the DIS specification in single-step mode is the more appropriate choice because this allows the tester to observe the changed states and messages passed between modules at his or her own pace.

4.1.1 Valid Testing

The test plan is written in a tabular format. It consists of Valid test scripts. Four tables describe the DIS state transitions. These are:

- Logistic / Logistic - Resupply Service
- Logistic / Logistic - Repair Service
- Logistic / Assess
- Fire / Assess

Each table is separated into an Initiator column and a Responder column. Both Initiator and Responder are separated again into four other columns: Input, Output, Current State and Next State. In this fashion, the tables follow a natural sequence of the state transitions, which facilitate the observation of the transitions and messages that are conveyed between two modules. The tables used in the test plan are shown in Tables 4.1 - 4.4.

Logistic (Initial) / Logistic (Respond)	Initial Transition				Respond Transition			
	Input	Output	Current State	Next State	Input	Output	Current State	Next State
Request Service / Offer Supplies	Request Resupply	Service Request PDU	Idle	Requesting	Service Request PDU	Resupply Offer PDU	Idle	Offering
Offer Supplies / Receive Offer	Service Request PDU	Resupply Offer PDU	Idle	Offering	Resupply Offer PDU	-	Requesting	Receiving
Accept Service / Resupply Complete	-	Resupply Received PDU	Receiving	Idle	Resupply Received PDU	-	Offering	Idle
Reject Offer / Resupply Cancelled (by Receiver)	Resupply Cancel	Resupply Cancel PDU	Receiving	Idle	Resupply Cancel PDU	-	Offering	Idle
Resupply Cancelled (by Supplier) / Transfered Cancel	Resupply Cancel	Resupply Cancel PDU	Offering	Idle	Resupply Cancel PDU	-	Receiving	Idle
Repeat Request / Offer Supplies	-	Service Request PDU	Requesting	Requesting	Service Request PDU	Resupply Offer PDU	Idle	Offering
Resupply Refuse / Cancel Request	Resupply Cancel	Resupply Cancel PDU	Idle	Idle	Resupply Cancel PDU	-	Requesting	Idle

Table 4.1 - Logistic / Logistic - Resupply Service

Logistic (Initial) / Logistic (Respond)	**Initial Transition**				**Respond Transition**			
	Input	**Output**	**Current State**	**Next State**	**Input**	**Output**	**Current State**	**Next State**
Repair Request (receiver) / Repair Request (repairer)	Request Repair	Service Request PDU	Idle	Requesting/ Receiving	Service Request PDU	-	Idle	Offering
Repair Complete (repairer) / Repair Complete (receiver)	Repair Complete	Repair Complete PDU	Offering	Idle	Repair Complete PDU	Repair Response PDU	Requesting/ Receiving	Idle
Cancel Request / Repair Cancel	Cancel Request	-	Requesting/ Receiving	Idle	-	-	Offering	Idle
Repeat Request / Repeat Offer	-	Service Request PDU	Requesting/ Receiving	Requesting/ Receiving	Service Request PDU	-	Offering	Offering

Table 4.2 - Logistic / Logistic - Repair Service

Logistic (Initial) / Assess (Respond)	Initial Transition				Respond Transition			
	Input	Output	Current State	Next State	Input	Output	Current State	Next State
Update / Assess Location	-	Entity State PDU	Idle	Idle	Entity State PDU	-	Idle	Assess Entity State
Assess Location / -	-	-	Assess Entity State	Idle	-	-	-	-
Move / Assess Location	move	Entity State PDU	Idle	Move	Entity State PDU	-	Idle	Assess Entity State
Assess Location / -	-	-	Assess Entity State	Idle	-	-	-	-
Stop / Assess Location	Stop	Entity State PDU	Move	Idle	Entity State PDU	-	Idle	Assess Entity State
Assess Location / -	-	-	Assess Entity State	Idle	-	-	-	-
Collision / Assess Collision	collision	Collision PDU	Move	Collision	Collision PDU	Collision PDU	Idle	Assess Collision
Assess Collision / Collision Reply	Collision PDU	Collision PDU	Assess Collision	Idle	Collision PDU	-	Idle	Assess Collision
Assess Collision / -	-	-	Assess Collision	Idle	-	-	-	-
move / assess collision	move	Entity State PDU	Collision	Move	Entity State PDU	-	Idle	Assess Entity State
stop / assess location	stop	Entity State PDU	Collision	Idle	Entity State PDU	-	Idle	Assess Entity State

Table 4.3 - Logistic / Assess

Fire (Initial) / Assess (Respond)	Initial Transition				Respond Transition			
	Input	Output	Current State	Next State	Input	Output	Current State	Next State
fire / Assess Fire	fire	Fire PDU	Idle	Fire	Fire PDU	-	Idle	Assess Fire
Assess Fire / -	-	-	Assess Fire	Idle	-	-	-	-
delay / Assess Detonation	-	Detonation PDU	Fire	Idle	Detonation PDU	-	Idle	Assess Detonation
Assess Detonation / -	-	Entity State PDU	Assess Detonation	Idle	-	-	-	-

Figure 4.4 - Fire / Assess

Table 4.1 shows the Initial transitions and the corresponding responding transitions for the resupply services available in the DIS protocol. Resupply services may include resupplying for fuel or munitions. Using Table 4.1 and following the transitions, a list of the Initial/Respond transitions can be obtained and checked against the DIS standard.

For example, an entity may request resupplies by sending a Service Request PDU. The resupply receiver, will change state, transitioning from Idle to the Requesting State and will respond by sending a Resupply Offer PDU. The supplier entity will then transition from Idle to the Offering State. This set of transitions can be traced by observing the first row of Table 4.1. Executing the DIS specification in single-step mode allows the tester to verify the steps and observe the messages in the queues.

When the Resupply Offer PDU is transmitted by the supplier, the resupply receiver receives the PDU and transitions from the Requesting State to Receiving State. This set of transitions are described in the second row of Table 4.1.

As can be seen from the above scenario, a set of initial and response transitions can be observed by following the test script in Table 4.1. Tables 4.2, 4.3 and 4.4 are similar to Table 4.1. Table 4.2 shows the initial transition and its corresponding responding transitions for the repair services defined in the DIS protocols. Tables 4.3 and 4.4 show the transitions during fire and collision scenarios with the Assess Module playing the role of the responder.

4.1.2 Inopportune Testing

The inopportune tests identify shortcomings in the protocol due to network failure and tests the protocols response in such situations.

For the inopportune tests, IST has developed test cases dealing with recovery from PDU loss. In all cases, the protocol recovered by a timeout mechanism. In test cases identified, DIS would simply discard the PDUs that do not apply to the state of the transition model.

For example, in row 1 of Table 4.1, if the Service Request PDU sent by the requester is lost , the responder remains in the Idle state. The requester would repeat the request by sending a Service Request PDU every 5 seconds until either it receives a response for its request or it gives up the request. A lost Service Request PDU has no great consequences.

Another example is row 4 of Table 4.2. If three or more consecutive Service Request PDUs are lost (e.g., the requester dropped out from the network), then the repairer would timeout after 12 seconds and cancel the repair activity by returning to the Idle state. This event is not expected to happen frequently, but if it does happen, it would be handled properly by the DIS protocol.

4.2 Testing Results

The tables presented in the previous section show a detailed testing procedure for the DIS Protocols. The tables are arranged into an Initiator/Responder analysis. All the initiated transitions were verified with the corresponding responder transitions. The experience gained during the testing phase was used to enhance the quality of the specification and the testing process. If an ambiguity in the specification was observed, the specification would be suitably modified.

As a result of the testing phase, two inconsistencies were found in the Logistic Module. In the first case, the Cancel Request transition in the resupply receiver module (Figure 5.1) did not occur. This was because no interaction (Resupply Cancel PDU) originated from the supplier module to allow the receiver's Cancel Request transition to take place. The specification was modified to correct this inconsistency. With the addition made to the DIS specification, the testing procedures were changed to reflect this modification.

For the second case, the Transfer Cancel transition in the receiver resupply module (Figure 5.1) would not occur. As with the first case, the reason was that there was no interaction (Resupply Cancel PDU) originated from the supplier module to allow the receiver's Transfer Cancel transition to take place. Again, the specification was modified to handle this deficiency. The deficiencies and their corrections are are explained in detail in section 5.

The new state transition models for the supplier/receiver resupply module is shown in section 5. Further explanations are given in section 5 in regard to the new state transitions. The Fire and Assess state transition models were checked based on a number of assumptions on the standard. and no ambiguities were found. These assumptions are listed in Section 3.3 in this report.

The DIS standard has provisions for the loss of packets in the case of the repair/resupply activities due to network failures. Because of this, the Inopportune test has shown those cases were handled properly. In the current DIS standard, a timeout mechanism is incorporated and has been verified to be adequate.

5 RECOMMENDATIONS TO THE DIS WORKSHOP

Title:

Specification of DIS in the Estelle Formal Description Technique for High Level Protocol Machine Validation.

Introduction:

The goal of the DIS workshops is to gather expertise from the simulation community to develop standards that allow defense simulations (existing and future) to interact through simulation networking, realizing a realistic battle training environment. This is a shift from the traditional single isolated training device approach.

The developed standards must be tested for validity and interoperability. It is therefore recommended that the initial DIS standard release be validated.

The Institute for Simulation and Training at University of Central Florida has experience in many areas of computer and simulation networking. In its effort to assess the usability of existing OSI protocols in real-time simulations, IST has developed a formal specification of the DIS draft standard using the Estelle Formal Description Technique (FDT) and the PET-DINGO compiler. This task has identified areas in the DIS standard that need to be modified or clarified.

Estelle is a standard FDT developed in 1989 by the International Organization for Standardization (ISO) to specify distributed concurrent information processing systems using a Pascal like language.

The National Institute of Standards and Technology (NIST) developed the PET-DINGO compiler, which is a protocol prototyping tool that accepts an Estelle specification and produces a runtime environment simulating the specified protocol behavior. PET-DINGO was developed to familiarize the scientific community with the Estelle specification language and hence promote its popularity.

Recommendations for Addition to the DIS Standard:

The current DIS protocol's resupply event is derived almost entirely from the SIMNET protocol, but with some additions in the receiver's state transition model to allow more possibilities for supplier cancellation. However, the necessary transitions on the supplier side of the protocol associated with such additions are missing. In order to establish a correct transition model, there is a need for additions to the supplier's state transition model.

Figure 5.1 shows the resupply receiver's state transition model. The dotted lines show the transitions expected by the supplier interaction, but not provided in the supplier's state transition model. Figure 5.2 shows the resupply supplier's state transition model with the proposed addition shown by the dotted line.

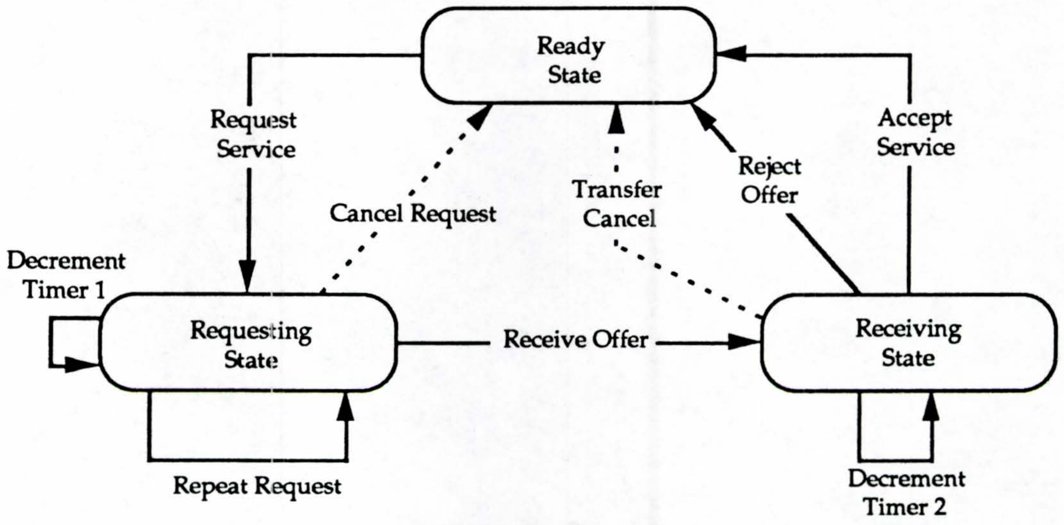


Figure 5.1 - Resupply Receiver State Transition Model

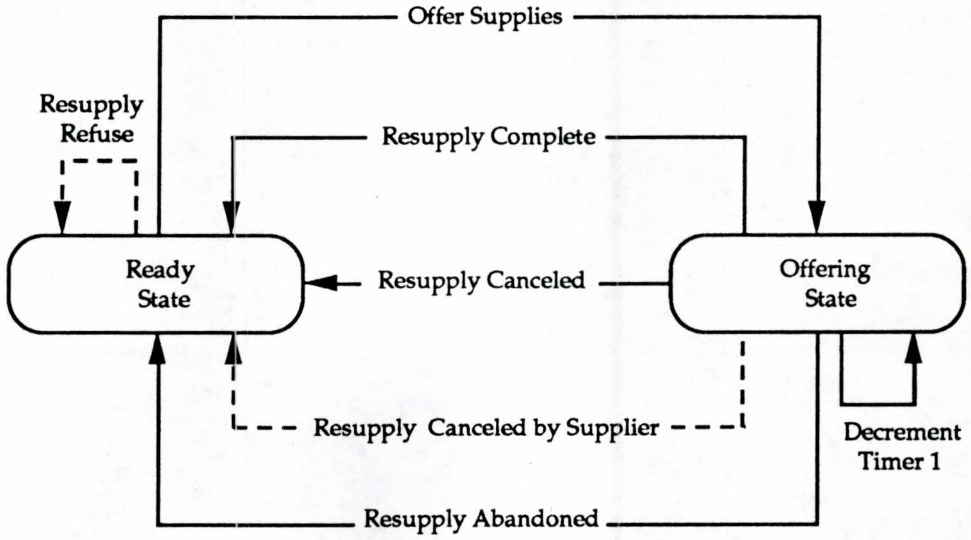


Figure 5.2 - Resupply Supplier State Transition Model

The proposed additions to the DIS standard are:

Resupply Refuse:	When a Service Request PDU is received, the supplier may refuse the request and issue a Request Cancel PDU. The entity shall remain in the Ready State. (Rational: The supplier's Resupply Refuse transition causes a receiver's Cancel Request transition.)
Resupply Canceled: by supplier	When conditions for resupply are no longer met, a Resupply Cancel PDU Shall be issued, Timer 1 shall be canceled, and the entity shall proceed from the Offering state to the Ready state. (Rational: The supplier's Resupply Canceled by Supplier transition causes a receiver's Transfer Cancel transition.)

6 CONCLUSION

This work was successful in specifying the DIS standard within the context of a formal description technique. However, because of the nature of the current standard, several assumptions and restrictions were imposed on the entity responses defined in the PDUs. Without such assumptions and restrictions, this work would be cumbersome and unnecessarily extensive.

The current standard is not fault tolerant, i.e., the protocol can present misbehavior caused by network failures. However, the protocol works well in an environment of low network failure rate (e.g., SIMNET packet loss rate is in the order of one in one million).

APPENDIX A

Estelle Overview

Estelle FDT defines Modules with their interaction points and connections. Each module may have an extended state transition model. Figure A.1 shows a possible structure for an Estelle specification:

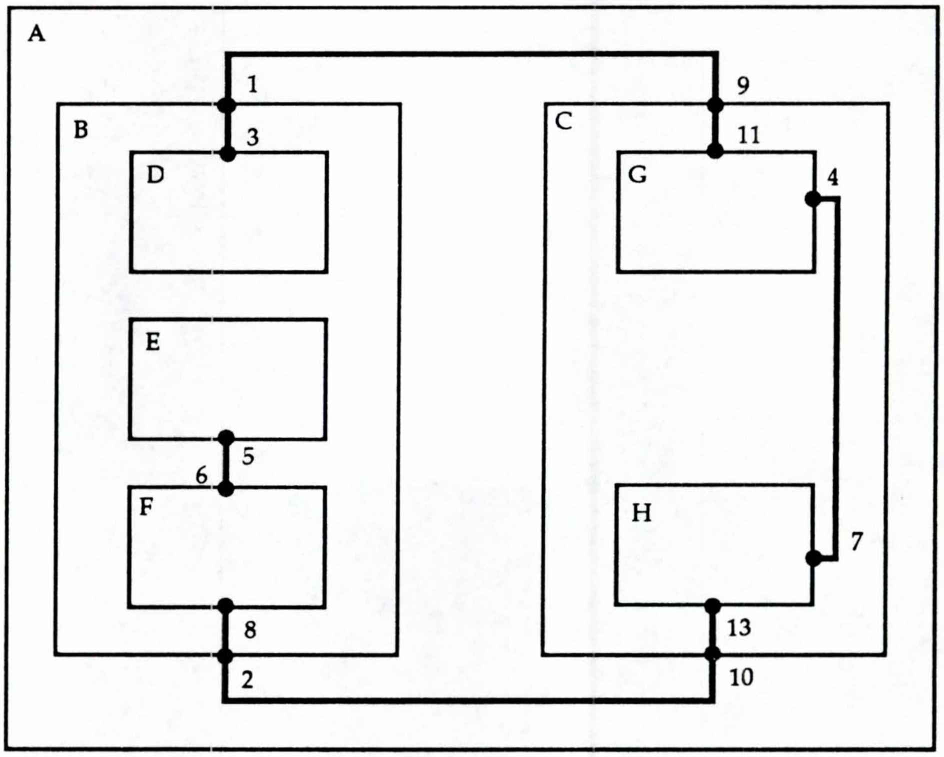


Figure A.1 - Estelle Structure

Figure A.1 presents three levels of module hierarchy. The first level is module A, the second level consists of modules B and C, and the third level consists of modules D, E, F, G and H.

The dots numbered from 1 to 11 are the interaction points associated with the modules. The thicker lines represent four channels and four attachments linking the modules. The channels are those connecting interaction points of sibling modules (modules belonging to the same level of hierarchy and within the same parent module), such as the channel connecting 1 to 9 and the channel connecting 4 to 7. The attachments join interaction points from modules belonging to different levels in the hierarchy, such as the attachment 1 to 3 and the attachment 13 to 10.

The collection of channels and attachments can establish end-to-end linkage between modules. For instance, the 1 to 3 attachment combined with the 1 to 9 connection and 9 to 11 attachment yields a connection from module D to module G.

In the Estelle specification, the channels, but not the attachments, must be defined at the beginning of the specification. Attachments can be defined at the time of attachment.

An Extended State Transition Model can be represented graphically as follows:

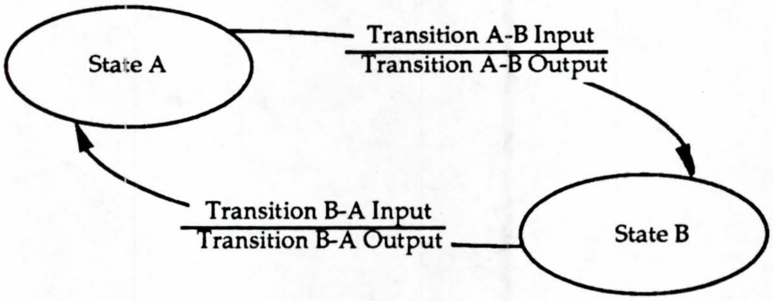


Figure A.2 - Example of an Extended State Transition Model

Figure A.2 shows an ESTM with two states A and B, and two transitions A to B (A-B) and B to A (B-A). An initial input (event) to the model can cause the transition A-B to take place possibly generating an output. The same applies to the transition B-A. There is no limit on the number of states allowed for each ESTM, and a state can have any state in the model, even itself, as its next state.

The interaction transmitted from one module to the other can be defined as a variable of any type in the Pascal language. It can be a data structure or simply a name with no type associated to it.

The structure of an Estelle specification is as follows:

Description	Estelle Keyword
A. Module Header	
1. Name, class, parameters	module
2. Interaction points	ip
3. exported variables	export
4. termination	end;
B. Module Body	
1. Declaration part	
a. body name	body
b. constant	const
c. types	type
d. variables	var
e. procedures/functions	proc/func
f. channels	channel
g. children modules	module, body
h. module variables	modvar
i. states	state, state-sets

Description	Estelle Keyword
2. Initialization part	
a. keyword	initialize
b. initial state	to
c. block/instance creation	begin
i. initialize	init (release)
ii. connect	connect (disconnect)
iii. attach	attach (detach)
d. termination	end;
3. Transition declaration	
a. start	begin
b. transitions	trans
i. priority	priority
ii. when	when
iii. provided	provided
iv. delay	delay
v. from X to Y	from to
c. termination	end;
4. Termination	end.

APPENDIX B

Estelle Specification of DIS Standard

```
{************************************************}
{                                                }
{     Institute for Simulation and Training      }
{         University of Central Florida          }
{                                                }
{ Name:...........dis.e                          }
{                                                }
{ Purpose:........to define the services and     }
{                 protocols of the Distributed   }
{                 Interactive Simulation (DIS)   }
{                 using Estelle; a Formal Des-   }
{                 cription Technique (FDT).      }
{                                                }
{ Author:.........David T. Shen                  }
{                                                }
{ Date:...........18 February 1992               }
{                                                }
{************************************************}

specification DIS;

default common queue;

{ specification of the Distributed Interactive Simulation}

{
                -------------
               |   driver    |
                -------------
                      |
                      |
                      |
  --------------------|--------------------
 |core                |                    |
 |                    |                    |
 |              -------------              |
 |             |  splitter   |             |
 |              -------------              |
 |                 |     |                 |
 |        ---------      |                 |
 |       |               |                 |
 |  -----------    ---------    ---------- |
 | |logistics|    |  fire  |   | assess  | |
 |  -----------    ---------    ---------- |
 |       |               |          |      |
 |        -------        |    ------       |
 |               |       |   |             |
 |             --------------              |
 |            |  combiner    |             |
 |             --------------              |
  --------------------|--------------------
                      |
                      |
             -------------   -------------
            |  network    |---| netswitch |
             -------------   -------------

}

timescale second;

type
```

```
  request_type = ( resupply, repair);
  id_type = integer;

{ service request is for resupply or repair }

  ServRequestPDU_type = record
    service : request_type;
  end;

{ collision id in the collision PDU for event identification}

  CollisionPDU_type = record
    id : id_type;
  end;

{ resupply time is arbitrarily set to be 15 seconds }

const resupply_time = 15;

{ channel definition for connecting the driver and core }

channel driver_upcore (driver, core);
  by driver:
    repair_request_signal;
    repair_cancel_signal;
    repair_complete_signal;
    resupply_request_signal;
    resupply_cancel_signal;
    move_signal;
    stop_signal;
    collision_signal;
    fire_signal;


{ channel definition for connecting the splitter and logistic }

channel splitter_logistic (splitter, logistic);
  by splitter:
    repair_request_signal;
    repair_cancel_signal;
    repair_complete_signal;
    resupply_request_signal;
    resupply_cancel_signal;
    move_signal;
    stop_signal;
    collision_signal;


{ channel definition for connecting the splitter and fire }

channel splitter_fire (splitter, fire);
  by splitter:
    fire_signal;


{ channel definition for connecting the logistic and combiner }

channel logistic_combiner (logistic, combiner);
  by logistic, combiner:
    ServRequestPDU(serv_data: ServRequestPDU_type);
    RepairResponsePDU;
    RepairCompletePDU;
    ResupplyOfferPDU;
    ResupplyCancelPDU;
    ResupplyReceivedPDU;
    collision_signal;

  by logistic:
    EntityStatePDU;
    CollisionPDU(event: collisionPDU_type);


{ channel definition for connecting the fire and combiner }

channel fire_combiner (fire, combiner);
  by fire:
    FirePDU;
    DetonationPDU;


{ channel definition for connecting the assess and combiner }

channel assess_combiner (assess, combiner);
  by assess, combiner:
    EntityStatePDU;
    CollisionPDU(event : collisionPDU_type);
    collision_signal;

  by combiner:
    DetonationPDU;
    FirePDU;


{ channel definition for connecting the core and network }

channel lowcore_network (core, network);
  by core, network:
    ServRequestPDU(serv_data: ServRequestPDU_type);
    RepairResponsePDU;
    RepairCompletePDU;
    ResupplyOfferPDU;
    ResupplyCancelPDU;
    ResupplyReceivedPDU;
    EntityStatePDU;
    CollisionPDU(event : collisionPDU_type);
    FirePDU;
    DetonationPDU;


{ channel definition for connecting the network and netswitch }

channel net_action (switch, network);
  by switch:
    dropPDU_signal;
    sendPDU_signal;


{ function system_time is found external to this program }

function system_time: integer; primitive;
```

```
{ declaration part for driver module }

module driverhead systemprocess;
  ip A:driver_upcore(driver) ;
end;

body driverbody for driverhead;

  procedure initPlayer; primitive;
  procedure playerDriver; primitive;

  initialize
    begin
      initPlayer
    end;

  trans
    begin
      playerDriver
    end;
  end;


{ declaration part for netswitch module }

module netswitchhead systemprocess;
  ip L: net_action(switch) individual queue;
end;

body netswitchbody for netswitchhead;

  procedure initPlayerN; primitive;
  procedure playerDriverN; primitive;

  initialize
    begin
      initPlayerN
    end;

  trans
    begin
      playerDriverN
    end;
end;


{ declaration part for core module }

module corehead systemprocess;
  ip A:driver_upcore(core) ;
     J:lowcore_network(core) ;
end;

body corebody for corehead;

  module splitterhead process;
    ip B:driver_upcore(core) ;
       C:splitter_logistic(splitter) ;
      D:splitter_fire(splitter) ;
 end;

 body splitterbody for splitterhead;
   trans

          { the following signals received from the driver, i.e.,
               repair_request_signal,
               repair_cancel_signal,
               repair_complete_signal,
               resupply_request_signal,
               resupply_cancel_signal,
               collision_signal,
               move_signal,
               stop_signal
             will be directed to the logistic module.    }

     when B.repair_request_signal
       begin
         output C.repair_request_signal;
       end;
     when B.repair_cancel_signal
       begin
         output C.repair_cancel_signal;
       end;
     when B.repair_complete_signal
       begin
         output C.repair_complete_signal;
       end;
     when B.resupply_request_signal
       begin
         output C.resupply_request_signal;
       end;
     when B.resupply_cancel_signal
       begin
         output C.resupply_cancel_signal;
       end;
     when B.collision_signal
       begin
         output C.collision_signal;
       end;
     when B.move_signal
       begin
         output C.move_signal;
       end;
     when B.stop_signal
       begin
         output C.stop_signal;
       end;

          { the fire_signal received from the driver will be }
          { directed to the fire module.                     }

     when B.fire_signal
       begin
         output D.fire_signal;
       end;
 end;
```

```
{ declaration part for logistic module }

 module logistichead process;
   ip C:splitter_logistic(logistic) ;
      F:logistic_combiner(logistic) ;
 end;{logistichaed}

 body logisticbody for logistichead;

   { different states when logistics are being performed }

   state IDLE, REPAIR_REQUEST, REPAIR_OFFERING, RESUPPLY_REQUEST,
         RESUPPLY_RECEIVING, RESUPPLY_OFFERING, MOVE, COLLISION;

   var initial_t, setup_t : integer;
       local_collision_id : integer;
       serv_data : ServRequestPDU_type;
       event : collisionPDU_type;

   {  procedure set_timer will work together with function timeout, }
   {  where set_timer will initialize the variables initial_t with  }
   {  the system's time and setup_t with the given countdown time   }

   procedure set_timer(t: integer);
   begin
     initial_t := system_time;
     setup_t := t;
   end; {set_timer}

   {  Function timeout will test for the timer expiration.          }

   function timeout : boolean;
   var delta, current_t: integer;
   begin
     current_t := system_time;
     if current_t < initial_t then
       delta := 60
     else
       delta := 0;
     if current_t + delta - initial_t < setup_t then
       timeout := false
     else
       timeout := true;
   end;{timeout}

   initialize
     to IDLE
       begin
         local_collision_id := 1;
       end;

   { resupply state transitions for resupply receiver }

   trans

     { request service for receiver behavior during resupply }
     when C.resupply_request_signal
       from IDLE to RESUPPLY_REQUEST
         begin
           serv_data.service:=resupply;
```

```
            output F.ServRequestPDU(serv_data);
            { requesting interval is set to 5 seconds for Timer1 }
            set_timer(5);
          end;

      { cancel request for receiver behavior during resupply }
      when F.ResupplyCancelPDU
        from RESUPPLY_REQUEST to IDLE
          begin
          end;

      { receive offer for receiver behavior during resupply }
      when F.ResupplyOfferPDU
        from RESUPPLY_REQUEST to RESUPPLY_RECEIVING
          begin
            { resupply_time is arbitrarily set to 15 seconds for Timer 2 }
            set_timer(resupply_time);
          end;

      { reject offer for receiver behavior during resupply }
      when C.resupply_cancel_signal
        from RESUPPLY_RECEIVING to IDLE
          begin
            output F.ResupplyCancelPDU;
          end;

      { transferred canceled for receiver behavior during resupply }
      when F.ResupplyCancelPDU
        from RESUPPLY_RECEIVING to IDLE
          begin
          end;

      trans
      { accept service for receiver behavior during resupply }
      from RESUPPLY_RECEIVING to IDLE
        provided timeout
          begin
            output F.ResupplyReceivedPDU;
          end;

      { not timeout yet for Timer 2, so it remains in the same state }
      from RESUPPLY_RECEIVING to same
        provided otherwise
          begin
          end;

    trans
    { if Timer 1 expires, the Service Request PDU shall be reissued }
    { and timer shall be reset for five seconds.                    }
      from RESUPPLY_REQUEST to same
        begin
          if timeout then
            begin
              serv_data.service := resupply;
              output F.ServRequestPDU(serv_data);
              set_timer(5);
            end;
        end;

    { repair state transitions for repair receiver }
```

```
trans

  { repair request for receiver behavior during repair }
  when C.repair_request_signal
    from IDLE to REPAIR_REQUEST
      begin
        serv_data.service:=repair;
        output F.ServRequestPDU(serv_data);
        { requesting interval is set to 5 seconds for Timer1 }
        set_timer(5);
      end;
    from REPAIR_REQUEST to same
      begin
      end;

  { repair complete for receiver behavior during repair }
  when F.RepairCompletePDU
    from REPAIR_REQUEST to IDLE
      begin
        output F.RepairResponsePDU;
      end;
    from IDLE to same
      begin
      end;

  { cancel request for receiver behavior during repair }
  when C.repair_cancel_signal
    from REPAIR_REQUEST to IDLE
      begin
      end;
    from IDLE to same
      begin
      end;

{ if Timer 1 expires, the Service Request PDU shall be reissued }
{ and timer shall be reset for five seconds.                    }
trans
  from REPAIR_REQUEST to same
    begin
    if timeout then
      begin
        serv_data.service:=repair;
        output F.ServRequestPDU(serv_data);
        set_timer(5);
      end;
    end;

{ resupply state transitions for resupply supplier }
trans
  { offer supplies for supplier behavior during resupply,         }
  { Resupply Offer PDU is issued and timer is set to one minute. }
  when F.ServRequestPDU
    provided serv_data.service = resupply
      from IDLE to RESUPPLY_OFFERING
        begin
          output F.ResupplyOfferPDU;
          set_timer(59);
        end;

  { resupply complete for supplier behavior during resupply }
  when F.ResupplyReceivedPDU
    from RESUPPLY_OFFERING to IDLE
      begin
      end;

  { resupply canceled for supplier behavior during resupply }
  when F.ResupplyCancelPDU
    from RESUPPLY_OFFERING to IDLE
      begin
      end;

  { if resupply_cancel_signal is sent by driver, supplier behavior }
  { will remain at IDLE state if it was at IDLE originally, or it  }
  { will transition from RESUPPLY_OFFERING state to IDLE if it was }
  { originally at RESUPPLY_OFFERING state.                         }
  when C.resupply_cancel_signal
    from IDLE to same
      begin
        output F.ResupplyCancelPDU;
      end;
  when C.resupply_cancel_signal
    from RESUPPLY_OFFERING to IDLE
      begin
        output F.ResupplyCancelPDU;
      end;

trans
  { if Timer 1 expires, the transfer shall be abandoned or else }
  { remain in RESUPPLY_OFFERING state.                          }
  from RESUPPLY_OFFERING to IDLE
    provided timeout
      begin
      end;
  from RESUPPLY_OFFERING to same
    provided otherwise
      begin
      end;

{ repair state transitions for repair supplier }
trans
  { repair request for supplier behavior during repair }
  when F.ServRequestPDU
    provided serv_data.service = repair
      from IDLE to REPAIR_OFFERING
        begin
        set_timer(12);
        end;
      from REPAIR_OFFERING to same
        begin
          set_timer(12);
        end;

  { repair response for supplier behavior during repair }
  when F.RepairResponsePDU
    from IDLE to same
      begin
      end;

  { if repair_complete signal is sent by driver, supplier behavior }
  { will remain at IDLE state if it was at IDLE originally, or it  }
```

```
    { will transition from REPAIR_OFFERING state to IDLE if it was     }
    { originally at REPAIR_OFFERING state.                             }
    when C.repair_complete_signal
      from REPAIR_OFFERING to IDLE
        begin
          output F.RepairCompletePDU
        end;
      from IDLE to same
        begin
        end;

  trans
    { if Timer 1 expires, the transfer shall be abandoned or else }
    { remain in REPAIR_OFFERING state.                            }
    from REPAIR_OFFERING to IDLE
      provided timeout
        begin
        end;
    from REPAIR_OFFERING to same
      provided otherwise
        begin
        end;

  { entity behavior during actions: moving, stopping, and colliding }
  trans
    when C.move_signal
      from IDLE to MOVE
        begin
          output F.EntityStatePDU;
        end;
      from COLLISION to MOVE
        begin
          output F.EntityStatePDU;
        end;
      from MOVE to same
        begin
        end;
    when C.stop_signal
      from MOVE to IDLE
        begin
          output F.EntityStatePDU;
        end;
      from COLLISION to IDLE
        begin
          output F.EntityStatePDU;
        end;
      from IDLE to same
        begin
        end;
    when C.collision_signal
      from MOVE to COLLISION
        begin
          event.id := local_collision_id;
          output F.CollisionPDU(event);
          output F.collision_signal;
          local_collision_id := local_collision_id + 1;
        end;
      from IDLE to same
        begin
        end;
      from COLLISION to same
        begin
        end;
    when F.collision_signal
      begin
        local_collision_id := local_collision_id + 1;
      end;

  { entity will issue Entity State (ES) PDU every 4 seconds in IDLE state, }
  { or issue this PDU based on the high fidelity model in the MOVE state.  }
  { For the specification sake, it is assume that the ES PDU is issued     }
  { every second in the MOVE state.                                        }
  trans
    from IDLE to same
    delay(4)
      begin
        output F.EntityStatePDU;
      end;
    from MOVE to same
    delay(1)
      begin
        output F.EntityStatePDU;
      end;

 end;{body logisticbody}

{ declaration part for fire module }

 module firehead process;
   ip D:splitter_fire(fire) ;
      G:fire_combiner(fire) ;
 end;{firehead}

 body firebody for firehead;
   state READY, FIRE;
   initialize to READY
     begin
     end;

   { entity behavior during fire }
   trans
     when D.fire_signal
       from READY to FIRE
         begin
           output G.FirePDU;
         end;

   { delay 2 seconds is arbitrary, this value just gives a delay }
   { before a Detonation PDU is issued by the entity.            }
   trans
     from FIRE to READY
       delay(2)
       begin
         output G.DetonationPDU;
       end;

 end;{firebody}


 { declaration part for assess module }
```

```
module assesshead process;
  ip H:assess_combiner(assess) ;
end;{assesshead}

body assessbody for assesshead;
  state IDLE, ASSESS_DETONATION, ASSESS_FIRE,
        ASSESS_ENTITY_STATE, ASSESS_COLLISION;

  var local_collision_id : integer;

  initialize to IDLE
    begin
      local_collision_id := 1;
    end;

  { entity behavior when it receives the following PDUs: }
  {      EntityStatePDU                                  }
  {      FirePDU                                         }
  {      DetonationPDU                                   }
  {      CollisionPDU                                    }

  trans
    when H.EntityStatePDU
      from IDLE to ASSESS_ENTITY_STATE
        begin
        end;
    when H.FirePDU
      from IDLE to ASSESS_FIRE
        begin
        end;
    when H.DetonationPDU
      from IDLE to ASSESS_DETONATION
        begin
        end;
    when H.CollisionPDU
      provided event.id = local_collision_id
        from IDLE to ASSESS_COLLISION
          begin
            output H.CollisionPDU(event);
            output H.collision_signal;
            local_collision_id := local_collision_id + 1;
          end;
      provided otherwise
        begin
        end;
    when H.collision_signal
      begin
        local_collision_id := local_collision_id + 1;
      end;

  trans
    from ASSESS_ENTITY_STATE to IDLE
      begin
      end;
    from ASSESS_FIRE to IDLE
      begin
      end;
    from ASSESS_DETONATION to IDLE
      begin
```

```
        output H.EntityStatePDU;
      end;
    from ASSESS_COLLISION to IDLE
      begin
      end;

 end;{assessbody}


{ declaration part for combiner module }

module combinerhead process;
  ip F:logistic_combiner(combiner) ;
     G:fire_combiner(combiner) ;
     H:assess_combiner(combiner) ;
     I:lowcore_network(core) ;
end;


{ the following PDUs received from the logistic module, i.e.,
           ServRequestPDU,
           RepairResponsePDU,
           RepairCompletePDU,
           ResupplyOfferPDU,
           ResupplyCancelPDU,
           ResupplyReceivedPDU,
           EntityStatePDU,
           CollisionPDU,
   will be directed to the network module.    }

body combinerbody for combinerhead;
  trans
    when F.ServRequestPDU
      begin
        output I.ServRequestPDU(serv_data);
      end;
    when F.RepairResponsePDU
      begin
        output I.RepairResponsePDU;
      end;
    when F.RepairCompletePDU
      begin
        output I.RepairCompletePDU;
      end;
    when F.ResupplyOfferPDU
      begin
        output I.ResupplyOfferPDU;
      end;
    when F.ResupplyCancelPDU
      begin
        output I.ResupplyCancelPDU;
      end;
    when F.ResupplyReceivedPDU
      begin
        output I.ResupplyReceivedPDU;
      end;
    when F.EntityStatePDU
      begin
        output I.EntityStatePDU;
      end;
     when F.CollisionPDU
       begin
         output I.CollisionPDU(event);
       end;


{ the following PDUs received from the network module, i.e.,
           ServRequestPDU,
           RepairResponsePDU,
           RepairCompletePDU,
           ResupplyOfferPDU,
           ResupplyCancelPDU,
           ResupplyReceivedPDU,
           EntityStatePDU,
           CollisionPDU,
    will be directed to the logistic module.    }


     when I.ServRequestPDU
       begin
         output F.ServRequestPDU(serv_data);
       end;
     when I.RepairResponsePDU
       begin
         output F.RepairResponsePDU;
       end;
     when I.RepairCompletePDU
       begin
         output F.RepairCompletePDU;
       end;
     when I.ResupplyOfferPDU
       begin
         output F.ResupplyOfferPDU;
       end;
     when I.ResupplyCancelPDU
       begin
         output F.ResupplyCancelPDU;
       end;
     when I.ResupplyReceivedPDU
       begin
         output F.ResupplyReceivedPDU;
       end;
     when H.collision_signal
       begin
         output F.collision_signal;
       end;


    { the following PDUs received from the fire module, i.e.,
            FirePDU,
            DetonationPDU,
    will be directed to the network module.    }

     when G.FirePDU
       begin
         output I.FirePDU;
       end;
     when G.DetonationPDU
       begin
         output I.DetonationPDU;
```

```
      end;

    { EntityStatePDU received from assess module, }
    { will be directed to network module.         }

    when H.EntityStatePDU
      begin
        output I.EntityStatePDU;
      end;
    when H.CollisionPDU
      begin
        output I.CollisionPDU(event);
      end;

   { the following PDUs received from the network module, i.e.,
            FirePDU,
            EntityStatePDU,
            DetonationPDU,
   will be directed to the assess module.    }

    when I.FirePDU
      begin
        output H.FirePDU;
      end;
    when I.EntityStatePDU
      begin
        output H.EntityStatePDU;
      end;
    when I.DetonationPDU
      begin
        output H.DetonationPDU;
      end;
    when I.CollisionPDU
      begin
        output H.CollisionPDU(event);
      end;
    when F.collision_signal
      begin
        output H.collision_signal;
      end;

end;{body combinerbody}

modvar
  splitter : splitterhead;
  logistic : logistichead;
  fire : firehead;
  assess : assesshead;
  combiner : combinerhead;

initialize
  begin
    init splitter with splitterbody;
    init logistic with logisticbody;
    init fire with firebody;
    init assess with assessbody;
    init combiner with combinerbody;

    attach A to splitter.B;
    connect splitter.C to logistic.C;
```

```
      connect splitter.D to fire.D;
      connect logistic.F to combiner.F;
      connect fire.G to combiner.G;
      connect assess.H to combiner.H;
      attach J to combiner.I;
  end;

end;{body corebody}


{ declaration for network module }

module networkhead systemprocess;
  ip J:lowcore_network(network) individual queue;
     K:lowcore_network(network) individual queue;
     L:net_action(network) individual queue;
end;{networkhead}

body networkbody for networkhead;
  state SEND, DROP;

  initialize to SEND
    begin
    end;

  { network behavior either send or drop the PDU }
  trans
    when
      J.ServRequestPDU
        from SEND to same
          begin
            output K.ServRequestPDU(serv_data);
          end;
        from DROP to same
          begin
          end;
    when
      J.RepairResponsePDU
        from SEND to same
          begin
            output K.RepairResponsePDU;
          end;
        from DROP to same
          begin
          end;
    when
      J.RepairCompletePDU
        from SEND to same
          begin
            output K.RepairCompletePDU;
          end;
        from DROP to same
          begin
          end;
    when
      J.ResupplyCancelPDU
        from SEND to same
          begin
            output K.ResupplyCancelPDU;
          end;
    from DROP to same
      begin
      end;
when
  J.ResupplyReceivedPDU
    from SEND to same
      begin
        output K.ResupplyReceivedPDU;
      end;
    from DROP to same
      begin
      end;
when
  J.ResupplyOfferPDU
    from SEND to same
      begin
        output K.ResupplyOfferPDU;
      end;
    from DROP to same
      begin
      end;
when
  K.ServRequestPDU
    from SEND to same
      begin
        output J.ServRequestPDU(serv_data);
      end;
    from DROP to same
      begin
      end;
when
  K.RepairResponsePDU
    from SEND to same
      begin
        output J.RepairResponsePDU;
      end;
    from DROP to same
      begin
      end;
when
  K.RepairCompletePDU
    from SEND to same
      begin
        output J.RepairCompletePDU;
      end;
    from DROP to same
      begin
      end;
when
  K.ResupplyCancelPDU
    from SEND to same
      begin
        output J.ResupplyCancelPDU;
      end;
    from DROP to same
      begin
      end;
when
  K.ResupplyReceivedPDU
    from SEND to same
```

```
        begin
          output J.ResupplyReceivedPDU;
        end;
      from DROP to same
        begin
        end;
when
  K.ResupplyOfferPDU
      from SEND to same
        begin
          output J.ResupplyOfferPDU;
        end;
      from DROP to same
        begin
        end;
when
  J.EntityStatePDU
      from SEND to same
        begin
          output K.EntityStatePDU;
        end;
      from DROP to same
        begin
        end;
when
  J.CollisionPDU
      from SEND to same
        begin
          output K.CollisionPDU(event);
        end;
      from DROP to same
        begin
        end;
when
  K.EntityStatePDU
      from SEND to same
        begin
          output J.EntityStatePDU;
        end;
      from DROP to same
        begin
        end;
when
  K.CollisionPDU
      from SEND to same
        begin
          output J.CollisionPDU(event);
        end;
      from DROP to same
        begin
        end;
when
  J.FirePDU
      from SEND to same
        begin
          output K.FirePDU;
        end;
      from DROP to same
        begin
        end;
```

```
    when
      K.FirePDU
        from SEND to same
          begin
            output J.FirePDU;
          end;
        from DROP to same
          begin
          end;
    when
      J.DetonationPDU
        from SEND to same
          begin
            output K.DetonationPDU;
          end;
        from DROP to same
          begin
          end;
    when
      K.DetonationPDU
        from SEND to same
          begin
            output J.DetonationPDU;
          end;
        from DROP to same
          begin
          end;
    when
      L.sendPDU_signal
        from SEND to same
          begin
          end;
        from DROP to SEND
          begin
          end;
    when
      L.dropPDU_signal
        from SEND to DROP
          begin
          end;
        from DROP to same
          begin
          end;
end;{body networkbody}

modvar
  driverA : driverhead;
  driverB : driverhead;
  coreA: corehead;
  coreB: corehead;
  network : networkhead;
  netswitch : netswitchhead;

initialize
  begin
    init driverA with driverbody;
    init driverB with driverbody {@"ibis"};
    init netswitch with netswitchbody;
    init coreA with corebody;
    init coreB with corebody {@"ibis"};
    init network with networkbody;

    connect driverA.A to coreA.A;
    connect coreA.J to network.J;
    connect network.K to coreB.J;
    connect coreB.A to driverB.A;
    connect netswitch.L to network.L;
end;

end.
```

APPENDIX C

PET-DINGO User Defined Routines and Makefile

User interface modules are specified in the DIS Estelle specification: the Driver module and the Network Switch module. These modules allow the user to interact with the run-time processes and to set specific scenarios to test the DIS protocol. The interface modules are purely for testing purposes and are not part of the DIS standard.

_Driver.c implements the Driver module in the specification. It allows the user to determine the action to be taken by the simulation entity at any time. The user causes the protocol residing in the associated processes to interact, modeling the behavior of the protocol described in the DIS standard. Examples of possible user actions include: request for repair, request for resupply, move, fire, or response for repair complete.

_Switch.c implements the Network Switch module in the specification. It allows the user to disrupt the network module at any instant, simulating a network failure. This module is aimed at testing the protocol under network failure conditions.

The Drivers **_Driver.c** and **_Switch.c** are written in both C and C++ languages. They supply the human interface.

Timer.c supplies a system clock reader used in the logistic activities.

The Makefile generated by DINGO compiles a dynamic model into a run-time environment.

```
/*
   _Driver.c

   function to create a pop up menu from a null terminated list
   of strings; the menu puts the number of the selected item in
   a variable also specified in the creation call
*/
extern "C" {
#include <stdio.h>

#include <X11/Xatom.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Box.h>
#include <X11/Xaw/Form.h>
#include <X11/Xaw/Label.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>

#include <X11/Xaw/Cardinals.h>

#include "rtwin.h"

/* custom interface of player module */

extern void playNext();

static void send( Widget w, XtPointer closure, XtPointer callData)
{
  playNext( *(int*) closure);
}

void playerInterface(   struct __MIwin* miwin)
{
  Widget sendPanel, button, button2, button3, button4, button5;
  static int x1=0;
  static int x2=1;
  static int x3=2;
  static int x4=3;
  static int x5=4;
  static int x6=5;
  static int x7=6;
  static int x8=7;
  static int x9=8;
  Arg args[10];
  sendPanel = XtCreateManagedWidget( "Send Panel", formWidgetClass,
    miwin->outer, NULL, ZERO);
  XawPanedSetMinMax( sendPanel, 35, 155);

  button = XtCreateManagedWidget( "Request Repair", commandWidgetClass,
    sendPanel, NULL, ZERO);
  XtAddCallback( button, XtNcallback, send, (XtPointer)&x1);

  XtSetArg(args[0], XtNfromHoriz, button);
  button = XtCreateManagedWidget( "Repair Cancel", commandWidgetClass,
    sendPanel, args, ONE);
  XtAddCallback( button, XtNcallback, send, (XtPointer)&x2);

  XtSetArg(args[0], XtNfromVert, button);
```

```
    button2 = XtCreateManagedWidget( "Request Resupply", commandWidgetClass,
      sendPanel, args, ONE);
    XtAddCallback( button2, XtNcallback, send, (XtPointer)&x3);

    XtSetArg(args[0], XtNfromVert, button);
    XtSetArg(args[1], XtNfromHoriz, button2);
    button = XtCreateManagedWidget( "Resupply Cancel", commandWidgetClass,
      sendPanel, args, TWO);
    XtAddCallback( button, XtNcallback, send, (XtPointer)&x4);

    XtSetArg(args[0], XtNfromVert, button2);
    button3 = XtCreateManagedWidget( "Repair Complete", commandWidgetClass,
      sendPanel, args, ONE);
    XtAddCallback( button3, XtNcallback, send, (XtPointer)&x5);

    XtSetArg(args[0], XtNfromVert, button3);
    button4 = XtCreateManagedWidget( "Move", commandWidgetClass,
      sendPanel, args, ONE);
    XtAddCallback( button4, XtNcallback, send, (XtPointer)&x6);

    XtSetArg(args[0], XtNfromVert, button3);
    XtSetArg(args[1], XtNfromHoriz, button4);
    button4 = XtCreateManagedWidget( "Stop", commandWidgetClass,
      sendPanel, args, TWO);
    XtAddCallback( button4, XtNcallback, send, (XtPointer)&x7);

    XtSetArg(args[0], XtNfromVert, button3);
    XtSetArg(args[1], XtNfromHoriz, button4);
    button5 = XtCreateManagedWidget( "Collision", commandWidgetClass,
      sendPanel, args, TWO);
    XtAddCallback( button5, XtNcallback, send, (XtPointer)&x9);

    XtSetArg(args[0], XtNfromVert, button5);
    button = XtCreateManagedWidget( "Fire", commandWidgetClass,
      sendPanel, args, ONE);
    XtAddCallback( button, XtNcallback, send, (XtPointer)&x8);
  }
  }
```

```
/*
   _Switch.c

   function to create a pop up menu from a null terminated list
   of strings; the menu puts the number of the selected item in
   a variable also specified in the creation call
*/
extern "C" {
#include <stdio.h>

#include <X11/Xatom.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

#include <X11/Xaw/Box.h>
#include <X11/Xaw/Form.h>
#include <X11/Xaw/Label.h>
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Paned.h>

#include <X11/Xaw/Cardinals.h>

#include "rtwin.h"

/* custom interface of player module */

extern void playNextN();

static void sendN( Widget w, XtPointer closure, XtPointer callData)
{
  playNextN( *(int*) closure);
}

void playerInterfaceN(   struct __MIwin* miwin)
{
  Widget sendPanel, button, button2;
  static int x1=0;
  static int x2=1;
  Arg args[10];
  sendPanel = XtCreateManagedWidget( "Send Panel", formWidgetClass,
    miwin->outer, NULL, ZERO);
  XawPanedSetMinMax( sendPanel, 35, 155);

  button = XtCreateManagedWidget( "Send PDU", commandWidgetClass,
    sendPanel, NULL, ZERO);
  XtAddCallback( button, XtNcallback, sendN, (XtPointer)&x1);
  setGreyWinBg( button);

  XtSetArg(args[0], XtNfromHoriz, button);
  button2 = XtCreateManagedWidget( "Drop PDU", commandWidgetClass,
    sendPanel, args, ONE);
  XtAddCallback( button2, XtNcallback, sendN, (XtPointer)&x2);
  setGreyWinBg( button2);

}
}
```

```
/*
  timer.c

  Routine to return system time
*/

  #include <stdio.h>
#include <time.h>

int _System_time( __MInstance* __MI)
{
struct tm *TIME();
return TIME()->tm_sec;
}

struct tm *TIME()
{
        struct tm *xx;
        long clk, ss;
        char *tt;
        ss=time(&clk);
        xx=localtime(&ss);
        return(xx);
}
```

```
#  Generated Makefile for Specification _DIS

#NOTES: 1. dependencies are incomplete with respect to .hxx files;
#          'make all' forces the entire regeneration of object files
#       2. the user should complete the makefile and rename it
#          to avoid losing changes made
#
#
#
# EDIT THIS:
# -------- LOGTRANS: log transitions when they are executed
# -------- LOCMODS: modules to run on this particular site
# -------- LIBPATH: path to dingo runtime library
# -------- XLIBDIR: path to X11 runtime libraries
# -------- INCLPATH: path to dingo runtime library header files
# -------- GNUINCLUDEDIR: path to GNU g++ library header files
# -------- WINS: do you want windows linked with all modules ?
# -------- WINTIMER: value of wait timer for window input
# -------- USERDEF: object code to be linked with every executable
# -------- CCXX: invokation of C++ compiler (to object code)
# -------- EXEC: invokation of C++ compiler to executable


LOGTRANS= -DLOGT


LOCMODS = _DIS _Corebody _Driverbody _Networkbody _Netswitchbody
LIBPATH= /usr3/estelle/petdingo/
XLIBDIR= /usr3/estelle/X11/lib/
XLIBINCL= /
MIWINDIR= /usr1/GNU/libg++-1.39.0/g++-include/
INCLPATH= /usr3/estelle/petdingo/
GNUINCLUDEDIR= /usr1/GNU/libg++-1.39.0/g++-include/
WINS= -DDEFWINS
WINTIMER = -DWINWAIT=1
USERDEF= _Driver.o _Switch.o _DIS.impl.o


CCXX=g++ -DGNU -w -c -I$(INCLPATH) -I$(GNUINCLUDEDIR) -I/usr/include
EXEC=g++ -DGNU -w -I$(INCLPATH) -I$(MIWINDIR)
WINLIBS= -L$(XLIBDIR) -lXaw -lXmu -lXt -lX11 -lXext
SYSDEP= -L$(LIBPATH) -lestrt -lestrtwin -lnetif_sock
LIB= $(USERDEF) _DIS.glob.o _DIS.a $(SYSDEP) $(WINLIBS) -lm -lg++

default: $(LOCMODS)

_DIS.a: _DIS.glob.o  _DIS.dummy _Driverbody.dummy _Netswitchbody.dummy _Corebody.dummy _Splitterbody.dummy _Logisticbo
        rm -f _DIS.a
        ar q _DIS.a _DIS.o _DIS.tc.o _DIS.ch.o
        ar q _DIS.a _Driverbody.o _Driverbody.tc.o _Driverbody.ch.o
        ar q _DIS.a _Netswitchbody.o _Netswitchbody.tc.o _Netswitchbody.ch.o
        ar q _DIS.a _Corebody.o _Corebody.tc.o _Corebody.ch.o
        ar q _DIS.a _Splitterbody.o _Splitterbody.tc.o _Splitterbody.ch.o
        ar q _DIS.a _Logisticbody.o _Logisticbody.tc.o _Logisticbody.ch.o
        ar q _DIS.a _Firebody.o _Firebody.tc.o _Firebody.ch.o
        ar q _DIS.a _Assessbody.o _Assessbody.tc.o _Assessbody.ch.o
        ar q _DIS.a _Combinerbody.o _Combinerbody.tc.o _Combinerbody.ch.o
        ar q _DIS.a _Networkbody.o _Networkbody.tc.o _Networkbody.ch.o
        ranlib _DIS.a
_DIS.glob.o: _DIS.glob.cxx _DIS.bindirs
```

```
	$(CCXX) -DOPTIMIZE _DIS.glob.cxx

_DIS.dummy: _DIS.tc.hxx _DIS.tc.cxx _DIS.ch.hxx _DIS.ch.cxx _DIS.cxx _DIS.hxx
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _DIS.tc.cxx _DIS.ch.cxx _DIS.cxx
	rm -f _DIS.dummy
	echo ' ' > _DIS.dummy

_Driverbody.dummy: _Driverbody.tc.hxx _Driverbody.tc.cxx _Driverbody.ch.hxx _Driverbody.ch.cxx _Driverbody.cxx _Driver
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Driverbody.tc.cxx _Driverbody.ch.cxx _Driverbody.cxx
	rm -f _Driverbody.dummy
	echo ' ' > _Driverbody.dummy

_Netswitchbody.dummy: _Netswitchbody.tc.hxx _Netswitchbody.tc.cxx _Netswitchbody.ch.hxx _Netswitchbody.ch.cxx _Netswit
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Netswitchbody.tc.cxx _Netswitchbody.ch.cxx _Netswitchbody.cxx
	rm -f _Netswitchbody.dummy
	echo ' ' > _Netswitchbody.dummy

_Corebody.dummy: _Corebody.tc.hxx _Corebody.tc.cxx _Corebody.ch.hxx _Corebody.ch.cxx _Corebody.cxx _Corebody.hxx
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Corebody.tc.cxx _Corebody.ch.cxx _Corebody.cxx
	rm -f _Corebody.dummy
	echo ' ' > _Corebody.dummy

_Splitterbody.dummy: _Splitterbody.tc.hxx _Splitterbody.tc.cxx _Splitterbody.ch.hxx _Splitterbody.ch.cxx _Splitterbody
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Splitterbody.tc.cxx _Splitterbody.ch.cxx _Splitterbody.cxx
	rm -f _Splitterbody.dummy
	echo ' ' > _Splitterbody.dummy

_Logisticbody.dummy: _Logisticbody.tc.hxx _Logisticbody.tc.cxx _Logisticbody.ch.hxx _Logisticbody.ch.cxx _Logisticbody
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Logisticbody.tc.cxx _Logisticbody.ch.cxx _Logisticbody.cxx
	rm -f _Logisticbody.dummy
	echo ' ' > _Logisticbody.dummy

_Firebody.dummy: _Firebody.tc.hxx _Firebody.tc.cxx _Firebody.ch.hxx _Firebody.ch.cxx _Firebody.cxx _Firebody.hxx
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Firebody.tc.cxx _Firebody.ch.cxx _Firebody.cxx
	rm -f _Firebody.dummy
	echo ' ' > _Firebody.dummy

_Assessbody.dummy: _Assessbody.tc.hxx _Assessbody.tc.cxx _Assessbody.ch.hxx _Assessbody.ch.cxx _Assessbody.cxx _Assess
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Assessbody.tc.cxx _Assessbody.ch.cxx _Assessbody.cxx
	rm -f _Assessbody.dummy
	echo ' ' > _Assessbody.dummy

_Combinerbody.dummy: _Combinerbody.tc.hxx _Combinerbody.tc.cxx _Combinerbody.ch.hxx _Combinerbody.ch.cxx _Combinerbody
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Combinerbody.tc.cxx _Combinerbody.ch.cxx _Combinerbody.cxx
	rm -f _Combinerbody.dummy
	echo ' ' > _Combinerbody.dummy

_Networkbody.dummy: _Networkbody.tc.hxx _Networkbody.tc.cxx _Networkbody.ch.hxx _Networkbody.ch.cxx _Networkbody.cxx _
	$(CCXX) -DOPTIMIZE $(LOGTRANS) _Networkbody.tc.cxx _Networkbody.ch.cxx _Networkbody.cxx
	rm -f _Networkbody.dummy
	echo ' ' > _Networkbody.dummy

_DIS: _DIS.a _DIS.main.cxx $(USERDEF)
	$(EXEC) $(WINS) $(WINTIMER) _DIS.main.cxx -o _DIS $(LIB)

_Driverbody: _DIS.a $(USERDEF) _Driverbody.main.cxx
	$(EXEC) $(WINS) $(WINTIMER) _Driverbody.main.cxx -o _Driverbody $(LIB)

_Netswitchbody: _DIS.a $(USERDEF) _Netswitchbody.main.cxx
	$(EXEC) $(WINS) $(WINTIMER) _Netswitchbody.main.cxx -o _Netswitchbody $(LIB)
```

```
_Corebody: _DIS.a $(USERDEF) _Corebody.main.cxx
	$(EXEC) $(WINS) $(WINTIMER) _Corebody.main.cxx -o _Corebody $(LIB)

_Networkbody: _DIS.a $(USERDEF) _Networkbody.main.cxx
	$(EXEC) $(WINS) $(WINTIMER) _Networkbody.main.cxx -o _Networkbody $(LIB)

_DIS.impl.o: _DIS.impl.hxx _DIS.tc.hxx _DIS.ch.hxx _DIS.impl.cxx
	$(CCXX) -DOPTIMIZE _DIS.impl.cxx

_Driver.o: _Driverbody.tc.hxx _Driver.c _DIS.tc.hxx _DIS.ch.hxx _Driverbody.ch.hxx
	$(CCXX) -DOPTIMIZE -I$(XLIBINCL) _Driver.c

_Switch.o: _Netswitchbody.tc.hxx _Switch.c _DIS.tc.hxx _DIS.ch.hxx _Netswitchbody.ch.hxx
	$(CCXX) -DOPTIMIZE -I$(XLIBINCL) _Switch.c

all: clean $(LOCMODS)

clean:
	rm -f *.dummy
	rm -f _DIS.glob.o
```

APPENDIX D

Listing of PET-DINGO Generated Files from DIS Specification

```
DIS.symb                _DIS.hxx                _Netswitchbody.ch.cxx
_Assessbody.ch.cxx      _DIS.impl.hxx           _Netswitchbody.ch.hxx
_Assessbody.ch.hxx      _DIS.main.cxx           _Netswitchbody.cxx
_Assessbody.cxx         _DIS.make.tmpl          _Netswitchbody.hxx
_Assessbody.hxx         _DIS.tc.cxx             _Netswitchbody.main.cxx
_Assessbody.tc.cxx      _DIS.tc.hxx             _Netswitchbody.tc.cxx
_Assessbody.tc.hxx      _Driverbody.ch.cxx      _Netswitchbody.tc.hxx
_Combinerbody.ch.cxx    _Driverbody.ch.hxx      _Networkbody.ch.cxx
_Combinerbody.ch.hxx    _Driverbody.cxx         _Networkbody.ch.hxx
_Combinerbody.cxx       _Driverbody.hxx         _Networkbody.cxx
_Combinerbody.hxx       _Driverbody.main.cxx    _Networkbody.hxx
_Combinerbody.tc.cxx    _Driverbody.tc.cxx      _Networkbody.main.cxx
_Combinerbody.tc.hxx    _Driverbody.tc.hxx      _Networkbody.tc.cxx
_Corebody.ch.cxx        _Firebody.ch.cxx        _Networkbody.tc.hxx
_Corebody.ch.hxx        _Firebody.ch.hxx        _Splitterbody.ch.cxx
_Corebody.cxx           _Firebody.cxx           _Splitterbody.ch.hxx
_Corebody.hxx           _Firebody.hxx           _Splitterbody.cxx
_Corebody.main.cxx      _Firebody.tc.cxx        _Splitterbody.hxx
_Corebody.tc.cxx        _Firebody.tc.hxx        _Splitterbody.tc.cxx
_Corebody.tc.hxx        _Logisticbody.ch.cxx    _Splitterbody.tc.hxx
_DIS.bindirs            _Logisticbody.ch.hxx
_DIS.ch.cxx             _Logisticbody.cxx
_DIS.ch.hxx             _Logisticbody.hxx
_DIS.cxx                _Logisticbody.tc.cxx
_DIS.glob.cxx           _Logisticbody.tc.hxx
```

APPENDIX E

References

[BERTINE] Bertine, Herbert V.; W. B. Elsner; P. K. Verma; K. T. Tewani, Overview of Protocol Testing Programs, Methodologies, and Standards (AT&T Technical Journal, Jan/Feb 1990)

[BUDKO] Budkowski, S.; P. Dembinski, An Introduction to Estelle: A Specification Language for Distributed Systems (Computer Networks and ISDN Systems 14, 1987)

[DIS-STD] Military Standard, Protocol Data Units for Entity Information and Entity Interaction in a Distributed Interactive Simulation (Final Draft) (Institute for Simulation and Training IST-PD-91-1, October 30,1991)

[ISO7498] International Organization for Standardization, Open Systems Interconnection Reference Model (ISO7498, 1984)

[ISO9047] International Organization for Standardization, Estelle: A Formal Description Technique Based on an Extended State Transition Model (ISO9047, August 15, 1989)

[POPE] Pope, A; R. L. Schaffer, The SIMNET Network and Protocols (BBN Systems and Technologies, Report No. 7627, June 1991)

[SIJEL1] Sijelmassi, R.; B. Strausser, The Portable Estelle Translator: An Overview and User Guide (U. S. Department of Commerce, National Institute of Standards and Technology, Technical Report NCSL/SNA-91/1, January 91)

[SIJEL2] Sijelmassi, R.; B. Strausser, The Distributed Implementation Generator: An Overview and User Guide (U. S. Department of Commerce, National Institute of Standards and Technology, Technical Report NCSL/SNA-91/3, January 91)